Lecture Notes in Assembly Language

Short introduction to low-level programming

Piotr Fulmański

Łódź, 12 czerwca 2015

Spis treści

I have no doubt that there are many perfect books and materials about programming in assembler dedicated for Intel x86 family processors.

Unfortunatelly in my opinion none of them is perfect for didactic purposes. Even with a very good book sometimes it's hard to prepare **systematic and logic sequence of material** which can start from very beginning and finish at advanced topics.

My idea behind this book was not to replace existing assembler books but rather complement them by creating book which can be used as one semestr introduction to this field. Working on this all the time I had in my mind didactic aim of it. Material presented in this book should be enought for one semestr of lectures (30 hours) and tutorials (30 hours). The layout of this book is reflected by main aim. Typically books like this start from instroduction to computer history and overview of x86 familly architecture. This is good but not to perform classes: if I have lecture about history what can I do on tutorials? So this is why „theoretical" chapters are at the end of the book.

1. Lecture and tutorial (4 hours): chapter 1: section 1.1.
2. Lecture and tutorial (4 hours): chapter 1: section 1.2–1.4.
3. Lecture and tutorial (4 hours): chapter 1: section 1.5–1.7.
4. Lecture and tutorial (4 hours): chapter 1: section 1.8, chapter 2, chapter 4.

This book is not a reference book, it is an introductory book. It is therefore not suitable by itself to learn how to professionally program in x86 assembly language, as some details have been left out to make the learning process smoother. The point of the book is to help the reader understand how assembly language works.

ROZDZIAŁ 1

Before we begin

1.1 Simple assembler

Before we start, I think, that it's not bad idea to practise with a very simple assembler on very simple machine. Proposed assembler differ a little bit from real assemblers but it's main advantage is simplicity. Based on it, I want to introduce all important concepts.

We use decimal numbers and 5 digit instruction of the following format

```
operation code
|
xxxxx
 |  |
 opernad
```

The list of instruction is as follow

```
0 HLT stop the cpu
1 CPA copy value from memory to accumulator, M -> A
2 STO copy value from accumulator to memory, A -> M
3 ADD add value from specified memory cell to accumulator; result is stored
      in accumulator, M + A -> A
4 SUB subtract from accumulator value from specified memory cell; result
      is stored in accumulator A - M -> A
```

```
5 MUL multiply value from accumulator by value from specified memory cell;
      result is stored in accumulator M * A -> A
6 BRA unconditional branche to instruction located at specified address
7 BRN conditional branche to instruction located at specified address if value
      stored in accumulator is negative
8 BRZ conditional branche to instruction located at specified address if value
      stored in accumulator is equal to zero
```

Accumulator is a dedicated memory cell located in CPU. Such dedicated memory cells are also called **register(s)**. Memory (RAM – *random access memory*) consist of 10000 cells with numbers (addresses) from 0 to 9999. A sign-value representation is used to store negative/positive numbers – when most significante digit is set to 0, the number is positive and negative otherwise (i.e. when different than 0). All arithmetic instructions works on signed numbers. Instruction number 9 is reserved for future extensions.

1.1.1 Excercise 1

Write a program to calculate sum of numbers located in address 6, 7 and 8; result store in address 9.

```
Address Value
0006    20
0007    30
0008    40
0009    result
```

Solution 1.1

```
Address Value       Instruction    Comment
0010    10006       CPA 6          ; A=20
0011    30007       ADD 7          ; A=20+30
0012    30008       ADD 8          ; A=20+30+40
0013    20009       STO 9
0014    00000       HLT
```

1.1.2 Excercise 2

Write a program to calculate for given x a value of polynomial P

$$P(x) = ax + b$$

Address	Value
0004	result
0005	x = 2
0006	a = 3
0007	b = 4

Solution 2.1

Address	Value	Instruction	Comment
0010	10006	CPA 6	; A=3
0011	50005	MUL 5	; A=3*2
0012	30007	ADD 7	; A=3*2+4
0013	20004	STO 4	; Copy A to address 4
0014	00000	HLT	; Stop

1.1.3 Excercise 3

Write a program to calculate for given x a value of polynomial P

$$P(x) = ax^3 + bx^2 + cx + d$$

Address	Value
0004	result
0005	x = 2
0006	a = 3
0007	b = 4
0008	c = 5
0009	d = 6

Solution 3.1

```
Address Value       Instruction  Comment
```

```
0010     10005        CPA 5          ; Copy x to accumulator (A=x)
0011     50005        MUL 5          ; Multiply A by x, A=x^2
0012     50005        MUL 5          ; Multiply A by x, A=x^3
0013     50006        MUL 6          ; Multiply A by a, A=x^3*a
0014     20004        STO 4          ; Copy A to address 4 result
0015     10005        CPA 5          ; Copy x to accumulator (A=x)
0016     50005        MUL 5          ; Multiply A by x, A=x^2
0017     50007        MUL 7          ; Multiply A by b, A=x^2*b
0018     30004        ADD 4          ; Add to A value from result
0019     20004        STO 4          ; Copy A to address 4 result
0020     10005        CPA 5          ; Copy x to accumulator (A=x)
0021     50008        MUL 8          ; Multiply A by c, A=x*c
0022     30004        ADD 4          ; Add to A value from address 4 result
0023     20004        STO 4          ; Copy A to address 4 result
0024     10009        CPA 9          ; Copy d to accumulator (A=x)
0025     30004        ADD 4          ; Add to A value from address 4 result
0026     20004        STO 4          ; Copy A to address 4 result
0027     00000        HLT            ; Stop
```

Solution 3.2

```
Address Value       Instruction  Comment
0010     10005        CPA 5        ; Copy x to accumulator (A=x)
0011     50005        MUL 5        ; Multiply A by x, A=x^2
0012     50005        MUL 5        ; Multiply A by x, A=x^3
0013     50006        MUL 6        ; Multiply A by a, A=x^3*a
0014     20100        STO 100      ; Copy A to address 100
0015     10005        CPA 5        ; Copy x to accumulator (A=x)
0016     50005        MUL 5        ; Multiply A by x, A=x^2
0017     50007        MUL 7        ; Multiply A by b, A=x^2*b
0018     20101        STO 101      ; Copy A to address 101
0019     10005        CPA 5        ; Copy x to accumulator (A=x)
0020     50008        MUL 8        ; Multiply A by c, A=x*c
```

```
0021     20112        STO 102      ; Copy A to address 102
0022     10009        CPA 9        ; Copy d to accumulator (A=d)
0023     30100        ADD 100      ; Add x^3*a to accumulator (A=x^3*a+d)
0024     30111        ADD 101      ; Add x^2*b to accumulator (A=x^3*a+x^2*b+d)
0025     30112        ADD 102      ; Add x*c to accumulator (A=x^3*a+x^2*b+x*c+d)
0026     20004        STO 4        ; Copy A to address 4 result
0027     00000        HLT          ; Stop
```

Solution 3.3

```
Address Value        Instruction    Comment
0010     10006        CPA 6          ; A=a
0011     50005        MUL 5          ; A=ax
0012     30007        ADD 7          ; A=ax + b
0013     50005        MUL 5          ; A=(ax + b)x
0014     30008        ADD 8          ; A=(ax+b)x+c
0015     50005        MUL 5          ; A=((ax+b)x+c)x
0016     30009        ADD 9          ; A=((ax+b)x+c)x+d
0017     20004        STO 4          ; Copy A to address 4 result
0018     00000        HLT            ; Stop
```

1.1.4 Excercise 4

Calculate a^b, where a – integer number, b – integer nonnegative number.

```
Address Value
0001     a
0002     b
```

Solution 4.1

```
Address Value        Instruction  Comment
0001     xxxxx        a
0002     xxxxx        b
0003     00001        1             ; Iterator
```

```
0004    xxxxx       result
0005    10003       CPA 3         ; Copy 1 to A
0006    20004       STO 4         ; Copy A to result
0007    10002       CPA 2         ; Copy b to A
0008    80015       BRZ 15        ; Jump to 15 if A=b is zero
0009    40003       SUB 3         ; Subtract 1 from A=b
0010    20002       STO 2         ; Copy A to b (A=b-1)
0011    10004       CPA 4         ; Copy result to A
0012    50001       MUL 1         ; Multiply A=result by a
0013    20004       STO 4         ; Copy A to result
0014    10002       CPA 2         ; Copy b to A
0014    80007       BRZ 7         ; Jump to 7 if A=b < 0
0015    00000       HLT           ; Stop
```

Solution 4.2

```
Address Value       Instruction  Comment
0001    xxxxx       a
0002    xxxxx       b
0003    00001       1              ; Iterator
0004    00001       result
0005    10002       CPA 2          ; Copy b to A
0006    80013       BRZ 13         ; Jump if b<0
0007    40003       SUB 3          ; Subtract iterator from b
0008    20002       STO 2          ; Save iterator
0009    10004       CPA 4          ; Copy result to A
0010    50001       MUL 1          ; Multiply A by a
0011    20004       STO 4          ; Save as result
0012    60005       BRA 5          ; End loop - jump to the begining of the loop
0013    00000       HLT            ; Stop
```

1.1.5 Excercise 5

Calculate $\frac{a}{b}$, where a is nonnegative and $b > 0$.

```
Address Value
0001    a
0002    b
```

Integer part of division is stored at address 0003, fractional part at address 0004.

Solution 5.1

```
Address Value       Instruction  Comment
0001    xxxxx       a
0002    xxxxx       b
0003    0           result (integer part)
0004    xxxxx       result (fractional part)

0005    00001                    ; Constant value for counter

                                 ; Main part
0006    10001       CPA 1        ; Copy a to A
0007    40002       SUB 2        ; Subtract b
0008    10002       BRN 16       ; If A is negative than go to the end
0009    20001       STO 1        ; Copy a-b to a
                                 ; Increment integer part
0010    10003       CPA 3        ; Copy integer part to A
0011    30005       ADD 5        ; Add constant 1 to A
0012    20003       STO 3        ; Save incremented integer part
                                 ; End increment
0013    60010       BRA 6        ; Go to the begining of the loop

0014    10001       CPA 1        ; Copy a to A
0015    20004       STO 4        ; Copy A=a to fractional part
0016    00000       HLT          ; Stop
```

1.2 Improvements, part I: addressing

Studying the last excercise one can draw the following conclusion

- Instruction list missed instruction to increment or decrement given value. Without this, instead of one instruction, three have to be used, sequence like

  ```
  CPA X ; X - address of the value to increment
  ADD Y ; add value from address Y (very often simply equal to 1)
  STO X ; store X incremented by Y
  ```

  That's why it's good to extend instruction list with two instructions

  ```
  01xxx INC address
  02xxx DEC address
  ```

  In this case we intentionaly avoid the number 9 as the first digit in the code (having in mind that 9 was reserved for extensions) to get more handy „pattern" for instructon numbering – see next part of this chapter.

- Addressing mode used so far is a type of **direct addressing** e.g addressing which uses operand as a value of memory address where actual argument is stored

  ```
  +-code for ADD
  |
  | +-operand (0123)
  | |
  | |       Address     Value
  30123         ...  |        |
    |         (0122) |        |
    +-------> (0123) | 00035  |
              (0124) |        |
                ...  |        |
  ```

  In the example above instruction `ADD` adds value (35) from the addres 0123. In other words, operand points to memory cell and to execute this type of instruction two memory access are needed: one to get instruction and second to get value.

There are situation when it is useful to treat operand not as memory address but as value. For example, when we want to add 5 to value in accumulator, instead of

```
ADD 35 ; we assume that value 5 is stored at address 35
```

more intuitive is to write

```
ADD 5 ; 5 is not an address but value
```

The question is:

– How to distinguish between these two variants?

– When operand treat as address and when as value?

To do this the following convention is used. Notation

```
inst number
```

means: executing instruction `inst` as a value (argument) use number taken from the address `number`, while notation

```
inst (number)
```

means: executing instruction `inst` as a value (argument) use number `number`.

This leads to the second type of addressing – addressing when value is "*in*" instruction and is accessible immediately after instruction read – so called **immediate addressing**.

```
+-code for ADD
|
| +-operand (0123) - value of the argument
| |
| |
30123
```

Introducing this type of addressing entails new codes for instruction because computers like humans have to distinguisg variants of addressing

```
            Direct addressing   Immediate addressing

Human       ADD 35              ADD (5)


Computer    30035               92305
```

```
9xxxx - to indicate extension of basic instruction set
x2xxx - addressing mode (2 for immediate, 1 byte length)
xx3xx - code for addition in basic instructions set
xxxx5 - immediate value - notice that this value is stored "in" instruction
```

Notice that value 5 is stored "in" instruction and there is no need of the next memory access – it means that this type of instruction is faster. Unfortunately there is a problem: what about instruction like

```
ADD (128)
```

It is not possible to squeeze value 128 and put "into" instruction like in case of value 5. The solution for this is to put another code for addition which assumes that value of the argument is put just after instruction, like in the following example

```
address   value
x         93300 - add
x + 1     00128 - value for add of code 9230
```

```
9xxxx - to indicate extension of basic instruction set
x3xxx - addressing mode (3 for immediate, 2 byte length)
xx3xx - code for addition in basic instructions set
```

This is in some sens a mixture of direct and immediate addresing: we have two memory access (one for instruction and the second to get value) but argument is always located next to instruction (after instruction) – we could say that we immediately know where the argument is.

1.2.1 Excercise 6

Calculate the dot product (sometimes scalar product or inner product) of two vectors of length 10.

Solution 6.1

You can try to find a solution but it seems to be unsolvable.

1.3 Improvements, part II: indirect addressing

- This problem seems to be unsolvable without concept of memory **indirect addressing**. Notation

  ```
  inst addr
  ```

  means: executing instruction `inst` as an address of the argument use `addr`, while notation

  ```
  inst [addr]
  ```

  means: executing instruction `inst` as an address of the argument use value taken from the address `addr`.

  ```
  +-code for ADD [x] ->--+
  |                      +->-- finally: ADD [6] and it adds 123
  |   +-operand (6) -->--+              to acumulator
  |   |
  |   |       Address     Value
  94306           ...  |        |
      |        (0005)  |        |
      +------> (0006)  | 00009  | ---+
               (0007)  |        |    |
                  ...  |        |    |
               (0009)  | 00123  | <--+
                  ...  |        |

  9xxxx - to indicate extension of basic instruction set
  x4xxx - addressing mode (4 for indirect)
  ```

```
xx3xx - code for addition in basic instructions set
```

We can think about `[ ]` "operator" as an substitution: having instruction `inst [addr]` take value from the address `addr`, name it `val`, substitute `[addr]` by `val` and finally execute instruction `inst val`.

Notice that in instruction set defined so far we have a mixture of addressing. For example `ADD xyz` uses direct addressing (we add to the value stored in the accumulator value taken from address `xyz`). On the other hand `STO abc` means: save value from accumulator at address `abc`. In this case immediate addressing is used – destination address is known just after instruction is read and there is no need for next memory access (like for `ADD`).

Taking into account all of the above an extension of the instruction set could be defined as follow

Instruction set is not correct!!!

General

```
00000 HLT stop the cpu
```

Direct (one-byte)

```
900xx INC increment  value in memory at specified address
909xx DEC decrement  value in memory at specified address
1xxxx CPA copy value from memory to accumulator, M -> A
902xx STO copy value from accumulator to memory, A -> M
3xxxx ADD add value from specified memory cell to accumulator; result is stored
          in accumulator, M + A -> A
4xxxx SUB subtract from accumulator value from specified memory cell; result
          is stored in accumulator A - M -> A
5xxxx MUL multiply value from accumulator by value from specified memory cell;
          result is stored in accumulator M * A -> A
906xx BRA unconditional branche to instruction located at specified address
907xx BRN conditional branche to instruction located at specified address if value
          stored in accumulator is negative
908xx BRZ conditional branche to instruction located at specified address if value
          stored in accumulator is equal to zero
```

Direct (two-byte)

```
91000 xxxxx INC

91900 xxxxx DEC

91100 xxxxx CPA

91200 xxxxx STO

91300 xxxxx ADD

91400 xxxxx SUB

91500 xxxxx MUL

91600 xxxxx BRA

91700 xxxxx BRN

91800 xxxxx BRZ
```

Immediate (one-byte)

```
01xxx INC increment  value in memory at specified address

02xxx DEC decrement  value in memory at specified address

921xx CPA

2xxxx STO copy value from accumulator to memory, A -> M

923xx ADD

924xx SUB

925xx MUL

6xxxx BRA unconditional branche to instruction located at specified address

7xxxx BRN conditional branche to instruction located at specified address if value
          stored in accumulator is negative

8xxxx BRZ conditional branche to instruction located at specified address if value
          stored in accumulator is equal to zero
```

Immediate (two-byte)

```
93000 xxxxx INC

93900 xxxxx DEC

93100 xxxxx CPA

93200 xxxxx STO
```

```
93300 xxxxx ADD

93400 xxxxx SUB

93500 xxxxx MUL

93600 xxxxx BRA

93700 xxxxx BRN

93800 xxxxx BRZ
```

Indirect (one-byte)

```
-- 940xx INC

-- 949xx DEC

941xx CPA

-- 942xx STO

943xx ADD

944xx SUB

945xx MUL

-- 946xx BRA

-- 957xx BRN

-- 948xx BRZ
```

Indirect (two-byte)

```
-- 95000 xxxxx INC

-- 95900 xxxxx DEC

95100 xxxxx CPA

-- 95200 xxxxx STO

95300 xxxxx ADD

95400 xxxxx SUB

95500 xxxxx MUL

-- 95600 xxxxx BRA

-- 95700 xxxxx BRN

-- 95800 xxxxx BRZ
```

Solution 6.2 – second approach (correct)

```
Address Value      Instruction
0001    00010       ; Address of the first component of vector 1
0002    00020       ; Address of the first component of vector 2
0003    00000       ; Result
0004    00010       ; n - length of vector
...
0010    xxxxx       ; First component of vector 1
...
0019    xxxxx       ; Last component of vector 1
0020    xxxxx       ; First component of vector 2
...
0029    xxxxx       ; Last component of vector 2

0030    10004       CPA 4
0031    80040       BRZ 40
0032    94101       CPA [1]
0033    94702       MUL [2]
0034    30003       ADD 3
0035    20003       STO 3
0036    92001       INC 1
0037    92002       INC 2
0038    92904       DEC 4
0039    60030       BRA 30
0040    00000       HLT
```

Solution 6.3 – like second approach but incorrect

Previous solution is correct, but when the code is reallocated into other place in the memory, symbolic names stays the same, but the binary code changes. In the realocated code in the example below (all the code was shifted by 100) symbolic names are correct but their addresses are not.

```
Address Value      Instruction
```

```
0101            address of the first component of vector 1
0102            address of the first component of vector 2
0103            result
0104            n - length of vector
...
0110            first component of vector 1
...
0119            last component of vector 1
0120            first component of vector 2
...
0129            last component of vector 2

0130            CPA 104
0131            BRZ 140
0132            CPA [101]
0133            MUL [102]
0134            ADD 103
0135            STO 103
0136            INC 101
0137            INC 102
0138            DEC 104
0139            BRA 130
0140            HLT
```

Explanation for this is obvious when binary codes for instructions is used.

```
Address Value   Instruction
0101    00020    address of the first component of vector 1
0102    00030    address of the first component of vector 2
0103    00000    result
0104    00010    n - length of vector
...
0110    xxxxx    first component of vector 1
```

```
...
0119    xxxxx       last component of vector 1
0120    xxxxx       first component of vector 2
...
0129    xxxxx       last component of vector 2

0130    10014       CPA 104
0131    80052       BRZ 142
0132    95100       CPA [101]
0133    00101
0134    95500       MUL [102]
0135    00102
0136    30103       ADD 103
0137    20103       STO 103
0138    01101       INC 101
0139    01102       INC 102
0140    02104       DEC 104
0141    60130       BRA 130
0142    00000       HLT
```

Explanation is as follow: not all instructions are one byte length. That's why simple change in the code entails "shift" of all instructions. Code

```
CPA [1]
```

generates machine code different than

```
CPA [101]
```

In the first case we have

```
Address Value      Instruction
x       94101       CPA [1]
```

and the second

```
Address Value      Instruction
x       95100       CPA [101]
x+1     00101
```

1.4 Improvements, part III: labels

- Problems with variable length instructions could be solved by the release of the explicit addresses usage. Instead of them, **labels** are used to indicate "places" in the memory. With this an "universal" solution of (1.2.1) could be as follow

Solution 6.4

```
Label /   Value /
Address   Instruction    Comment
.data 0                 ;start data block at address 0
v1:       xxxx          ;first component of vector 1
           ...
          xxxx          ;last component of vector 1
v2:       xxxx          ;first component of vector 2
           ...
          xxxx          ;last component of vector 2

a_v1:       v1          ;address of the first component of vector 1
a_v2:       v2          ;address of the first component of vector 2
result:      0          ;result
vec_len:    10          ;n - length of vector

.code 50                ;start code block at address 50
begin:     CPA vec_len
           BRZ end
           CPA [a_v1]
           MUL [a_v2]
           ADD result
           STO result
           INC a_v1
           INC a_v2
           DEC vec_len
           BRA begin
end:      HLT
```

1.4.1 Excercise 7: find substring in a string

Write a program to search substring in a string.

Solution 7.1

```
.data 1
string: 3 2 1 1 2 3 3 2 1 0
substr: 1 2 3 0
ptrStr: string
ptrSubStr: substr
isSubStr: 0
counter: 0

.code 200
                ; main
begin:
CPA [ptrSubStr] ; Take substr[ptrSubStr] element from the substring
BRZ end         ; If zero than substring is empty (or ended), so go to end
CPA [ptrStr]    ; Take string[ptrStr] element from the string
BRZ end         ; If zero than string is empty (or ended), so go to end
SUB [ptrSubStr] ; Subtract substr[ptrSubStr] element from substring
BRZ equal       ; If zero than both elements are equal
                ; If not then
CPA counter     ; If counter is nonzzero then
BRZ begin
                ; move string pointer,
                ; reset substring pointer,
                ; reset counter
INC ptrStr      ; Move string pointer
CPA (substr)    ; Reset substring pointer
STO ptrSubStr
CPA (0)         ; Reset counter
STO counter
BRA begin       ; Go to the beginning of the loop

equal:
INC ptrStr      ; Move string pointer
INC ptrSubStr   ; Move substring pointer
INC counter
BRA begin

end:
CPA counter     ; If counter is nonzero then
BRZ stop
CPA ptrStr      ; set isSubStr to point the beginning of the
SUB counter     ; substr in a string
STO isSubStr

stop:
HLT
```

Solution 7.2

This solution is not finished yet.

```
begin:
CPA [a_p]
SUB [a_c]
INC a_c
```

```
BRZ equal
CPA c_len
BRZ end
DEC c_len
CPA start_p
STO a_p
CPA p_len
STO iter_p
BRA begin

equal:
INC a_p
DEC iter_p
CPA iter_p
BRZ result
BRA begin

result:
CPA [a_c]
SUB p_len
STO znaleziono

end:
HLT
```

1.4.2 Excercise 8: improved polynomial

Solve the problem from the exercise 1.1.3 using solution from 1.1.4.

Solution 8.1

```
.data 0
; local variables for main code
coef:      A   ; coefficient A -- put an exact value here
```

```
           B
           C
           D
pow:      pA   ; power for coef. A -- put an exact value here
          pB
          pC
          pD
varX:      X   ; put an exact value as X

coefI:  coef   ; put as value of coef. iterator address of A
powI:    pow   ; put as value of power iterator address of pA
result:    0
counter:   4   ; indicate the number of components

;local variables for power subprogram

bas:       0
power:     0
resT:      0

.code 20
;main
begin: CPA varX       ; prepare local data for subprogram
       STO base
       CPA [powI]
       STO power
       BRA powerStart ; call subprogram
loop:  CPA resT       ; return from subprogram - we have a result of base^pow
       MUL [coefI]
       INC powI
       INC coefI
       ADD result
       STO result
       DEC counter
       CPA counter
       BRN end
       BRA begin
end:   HLT

;subprogram
powerBegin:   CPA (1)
              STO resT
powerLoop:    CPA power
              BRZ powerEnd
              DEC power
              CPA resT
              MUL base
              STO resT
              BRA powerLoop
powerEnd:     BRA loop
```

1.5 Improvements, part IV: flag register

Consider now a following sequence of instructions we used in previous programs

```
DEC counter
CPA counter
BRN end
```

The idea behind this is very simple: decrease variable (an iterator) and if it is negative (or zero if we use BRZ) then jump somewhere. The strange thing is that after we decrease our counter by DEC we have to load it into accumulator because jump instructions can work only on values stored in accumulator.

We can solve this if we take a following agreement: *every numerical instruction (INC, DEC, ADD, SUB, MUL) after execution sets some dedicated memory cells (registers) — called **flags** — located in CPU (like accumulator is located in cpu):*

- *ZF:* Zero Flag *this flag is set to 1 if last instruction's result is equal to zero, othervise is set to 0;*
- *NF:* Negative Flag *this flag is set to 1 if last instruction's result is neqative, othervise is set to 0;*

Now we can introduce new jump instructions set

- `BRNF`: if last instruction's result is negative
- `BRZF`: if last instruction's result is equal to zero

With this sequence

```
DEC counter
CPA counter
BRN end
```

can be substituted by more intuitive sequence

```
DEC counter
BRNF end
```

1.6 Improvements, part V: the stack

That's right – we can solve the problem (1.4.2) the way we proposed, but the method used to passing argument is far from perfection. Better choice is to use some data structure which help us to keep a correct order of the arguments – this is how we reach the concept of stack.

Generally speaking in computer science, a stack or LIFO (*last in, first out*) is an abstract data type that serves as a collection of elements, with two principal operations:

- **push** – adds an element to the collection;
- **pop** – removes the last element that was added.

The term LIFO stems from the fact that, using these operations, the last element "popped off" a stack in series of pushes and pops is the first element that was pushed in the sequence. This is equivalent to the requirement that the push and pop operations occur only at one end of the structure, referred to as the **top of the stack**. The nature of the pop and push operations means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition. Therefore, the lower elements are those that have been on the stack the longest.

If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an *overflow state* – which results a well known runtime message: *Stack Overflow*.

Notice one very important thing: stack in computers growth in direction of lower addresses. It means that if element y is above x in a stack the address of y is lower than x.

```
higher addresses


99999
...
xxxxx   x <-- base of the stack
xxxxx-1 ..
xxxxx-2 ..
xxxxx-3 y <-- top of the stack
...
00000


lower addresses


direction of stack growth
```

To keep things working we also have to introduce two new registers in our CPU

- BP – to keep information about base of the stack,
- SP – to keep information about top of the stack.

with instruction

```
PUSH
POP
```

For example

```
PUSH     -- add an element from accumulator to the stack;
PUSH 5   -- add an element from address 5 to the stack;
PUSH (5) -- add value 5 to the stack;
POP      -- removes the last element that was added to the stack and
            put it into accumulator;
POP 5    -- removes the last element that was added to the stack and
            put it at address 5.
```

1.6.1 Excercise 12

Set of examples is not correct Use stack to write a program which adds two arguments and saves result in a variable.

```
a: - first number
b: - second number
result: sum of a and b
```

Solution 12.1 – an introduction: solution without a stack

```
.data 0
a: 2
b: 5
result: 0

.code 10
         BRA dodaj
return:  HLT

dodaj:   CPA a
         ADD b
         STO result
         BRA return
```

Solution 12.2 – solution without a stack (incorrect)

```
Sketch of the second (incorrect) program

numbers: 5 7 8 12
tmp: 0
result: 0 0
addrFrom: numbers
addrResult: result
counter: 2

; Some code to call add

add:
CPA [addrFrom]
INC addrFrom
ADD [addrFrom]
STO [addrResult]
INC addrFrom
INC addrResult
```

Solution 12.3 – solution with a stack (incorrect)

```
Sketch of the third (incorrect) program

result:
tmp:

PUSH (5)
PUSH (7)
CALL add
POP result
```

```
; Do something with a result

PUSH (8)
PUSH (12)
CALL add
POP result

; Do something with a result

add:
POP
STO tmp
POP
ADD tmp
PUSH
RET
```

Solution 12.4 – solution with a stack

```
.data 0
a: 2
b: 5
wynik: 0

.code 10
start: PUSH wynik
       PUSH a
       PUSH b
       CALL dodaj
       POP wynik
       HLT

dodaj: CPA [SP + 1]
```

```
ADD [SP + 2]
STO [SP + 3]
RET 2
```

1.7 Improvements, part VI – function stack frame

The solution we found for *improved polynomial with a stack* (excercise ??) is almost perfect with the exception of one unsolved problem: *how do we know to which address should we return?* The problem is that we assumed that called function knows which function or part of the code was a caller – in our case, "main" code – and we hardcoded this value in our function. What if we try to call function from completely different place, for example other function? We return to "main" code which wouldn't be correct.

That is why functions (subrutines) are frequently set up with a **stack frame** to know where to return and to allow access to both function parameters, and automatic function variables. The idea behind a stack frame is that each subroutine can act independently of its location on the stack, and each subroutine can act as if it is the top of the stack. In other words, each subrutine can act as it would be the only subrutine in a code.

When a function is called, a new stack frame is created at the current SP location. A stack frame acts like a partition on the stack. All items from previous functions are higher up on the stack, and should not be modified. Each current function has access to the remainder of the stack, from the stack frame until the end of the stack page.

So how it works? When we want to call a function we have to perform a following sequence of instruction

```
...
CALL function ; Jump to 'function' address and push on the stack
              ; address of the next instruction
XXX           ; Next instruction
...
function:
              ; Prepare stack to be safely use in our function
     PUSH BP  ; Save the current value of BP on the stack
              ; Move SP to BP (set BP as equal to SP)
```

```
CPA SP   ; Read current top of the stack
STO BP   ; BP now points to the top of the stack
...
...      ; Exact function code starts
...      ; Do what you want to do
...
         ; Restore the stack
         ; Move BP to SP (set SP as equal to BP)
CPA BP   ; Read current base of the stack
STO SP   ; SP now points to the top of the stack
POP BP   ; Restore value of BP saved at the beginning
RET      ; Pop value from the stack and jump to this address
```

In the above code we have two new instructions:

- `CALL` – push on the stack address of the next instruction following this `CALL` instruction;
- `RET` – pop value from the stack and treating it as an address jump to instruction at this address.

Above sequence of instruction results in the following stack changes (we stop at the time when exact function code starts):

```
Initial   After              After              After move
stack     CALL               PUSH BP            SP to BP


BP -> A1  BP -> A1           BP -> A1                     A1
      ...       ...                ...                    ...
SP -> A2        A2                 A2                     A2
          SP -> XXX Addr.          XXX Addr.              XXX Addr.
                             SP -> BP           (BP, SP) -> BP
```

Here is a representation of the stack at the time when exact function code starts:

```
Frame stack:


higher addresses
```

```
Address   Value (Meaning)

BP + 1  (return address)
BP      (old BP value)

lower addresses

stack growth
```

When we want to call a function with some *arguments* and *local variables* very similar schema is used.

```
...
PUSH ARG_N
...
PUSH ARG_1
CALL function ; Jump to 'function' address and push on the stack
              ; address of the next instruction
XXX           ; Next instruction
...
function:
              ; Prepare stack to be safely use in our function
     PUSH BP  ; Save the current value of BP on the stack
              ; Move SP to BP (set BP as equal to SP)
     CPA SP   ; Read current top of the stack
     STO BP   ; BP now points to the top of the stack
              ; "allocate" space for the M local variables
     CPA SP   ; Read current SP
     SUB M    ; Move SP down (allocate space for M variables)
     STO SP   ; Save SP
     ...
     ...      ; Exact function code starts
```

```
    ...      ; Do what you want to do
    ...
             ; Restore the stack
             ; Move BP to SP (set SP as equal to BP)
    CPA BP   ; Read current base of the stack
    STO SP   ; SP now points to the top of the stack
    POP BP   ; Restore value of BP saved at the beginning
    RET N    ; Pop value from the stack and jump to this address
             ; but before the jump the stack is lowered by N
```

One thing which should be explained is `RET N` instruction. This instruction pops `N` elements from the stack and next jumps to instruction just after CALL. Saying the truth we don't carre about popped elements so technicaly speaking `RET N` instruction does not pop `N` element from the stack but simply move stack pointer by `N` to point lower elements – the easiest way to do it is simply subtract from `SP` value `N`.

Generaly speaking we have a following (function) frame on a stack every time we call a function (at the time when exact function code starts):

```
Frame stack:

higher addresses

Address   Value (Meaning)

BP + 1 + N (Nth function argument)
...
BP + 1 + 1 (1st function argument)
BP + 1     (return address)
BP         (old BP value)
BP - 1     (1st local variable)
...
BP - 1 - M (Mth local variable)
```

lower addresses

stack growth

Solution 12.5 – solution with a stack and frame stack

```
.data 0
a: 2
b: 5
wynik: 0

.code 10
start: PUSH wynik
       PUSH a
       PUSH b
       CALL dodaj
       POP wynik
       HLT

dodaj:
       ; Init the stack
       PUSH BP
       CPA SP
       STO BP

       ; Make some computations
       CPA [BP + 2]
       ADD [BP + 3]
       STO [BP + 4]

       ; Clean the stack
       CPA BP
       STO SP
```

```
        POP BP

        RET 2
```

1.8 Finall excercises

1.8.1 Excercise 13

Solve once again the problem from the exercise ?? using improved stack.

Solution 31.1

1.8.2 Excercise 14

Program porządkujący liczby.

Solution 14.1

1.8.3 Excercise 15

Program znajdujący najmniejszą i najwieksza sposrod 4 liczb.

Solution 15.1

1.8.4 Excercise 16

Write a program to calculate absolute value for given value v.

```
Address Value
1000    v
1001    result - abs(v)
```

Solution 16.1

1.8.5 Excercise 17

Find the greates comon divisors of two positive numbers. There are two possible approach to this problem.

Using prime factorizations Greatest common divisors (nwd) can in principle be computed by determining the prime factorizations of the two numbers and comparing factors. To compute, for example, $\mathrm{nwd}(16, 36)$, we find the prime factorizations $16 = 2 \cdot 2 \cdot 2 \cdot 2$ and $36 = 2 \cdot 2 \cdot 3 \cdot 3$. Notice that the "intersection" of the two expressions, which is $2 \cdot 3$ is $\mathrm{nwd}(16, 36) = 6$. In practice, this method is only feasible for small numbers; computing prime factorizations in general takes far too long.

Using Euclid's algorithm A much more efficient method is the Euclidean algorithm, which uses a division algorithm such as long division in combination with the observation that the nwd of two numbers also divides their difference. If the arguments are both greater than zero then the algorithm can be written as follows

$$\mathrm{nwd}(a, a) = a$$
$$\mathrm{nwd}(a, b) = \mathrm{nwd}(a - b, b), \text{ if } a > b$$
$$\mathrm{nwd}(a, b) = \mathrm{nwd}(a, b - a), \text{ if } b > a$$

Solution 17.1

ROZDZIAŁ 2

First program

2.1 Compiling, linking...

This section is not correct!!! We postpone detailed discussion about compiling and linking to further chapters. Now we want to introduce only some basic concepts behind both processes to allow us use them while we will make our first program.

2.1.1 Compiler and compiling

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code).[1] The most common reason for converting a source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. More generally, compilers are a specific type of translators.

Compilation refers to the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' file. This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled. For instance, if you compile (but don't link) three separate files, you will have three object files created as output, each with the name ¡filename¿.o or ¡filename¿.obj (the extension will depend on your compiler). Each of these files contains a translation of your source code file into a machine language file – but you can't run them yet! You need to turn them into executables your operating system can use. That's where the linker comes in.

2.1.2 Linker and linking

In computer science, a linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.

A simpler version that writes its output directly to memory is called the loader, though loading is typically considered a separate process.[1]

Linking refers to the creation of a single executable file from multiple object files. In this step, it is common that the linker will complain about undefined functions (commonly, main itself). During compilation, if the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file. If this isn't the case, there's no way the compiler would know – it doesn't look at the contents of more than one file at a time. The linker, on the other hand, may look at multiple files and try to find references for the functions that weren't mentioned.

2.1.3 Summary

To understand linkers, it helps to first understand what happens "under the hood" when you convert a source file (such as a C or C++ file) into an executable file (an executable file is a file that can be executed on your machine or someone else's machine running the same machine architecture).

Under the hood, when a program is compiled, the compiler converts the source file into object byte code. This byte code (sometimes called object code) is mnemonic instructions that only your computer architecture understands. Traditionally, these files have an .OBJ extension.

After the object file is created, the linker comes into play. More often then not, a real program that does anything useful will need to reference other files. In C, for example, a simple program to print your name to the screen would consist of:

printf(Hello Christina); When the compiler compiled your program into an obj file, it simply put a reference to the printf function. The linker resolves this reference. Most programming languages have a standard library of routines to cover the basic stuff expected from that language. The linker links your OBJ file with this standard library. The linker can also link your OBJ file with other OBJ files. You can create other OBJ files that have functions that can be called by another OBJ file. The linker works, almost like a word processor's copy and paste. It "copies" out all the necessary functions your program references and creates a single executable. Sometimes other libraries that are copied out are dependent on yet other OBJ or library files. Sometimes a linker has to get pretty recursive to do its job.

Note that not all operating systems create a single executable. Windows, for example, uses DLL's that keep all these functions together in a single file. This reduces the size of your executable, but makes your executable dependent on these specific DLLs. DOS used to use things called Overlays (.OVL files). This had many purposes, but one was to keep commonly used functions together in 1 file (another purpose it served, in case you're wondering, was to be able to fit large programs into memory. DOS has a limitation in memory and overlays could be "unloaded" from memory and other overlays could be "loaded" on top of that memory, hence the name, "overlays"). Linux has shared libraries, which is basically the same idea as DLL's (hard core Linux guys I know would tell me there are MANY BIG differences).

Hope this helps you understand!

2.2 32-bit basic stand alone program

2.2.1 Code for NASM

../programs/first_program/hello.asm

```
;  This program demonstrates basic text output to a screen.
;  No "C" library functions are used.
;  Calls are made to the operating system directly. (int 80 hex)
;
; assemble:      nasm -f elf hello.asm
; link:          ld hello.o -o hello
; run:           ./hello
; output is:     Hello World

section .data                ; Data section

text:    db "Hello World!", 10  ; The string to print, 10=LF
len:     equ $-text          ; "$" means "here"
                             ; len is a value, not an address
```

```
section .text                  ; Code section

global  _start                 ; Make label available to linker
                               ; We must export the entry point to the ELF linker or
                               ; loader. They conventionally recognize _start as their
                               ; entry point. Use ld -e foo to override the default.

_start:                        ; Standard  ld  entry point
        mov     edx, len       ; arg3: length of string to print
        mov     ecx, text      ; arg2: pointer to string
        mov     ebx, 1         ; arg1: where to write, so called file handler
                               ; in this case stdout (screen)
        mov     eax, 4         ; System call number (sys_write)
        int     0x80           ; Interrupt 80 hex, call kernel

; Exit
        mov     ebx, 0         ; Exit code, 0=normal
        mov     eax, 1         ; System call number (sys_exit)
        int     0x80           ; Interrupt 80 hex, call kernel
; End of the code
```

Verify correctnes of the code by assembling it

```
nasm -f elf hello.asm
```

linking

```
ld hello.o -o hello
```

and finally running

```
./hello
```

If no errors were raported the result should be as follow

```
fulmanp@fulmanp-k2:~/assembler$ ./hello
Hello World!
```

If you want to know more... 2.1 (Making 32-bit code on 64-bit system with NASM)**.** *When you try to make 32-bit program on 64-bit system assemby it as previously*

```
nasm -f elf hello.asm
```

but link as

```
ld -m elf_i386 hello.o -o hello
```

Such a program is a 32-bit program, which can be verified by `readelf` *Unix command*

```
fulmanp@fulmanp-k2:~/assembler$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048080
  Start of program headers:          52 (bytes into file)
  Start of section headers:          216 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 3
```

Presented code, without any changes, can be also assembled as 64-bit program[*] *with*

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 hello.asm
fulmanp@fulmanp-k2:~/assembler$ ld hello.o -o hello
```

[*]Note that this is not real 64-bit program.

```
fulmanp@fulmanp-k2:~/assembler$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4000b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          264 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         6
  Section header string table index: 3
```

If you want to know more... 2.2 (Getting content of assembled file)**.** *If you wander about content of assembled or linked file you can use* `xxd` *Unix command do dump these files in "readable" format*

```
fulmanp@fulmanp-k2:~/assembler$ xxd hello.o
0000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
0000010: 0100 0300 0100 0000 0000 0000 0000 0000  ................
0000020: 4000 0000 0000 0000 3400 0000 0000 2800  @.......4.....(.
0000030: 0700 0300 0000 0000 0000 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
0000060: 0000 0000 0000 0000 0100 0000 0100 0000  ................
0000070: 0300 0000 0000 0000 6001 0000 0d00 0000  ........`.......
0000080: 0000 0000 0000 0000 0400 0000 0000 0000  ................
0000090: 0700 0000 0100 0000 0600 0000 0000 0000  ................
00000a0: 7001 0000 2200 0000 0000 0000 0000 0000  p..."...........
00000b0: 1000 0000 0000 0000 0d00 0000 0300 0000  ................
00000c0: 0000 0000 0000 0000 a001 0000 3100 0000  ............1...
00000d0: 0000 0000 0000 0000 0100 0000 0000 0000  ................
00000e0: 1700 0000 0200 0000 0000 0000 0000 0000  ................
00000f0: e001 0000 7000 0000 0500 0000 0600 0000  ....p...........
0000100: 0400 0000 1000 0000 1f00 0000 0300 0000  ................
0000110: 0000 0000 0000 0000 5002 0000 1b00 0000  ........P.......
0000120: 0000 0000 0000 0000 0100 0000 0000 0000  ................
0000130: 2700 0000 0900 0000 0000 0000 0000 0000  '...............
0000140: 7002 0000 0800 0000 0400 0000 0200 0000  p...............
0000150: 0400 0000 0800 0000 0000 0000 0000 0000  ................
0000160: 4865 6c6c 6f20 576f 726c 6421 0a00 0000  Hello World!....
0000170: ba0d 0000 00b9 0000 0000 bb01 0000 00b8  ................
0000180: 0400 0000 cd80 bb00 0000 00b8 0100 0000  ................
0000190: cd80 0000 0000 0000 0000 0000 0000 0000  ................
00001a0: 002e 6461 7461 002e 7465 7874 002e 7368  ..data..text..sh
00001b0: 7374 7274 6162 002e 7379 6d74 6162 002e  strtab..symtab..
00001c0: 7374 7274 6162 002e 7265 6c2e 7465 7874  strtab..rel.text
00001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001f0: 0100 0000 0000 0000 0000 0000 0400 f1ff  ................
0000200: 0000 0000 0000 0000 0000 0000 0300 0100  ................
0000210: 0000 0000 0000 0000 0000 0000 0300 0200  ................
0000220: 0b00 0000 0000 0000 0000 0000 0000 0100  ................
0000230: 1000 0000 0d00 0000 0000 0000 0000 f1ff  ................
0000240: 1400 0000 0000 0000 0000 0000 1000 0200  ................
0000250: 0068 656c 6c6f 2e61 736d 0074 6578 7400  .hello.asm.text.
```

```
0000260: 6c65 6e00 5f73 7461 7274 0000 0000 0000  len._start......
0000270: 0600 0000 0102 0000 0000 0000 0000 0000  ................
```

Notice that this is not real 64-bit program because you still use 32-bit registers and function call convention – compare with section 2.3

Knowing that it works, now it's a time to explain why it works. Let's study the code line by line.

- Character `;` starts comment which and extend to the end of the line.
- `section .data`

  Start of the data section; mixing data and code is not allowed.
- `text: db "Hello World!", 10`

  Definition of the text to print ended by newline character(s). In this case we have code for Linux operating system so we use line feed character (LF, decimal code: 10).
- `len: equ $ - text`

  Definition of the constant value equal to: current address ($) minus address of the first element of variable `text` – this should be equal to the length of the text we are going to print. Notice that `len` is a value (constant of the compilation), not an address. If you prefer variables replace this line by `len dd $-text`
- `section .text`

  Start of the code (program) section; mixing data and code is not allowed.
- `global _start`

  Make label available to linker. We must export the entry point to the ELF linker or loader. They conventionally recognize `_start` as their entry point. Use `ld -e foo` to override the default.
- `_start:`

  Label; standard `ld` entry point.
- `mov edx, len` (or `mov edx, [len]` if you prefere variables than constants)

  Move (copy, insert, put) to EDX register (RDX)[†] length of the text to print – this would be

[†]EDX is a 32-bit register while RDX – 64-bit; in the whole book brackets are used to ditinguish 32-bit and 64-bit registers when both are in one sentence.

a third argument of the function we are going to call. In the first case length is a constant, in the second we take it from variable. Talking about `mov` notice that copying data from one memory cell to the other is not allowed

```
mov [dest], [src] ; this is not allowed
```

- `edx`
  We deferred discussion about registers untill section 14. Here we have to mention basics about registers so we could work throught next few sections. Generally speaking now we will use set of registers whose names are created with the following pattern:

  ```
  <register_name> ::= <name_prefix><letter><name_suffix>

  <letter> ::= A | B | C | D | E
  <name_prefix> ::= R | E
  <name_suffix> ::= X | H | L
  ```

  where, for example, correct register names for letter A are

  ```
  RAX, EAX, AX, AH, AL
  ```

  In this case we are talking about register A and it's different parts and sizes

  ```
  6            33       11   00   0
  3            21       65   87   0
  |             |        |   ||   |
  |             |        |.AH||.AL|   AH and  AL:  8 bits
  |             |        |...AX...|           AX: 16 bits
  |             |......EAX.......|          EAX: 32 bits
  |...........RAX...............|          RAX: 64 bits
  ```

- `mov ecx, text`
  Copy to ECX register (RSI) address of the first element of the text – this would be a second argument of the function we are going to call.

- `mov ebx, 1`

  Copy to EBX register (RDI) value 1 – this would be a first argument of the function we are going to call, so called file handler, indicating where to write (in this case stdout i.e. screen).

- `mov eax, 4`

  Copy to EAX register (RAX) value 4 (1). This is a number of Linux function (`sys_write`) we are going to call. Notice that these numbers are different for different architectures and operation systems.

- `int 0x80` (`syscall`)

  Interrupt to call system function selected by EAX register (RAX). In this case this is `sys_write` function which takes three arguments in registers EBX, ECX and EDX (RDI, RSI and RDX).

  32-bit system function takes at most 6 arguments from registers EBX, ECX, EDX, ESI, EDI and EBP. EAX is used to specify the number of a system function we are going to call.

  64-bit system function takes at most 6 arguments from registers RDI, RSI, RDX, R10, R8, R9. RAX is used to specify the number of a system function. Values in registers RCX and R11 are destroyed.

  More precisely: INT means interrupt, and the number 0x80 is the interrupt number. An interrupt „transfers” the program flow to whomever is handling that interrupt. In Linux, 0x80 interrupt handler is the kernel, and is used to make system calls to the kernel by other programs.

  The kernel is notified about which system call the program wants to make, by examining the value in the register EAX. Each system call have different requirements about the use of the other registers. For example, a value of 1 in EAX means a system call of `exit()`; in this case the value in EBX holds the value of the status code for `exit()`.

- `mov ebx, 0`

  Copy to EBX register (RDI) value 0 – this would be a first argument of the function we are going to call, so called errorlevel, indicating whether program was terminated correctly or not (0 means that everything was all right and program terminates normally).

- `mov eax, 1` Copy to EAX register (RAX) value 1 (60). This is a number of Linux function (`sys_exit`) we are going to call to terminate program.

- `int 0x80` (`syscall`)

  Interrupt to call system function selected by EAX register (RAX).

Sometimes, especially at the beginning of contact with the assembler, it's good to generate and examine listfile.

Explain what is list file

For the above code, the content of listfile is generated with command

```
nasm -l hello.lst  hello.asm
```

and returns the following output

```
     1                                  ;  This program demonstrates basic text output to
     2                                  ;  No "C" library functions are used.
     3                                  ;  Calls are made to the operating system directl
     4                                  ;
     5                                  ; assemble:     nasm -f elf hello.asm
     6                                  ; link:         ld hello.o -o hello
     7                                  ; run:          ./hello
     8                                  ; output is:    Hello World!
     9
    10                                  section .data               ; Data section
    11
    12 00000000 48656C6C6F20576F72-     text    db "Hello World!", 10   ; The string to pr
    13 00000009 6C64210A
    14                                  len     equ $-text          ; "$" means "here"
    15                                                              ; len is a value, not
    16
    17                                  section .text               ; Code section
    18
    19                                  global  _start              ; Make label available
    20                                                              ; We must export the e
    21                                                              ; loader. They convent
    22                                                              ; entry point. Use ld
```

```
23
24                               _start:                    ; Standard  ld  entry 
25 00000000 66BA0D000000               mov     edx,len      ; arg3: length of stri
26 00000006 66B9[00000000]             mov     ecx,text     ; arg2: pointer to str
27 0000000C 66BB01000000               mov     ebx,1        ; arg1: where to write
28 00000012 66B804000000               mov     eax,4        ; System call number (
29 00000018 CD80                       int     0x80         ; Interrupt 80 hex, ca
30
31                               ; Exit
32 0000001A 66BB00000000               mov     ebx,0        ; Exit code, 0=normal 
33 00000020 66B801000000               mov     eax,1        ; System call number (
34 00000026 CD80                       int     0x80         ; Interrupt 80 hex, ca
35                               ; End of the code
```

Reading this file, we can see that the first column from the left is simply the line number in the listing. The second column is the relative address, in hex, of where the code will be placed in memory. The third column is the actual compiled code.

For instance, in code `CD80` value `CD` is the x86 opcode[4] for `INT` instruction `INT imm8`; 80 is the decimal value 80 of interrupt vector number specified by immediate byte.

In code `66BA0D000000` Explain this

2.2.2 Code for GNU AS

Now take a look at the same program but written in differend dialect of asseBler: GNU Assembler (also GNU AS or simply GAS).

../programs/first_program/hello.s

```
/*  This program demonstrates basic text output to a screen.
 *  No "C" library functions are used.
 *  Calls are made to the operating system directly. (int 80 hex)
 *
 * assemble:     as hello.s -o hello.o
 * link:         ld hello.o -o hello
 * run:          ./hello
 * output is:    Hello World
 */
```

```
.data                          # Data section

text: .ascii "Hello␣World!\n"  # The string to print, 10=LF
len = . - text                 # "." means "here"
                               # len is a value, not an address

.text                          # code section

.global _start                 # Make label available to linker
                               # We must export the entry point to the ELF linker or
                               # loader. They conventionally recognize _start as their
                               # entry point. Use ld -e foo to override the default.

_start:                        # Standard ld entry point
       movl   $len, %edx       # arg3: length of string to print
       movl   $text, %ecx      # arg2: pointer to string
       movl   $1, %ebx         # arg1: where to write, so called file handler in this
                               # case stdout (screen)
       movl   $4, %eax         # System call number (sys_write)
       int    $0x80            # Interrupt 80 hex, call kernel

# Exit
       movl   $0, %ebx         # Exit code, 0=normal
       movl   $1, %eax         # System call number (sys_exit)
       int    $0x80            # Interrupt 80 hex, call kernel
# End of the code
```

The code looks a little bit strange but is equivalent to previously presented NASM version what we can verify assembling it

```
as hello.s -o hello.o
```

linking

```
ld hello.o -o hello
```

and finally runing

```
fulmanp@fulmanp-k2:~/assembler$ ./hello
Hello World!
```

If you want to know more... 2.3 (Making 32-bit code on 64-bit system with GNU AS). *As for NASM making 32-bit code on 64-bit system with GNU AS requires additional options usage*

```
fulmanp@fulmanp-k2:~/assembler$ as --32 hello.s -o hello.o
fulmanp@fulmanp-k2:~/assembler$ ld -m elf_i386 hello.o -o hello

fulmanp@fulmanp-k2:~/assembler$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048074
  Start of program headers:          52 (bytes into file)
  Start of section headers:          204 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 3
```

In the previous example the NASM syntax (Intel syntax) was used while now the GNU AS (AT&T syntax). See next section for more details; now only the most conspicuous differences would be commented.

- GAS supports two comment styles:

- Multi-line comments. As in C multi-line comments start and end with mirroring slash-asterisk pairs:

  ```
  /*
  comment
  */
  ```

- Single-Line comments. Single line comments have a few different formats varying on which architecture is being assembled for. For the platforms: i386, x86-64 (and many others) hash symbol (#)[‡] is used.

- In the source code instead of `mov` instruction `movl` is used[§]. It's specific to assemblers with AT&T syntax. The `l` is a size suffix that tells the compiler that we are working with dwords (double word = 4 bytes). To change the size, programmer changes the suffix (b, w, l, q for byte, word, dword, and qword). In NASM syntax instruction size is inferred by the operands..

- Register names are prefixed with %.

- Constant value/immediate are prefix with $.

- Opposite to the Intel syntax the source is on the left, and the destination is on the right.

2.2.3 AT&T vs. Intel assembly syntax

OK, GAS uses the AT&T assembly syntax (which is the UNIX standard) while NASM Intel syntax, but what does that mean to as?

Register name Register names are prefixed with %. To reference EAX:

```
AT&T:  %eax
Intel: eax
```

Source/Destination order In AT&T syntax the source is on the left, and the destination is on the right – opposite to the Intel syntax. To load EBX with the value in EAX

[‡]Semicolons is used on: AMD 29K family, ARC, H8/300 family, HPPA,PDP-11, picoJava, Motorola, and PowerPC; the at sign is used on the ARM platform; a vertical bar is used on 680x0; an exclamation mark on the Renesas SH platform etc.

[§]However this example would work also for `mov`.

```
AT&T:  movl %eax, %ebx
Intel: mov ebx, eax
```

Constant value/immediate value format Constant/immediate values are prefixed with `$`. To load EAX with the address of the variable `foo`

```
AT&T:  movl $foo, %eax
Intel: mov eax, foo
```

To load EBX with 1

```
AT&T:  movl $1, %ebx
Intel: mov ebx, 1
```

Operator size specification The instruction must be specified with one of `b`, `w`, or `l` to specify the width of the destination register as a byte, word or longword (double word).

```
AT&T:  movw %ax, %bx
Intel: mov bx, ax
```

Referencing memory Here is the canonical format for 32-bit addressing:

```
AT&T:  immed32(basepointer,indexpointer,indexscale)
Intel: [basepointer + indexpointer*indexscale + immed32]
```

The formula to calculate the address is

```
immed32 + basepointer + indexpointer * indexscale
```

We don't have to use all those fields, but we have to use at least one of `immed32` or `basepointer`. For example

- Addressing a particular variable

  ```
  AT&T:  foo
  Intel: [foo]
  ```

- Addressing what a register points to

Intel Code	AT&T Code
`mov eax,1`	`movl $1,%eax`
`mov ebx,0ffh`	`movl $0xff,\%ebx`
`int 80h`	`int $0x80`
`mov ebx, eax`	`movl %eax, %ebx`
`mov eax,[ecx]`	`movl (%ecx),%eax`
`mov eax,[ebx+3]`	`movl 3(%ebx),%eax`
`mov eax,[ebx+20h]`	`movl 0x20(%ebx),%eax`
`add eax,[ebx+ecx*2h]`	`addl (%ebx,%ecx,0x2),%eax`
`lea eax,[ebx+ecx]`	`leal (%ebx,%ecx),%eax`
`sub eax,[ebx+ecx*4h-20h]`	`subl -0x20(%ebx,%ecx,0x4),%eax`

Tabela 2.1: Intel vs. AT&T summary.

```
AT&T:  (%eax)
Intel: [eax]
```

- Addressing a variable offset by a value in a register

```
AT&T: variable(%eax)
Intel: [eax + variable]
```

- Addressing a value in an array of integers (scaling up by 4)

```
AT&T:  array(,%eax,4)
Intel: [eax*4 + array]
```

- Offsets with the immediate value

```
AT&T:  1(%eax)
Intel: [eax + 1]
```

- Addressing a particular char in an array of 8-character records (EAX holds the number of the record desired. EBX has the wanted char's offset within the record)

```
AT&T:  array(%ebx,%eax,8)
Intel: [ebx + eax*8 + array]
```

The table 2.1 summarizes all major differences between Intel and AT&T syntax.

2.3 64-bit basic stand alone program

2.3.1 Code for NASM

Listing 2.1: ../programs/first_program/hello_64.asm

```
;   This program demonstrates basic text output to a screen.
;   No "C" library functions are used.
;   Calls are made to the operating system directly.
;
; assemble:     nasm -f elf64 hello64.asm
; link:         ld hello64.o -o hello64
; run:          ./hello64
; output is:    Hello World

section .data               ; Data section

text:   db "Hello␣World!", 10   ; The string to print, 10=LF
len:    equ $-text          ; "$" means "here"
                            ; len is a value, not an address

section .text               ; Code section

global  _start              ; Make label available to linker
                            ; We must export the entry point to the ELF linker or
                            ; loader. They conventionally recognize _start as their
                            ; entry point. Use ld -e foo to override the default.

_start:                     ; Standard  ld  entry point
        mov     rdx, len    ; arg3: length of string to print
        mov     rsi, text   ; arg2: pointer to string
        mov     rdi, 1      ; arg1: where to write, so called file handler
                            ; in this case stdout (screen)
        mov     rax, 1      ; System call number (sys_write)
        syscall             ; Call a system function

; Exit
        mov     rdi, 0      ; Exit code, 0=normal
        mov     rax, 60     ; System call number (sys_exit)
        syscall             ; Call a system function
; End of the code
```

Verify correctnes of the code by assembling it

```
nasm -f elf64 hello_64.asm -o hello_64.o
```

linking

```
ld hello_64.o -o hello_64
```

and finally runing

```
fulmanp@fulmanp-k2:~/assembler$ ./hello_64
Hello World!
```

For the explanation of the code, see desciption of the code in section 2.2.

Notice that taking code from section 2.2 and replacing all 32-bit registers with 64-bit equvalents (e.g. replacing EAX with RAX), and even compiling it as 64-bit program the result we obtain is not a real 64-bit program. Just as in expert notes 2.1 any of the programs is not truly 64-bit.

2.3.2 Code for GNU AS

2.3.3 Excercise 1

Write 64-bit „hello word" program with AT&T syntax (GNU AS).

Solution

../programs/first_program/hello_64.s

```
NOT CORRECT !!!!!!!!!!!!!!!!!!!!!!!!

;  This program demonstrates basic text output to a screen.
;  No "C" library functions are used.
;  Calls are made to the operating system directly. (int 80 hex)
;
; assemble:      nasm -f elf64 hello64.asm
; link:          ld hello64.o -o hello64
; run:           ./hello64
; output is:     Hello World

section .data                  ; Data section

text:   db "Hello␣World!", 10   ; The string to print, 10=LF
```

```
len:      equ $-text              ; "$" means "here"
                                  ; len is a value, not an address

section  .text                    ; Code section

global   _start                   ; Make label available to linker
                                  ; We must export the entry point to the ELF linker or
                                  ; loader. They conventionally recognize _start as their
                                  ; entry point. Use ld -e foo to override the default.

_start:                           ; Standard  ld  entry point
         mov      rdx, len        ; arg3: length of string to print
         mov      rsi, text       ; arg2: pointer to string
         mov      rdi, 1          ; arg1: where to write, so called file handler
                                  ; in this case stdout (screen)
         mov      rax, 1          ; System call number (sys_write)
         syscall                  ; Call a system function

; Exit
         mov      rdi, 0          ; Exit code, 0=normal
         mov      rax, 60         ; System call number (sys_exit)
         syscall                  ; Call a system function
; End of the code
```

2.4 Multiple files

Imagine that we want distribute our code acros many files, like this

`File 1: routines.asm`

```
os_return:
    ;some code to return to os
do_something:
    ;some code to do something
```

`File 2: useRoutines.asm`

```
main:
   call do_something ; call function from separate file to do something
   ... maybe do something else here ...
   call os_return    ; call function from separate file to finish program
```

We can do this quite naural

../programs/first_program/routines.asm

```
section .data

strHello    db  "Hello", 10
strLen      equ $ - strHello

sys_exit    equ 1
sys_write   equ 4
stdout      equ 1

section .text

global do_something
global exit

do_something:
    mov     edx, strLen
    mov     ecx, strHello
    mov     eax, sys_write
    mov     ebx, stdout
    int     0x80
    ret

exit:
    mov     eax, sys_exit
    xor     ebx, ebx
    int     0x80
    ret
```

../programs/first_program/useRoutines.asm

```
section .text
```

```
extern  do_something
extern  exit
global  _start

_start:
    call    do_something
    call    exit
```

and compile, link and run almost usually

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf -o routines.o routines.asm
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf -o useRoutines.o useRoutines.asm
fulmanp@fulmanp-k2:~/assembler$ ld -m elf_i386 -o testSeparateRoutines routines.o useRout
fulmanp@fulmanp-k2:~/assembler$ ./testSeparateRoutines
Hello
```

If we want to use GCC to link our code, we have to change it a little bit in `useRoutines.asm`:

../programs/first_program/useRoutines_for_gcc.asm

```
section  .text

extern  do_something
extern  exit
global  main

main:
    call    do_something
    call    exit
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf -o routines.o routines.asm
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf -o useRoutines_for_gcc.o useRoutines_for_gcc.
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 -o testSeparateRoutine routines.o useRoutines_fo
fulmanp@fulmanp-k2:~/assembler$ ./testSeparateRoutine
Hello
```

ROZDZIAŁ 3

NASM syntax

Content of this chapter is a shortcut of offical documentation ([6]).

3.1 Layout of a NASM source line

Each NASM source line contains (unless it is a macro, a preprocessor directive or an assembler directive) some combination of the four fields

```
label:    instruction operands         ; comment
```

The presence or absence of any combination of a label, an instruction and a comment is allowed. Of course, the operand field is either required or forbidden by the presence and nature of the instruction field.

NASM uses backslash (\) as the line continuation character; if a line ends with backslash, the next line is considered to be a part of the backslash-ended line.

An identifier may also be prefixed with a $ to indicate that it is intended to be read as an identifier and not a reserved word.

Almost any floating-point instruction that references memory must use one of the prefixes DWORD, QWORD or TWORD to indicate what size of memory operand it refers to.

```
mov eax,[vec1+ecx*4]   ; Load [ecx] component of vector1
imul dword[vec2+ecx*4] ; Multiply eax by [ecx] component of vector2
                       ; Notice that we have to specify the size
                       ; of memory operand it refers to (dword).
```

3.2 Pseudo-instructions

Pseudo-instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them.

3.2.1 Declaring initialized data

NASM defines number of pseudo-instructions to declare initialized data in the output file.

```
db    0x55                 ; just the byte 0x55
db    0x55,0x56,0x57       ; three bytes in succession
db    'a',0x55             ; character constants are OK
db    'hello',13,10,'$'    ; so are string constants
dw    0x1234               ; 0x34 0x12
dw    'a'                  ; 0x61 0x00 (it's just a number)
dw    'ab'                 ; 0x61 0x62 (character constant)
dw    'abc'                ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678           ; 0x78 0x56 0x34 0x12
dd    1.234567e20          ; floating-point constant
dq    0x123456789abcdef0   ; eight byte constant
dq    1.234567e20          ; double-precision float
dt    1.234567e20          ; extended-precision float
dt    3.14159265358979323  ; pi
```

3.2.2 Declaring uninitialized data

NASM defines number of pseudo-instructions to declare uninitialized data. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve and are designed to be used in the BSS section of a module.

```
buffer:         resb    64              ; reserve 64 bytes
wordvar:        resw    1               ; reserve a word
realarray:      resq    10              ; array of ten reals
ymmval:         resy    1               ; one YMM register
zmmvals:        resz    32              ; 32 ZMM registers
```

3.2.3 Including external binary files

INCBIN pseudo-instruction includes a binary file verbatim into the output file. It can be called in one of these three ways:

```
    incbin  "file.dat"             ; include the whole file
    incbin  "file.dat",1024        ; skip the first 1024 bytes
    incbin  "file.dat",1024,512    ; skip the first 1024, and
                                   ; actually include at most 512
```

3.2.4 Defining constants

EQU defines a symbol to a given constant value: when EQU is used, the source line must contain a label. The action of EQU is to define the given label name to the value of its (only) operand. This definition is absolute, and cannot change later. For example:

```
message         db      'hello, world'
msglen          equ     $-message
```

3.2.5 Repeating instructions or data

The TIMES prefix causes the instruction to be assembled multiple times.

```
zerobuf:        times 64 db 0
```

The argument to TIMES is not just a numeric constant, but a numeric expression, so you can do things like

```
buffer: db      'hello, world'
times 64-$+buffer db ' '
```

which will store exactly enough spaces to make the total length of buffer up to 64.

3.3 Effective addresses

An effective address is any operand to an instruction which references memory. Effective addresses, in NASM, have a very simple syntax: they consist of an expression evaluating to the desired address, enclosed in square brackets. For example:

```
wordvar dw      123
        mov     ax,[wordvar]
        mov     ax,[wordvar+1]
        mov     ax,[es:wordvar+bx]
```

More complicated effective addresses, such as those involving more than one register, work in exactly the same way:

```
        mov     eax,[ebx*2+ecx+offset]
        mov     ax,[bp+di+8]
        mov     eax,[ebx+8,ecx*4]   ; ebx=base, ecx=index, 4=scale, 8=disp
```

3.4 Constants

3.4.1 Numeric constants

A numeric constant is simply a number. NASM allows you to specify numbers in a variety of number bases, in a variety of ways: you can suffix H or X, D or T, Q or O, and B or Y for hexadecimal, decimal, octal and binary respectively. NASM accept the prefix 0h for hexadecimal, 0d or 0t for decimal, 0o or 0q for octal, and 0b or 0y for binary. Numeric constants can have underscores (_) interspersed to break up long strings.

Some examples (all producing exactly the same code):

```
        mov     ax,200          ; decimal
        mov     ax,0200         ; still decimal
        mov     ax,0200d        ; explicitly decimal
        mov     ax,0d200        ; also decimal
        mov     ax,0c8h         ; hex
        mov     ax,$0c8         ; hex again: the 0 is required
        mov     ax,0xc8         ; hex yet again
        mov     ax,0hc8         ; still hex
        mov     ax,310q         ; octal
        mov     ax,310o         ; octal again
        mov     ax,0o310        ; octal yet again
        mov     ax,0q310        ; octal yet again
        mov     ax,11001000b    ; binary
        mov     ax,1100_1000b   ; same binary constant
        mov     ax,1100_1000y   ; same binary constant once more
        mov     ax,0b1100_1000  ; same binary constant yet again
        mov     ax,0y1100_1000  ; same binary constant yet again
```

3.4.2 String constants

String constants are character strings used in the context of some pseudo-instructions, namely the DB family and INCBIN (where it represents a filename.) They are also used in certain preprocessor directives. The following are equivalent:

```
      db    'hello'               ; string constant
      db    'h','e','l','l','o'   ; equivalent character constants
      dd    'ninechars'           ; doubleword string constant
      dd    'nine','char','s'     ; becomes three doublewords
      db    'ninechars',0,0,0     ; and really looks like this
```

3.4.3 Floating-point constants

Floating-point constants are acceptable only as arguments to DB, DW, DD, DQ, DT, and DO, or as arguments to the special operators __float8__, __float16__, __float32__, __float64__, __float80m__, __float80e__, __float128l__, and __float128h__.

Some examples:

```
      db    -0.2                    ; "Quarter precision"
      dw    -0.5                    ; IEEE 754r/SSE5 half precision
      dd    1.2                     ; an easy one
      dd    1.222_222_222           ; underscores are permitted
      dd    0x1p+2                  ; 1.0x2^2 = 4.0
      dq    0x1p+32                 ; 1.0x2^32 = 4 294 967 296.0
      dq    1.e10                   ; 10 000 000 000.0
      dq    1.e+10                  ; synonymous with 1.e10
      dq    1.e-10                  ; 0.000 000 000 1
      dt    3.141592653589793238462 ; pi
```

```
    mov     rax,__float64__(3.141592653589793238462)
```

3.4.4 Packed BCD constants

x87-style packed BCD constants can be used in the same contexts as 80-bit floating-point numbers. They are suffixed with `p` or prefixed with `0p`, and can include up to 18 decimal digits.

As with other numeric constants, underscores can be used to separate digits.

For example:

```
    dt 12_345_678_901_245_678p

    dt -12_345_678_901_245_678p

    dt +0p33

    dt 33p
```

ROZDZIAŁ 4

Basic CPU instructions

Typically instruction set is divided into four basic groups:

- arithmetic,
- logic,
- jump
- transfer.

We add to this list one more group: *utility instructions*.

4.1 Utility instructions

4.1.1 cbw

4.1.2 cwd

4.2 Arithmetic instructions

4.2.1 div

The DIV (unsigned divide) divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a generalpurpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode. Instruction formats:

```
Operand Size      Dividend Divisor Quotient Remainder Maximum Quotient
Word/byte         AX       r/m8    AL       AH        255
Doubleword/word   DX:AX    r/m16   AX       DX        65,535
Quadword/doubleword
                  EDX:EAX  r/m32   EAX      EDX       2^32 - 1
Doublequadword/
quadword          RDX:RAX  r/m64   RAX      RDX       2^64 - 1
```

As a good test let's try to write a code to print numbers.

../programs/basic_cpu_instructions/inst_64_div.asm

```
section .data                ; Data section

global  _start

_start:
        mov     dx, 0        ; dividend - higher half
        mov     ax, 16       ; dividend - lowere half
        mov     cx, 5        ; divisor
        div     cx           ; div dx:ax by cx

; Exit
                             ; Use exit code to get result
        mov     rdi, rax     ; Quotient
                             ; or
        ;mov    rdi, rdx     ; Remainder
        mov     rax, 60      ; System call number (sys_exit)
        syscall              ; Call a system function
; End of the code
```

```
[uncomment quotient, comment remainder]
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 inst_64_div.asm
fulmanp@fulmanp-k2:~/assembler$ ld inst_64_div.o -o inst_64_div
fulmanp@fulmanp-k2:~/assembler$ ./inst_64_div
```

```
fulmanp@fulmanp-k2:~/assembler$ $?
3: nie znaleziono polecenia
[comment quotient, uncomment remainder]
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 inst_64_div.asm
fulmanp@fulmanp-k2:~/assembler$ ld inst_64_div.o -o inst_64_div
fulmanp@fulmanp-k2:~/assembler$ ./inst_64_div
fulmanp@fulmanp-k2:~/assembler$ $?
1: nie znaleziono polecenia
```

If we know how DIV works we can try to implement function to print numbers.

4.2.2 Excercise 1

Write a program to print numbers.

Solution 1.1

../programs/basic_cpu_instructions/inst_64_print.asm

```
section .data                 ; Data section

transTab db "0123456789"      ; Translation Table

section .bss

result: resb 16               ; Reserve space for result
                              ; Max 16 digit

section .text

global _start

_start:
                              ; Put data to print into
                              ; edx:eax
        mov edx, 0
        mov eax, 12345
        jmp printNumber       ; Let's print
```

```
; Print number code: begin
; Init
printNumber:
        mov ebx, result     ; Set ebx to point begin of the buffer

; Prepare data
printLoop:
        mov ecx, 10
        div ecx             ; Div edx:eax by ecx
        mov ecx, [transTab + edx] ; Copy ASCII value of reminder to ECX
        mov [ebx], ecx      ; Copy ECX to 'result' buffer
        inc ebx             ; Move to the next byte in the buffer
        mov edx, 0          ; Restore edx
        cmp eax, 0          ; Compare EAX with immediate value: 0

        jne printLoop       ; Jump if operands of previous CMP instruction
                            ; are not equal - keep looping until EAX
                            ; is zero which means that all digits are
                            ; converted. When done go to
                            ; the print part

; Print result buffer
print:
        sub     ebx, result; Calculate length of string to print
        mov     rdx, rbx    ; arg3: length of string to print
        mov     rsi, result; arg2: pointer to string
        mov     rdi, 1      ; arg1: where to write, so called file handler
                            ; in this case stdout (screen)
        mov     rax, 1      ; System call number (sys_write)
        syscall             ; Call a system function

; Exit
        mov     rdi, 0      ; Exit code, 0=normal
        mov     rax, 60     ; System call number (sys_exit)
        syscall             ; Call a system function
; End of the code
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 inst_64_print.asm
```

```
fulmanp@fulmanp-k2:~/assembler$ ld inst_64_print.o -o inst_64_print
fulmanp@fulmanp-k2:~/assembler$ ./inst_64_print
54321
```

Our solution works but it's far from perfection: number 12345 was printed as 54321. Let's try to fix it.

Solution 1.2

../programs/basic_cpu_instructions/inst_64_print_02.asm

```
section .data               ; Data section

transTab db "0123456789"    ; Translation Table

section .bss

result: resb 16             ; Reserve space for result
                            ; Max 16 digit

section .text

global _start

_start:
                            ; Put data to print into
                            ; edx:eax
        mov edx, 0
        mov eax, 32123
        jmp printNumber     ; Let's print

; Print number code: begin
; Init
printNumber:
        mov ebx, result + 15 ; Set ebx to point end of the buffer  HERE

; Prepare data
printLoop:
        mov ecx, 10
        div ecx               ; Div edx:eax by ecx
        mov ecx, [transTab + edx] ; Copy ASCII value of reminder to ECX
                              ; This copy reads 4 bytes
                              ; Because now we print from right to left
                              ; so when we print 4-byte blok it will erase
                              ; previous digits.
                              ; This is why previous instruction:
        ; mov [ebx], ecx      ; Copy ECX to 'result' buffer HERE
                              ; have to be raplace by
        mov [ebx], cl         ; Copy 1 byte from ECX (which is CL)
                              ; to 'result' buffer HERE
        dec ebx               ; Move to the previous byte in the buffer  HERE
        mov edx, 0            ; Restore edx
        cmp eax, 0            ; Compare EAX with immediate value: 0

        jne printLoop         ; Jump if operands of previous CMP instruction
                              ; are not equal - keep looping until EAX
                              ; is zero which means that all digits are
                              ; converted. When done go to
                              ; the print part

; Print result buffer
print:
        xor     rax, rax      ; HERE
        mov     eax, result + 15 + 1 ; HERE new
        sub     eax, ebx      ; Calculate length of string to print  HERE
        mov     rdx, rax      ; arg3: length of string to print
        xor     rsi, rsi      ; HERE new
        mov     esi, ebx      ; arg2: pointer to string  HERE
        mov     rdi, 1        ; arg1: where to write, so called file handler
                              ; in this case stdout (screen)
        mov     rax, 1        ; System call number (sys_write)
        syscall               ; Call a system function

; Exit
        mov     rdi, 0        ; Exit code, 0=normal
        mov     rax, 60       ; System call number (sys_exit)
        syscall               ; Call a system function
; End of the code
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 inst_64_print_02.asm
fulmanp@fulmanp-k2:~/assembler$ ld inst_64_print_02.o -o inst_64_print_02
fulmanp@fulmanp-k2:~/assembler$ ./inst_64_print_02
32123
```

All changed parts of the code are marked by `HERE` string. General idea of the changes is clear: print digits in oposite dirrection, from end of the buffer to the begin. To do this, we have to set index to the last element of the buffer and decrease it every new character. Please not very subtle change in the code - rename ECX to CL. Explanation for this is as follow. Instruction

```
mov ecx, [transTab + edx]
```

copy ASCII value of reminder to ECX register. This copy reads 4 bytes. Because now we print from right to left so when we print 4-byte blok it will erase previous digits. This is why previous instruction

```
mov [ebx], ecx
```

have to be raplace by

```
mov [ebx], cl
```

when we copy 1 byte from ECX (which is CL) to 'result' buffer. Without the changes described above this result is not correct

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 inst_64_print_02.asm
fulmanp@fulmanp-k2:~/assembler$ ld inst_64_print_02.o -o inst_64_print_02
fulmanp@fulmanp-k2:~/assembler$ ./inst_64_print_02
34565
```

This could be explained as follow

```
We want to print 32123


When print
3 -> transTab -> 3456


result:
          111111
```

```
0123456789012345
               3456

When print
2 -> transTab -> 2345

result:
          111111
0123456789012345
               3456
              2345

When print
1 -> transTab -> 1234

result:
          111111
0123456789012345
               3456
              2345
             1234

When print
2 -> transTab -> 2345

result:
          111111
0123456789012345
               3456
              2345
             1234
            2345
```

```
When print
3 -> transTab -> 3456


result:
          111111
0123456789012345
               3456
              2345
             1234
            2345
           3456
           ||||| - we take last digit from every position
           34565
```

Solution 1.3

This solution is dedicated to MacOS

../programs/nie_moje/fj_liczba.asm

```
; Author:       Justyna Firkowska
; System:       Mac OS X, NASM 32-bit
; Assemble:     nasm -f macho liczba.asm
; Link:         ld liczba.o -o liczba
; Run:          ./liczba

section .data
   Digits   db "0123456789"

section .bss
   Result: resb 8         ; rezerwuje 8 bajtow na wynik

section .text

global start

start:
    mov    eax, 12345   ; zapisuje liczbe do wypisania w eax
    mov    ebx, 0xA     ; ustawia dzielnik na 10
    mov    ebp, Result + 6   ; zapisuje Result + 6 w ebp

    jnz    printLoop    ; skacze do petli

printLoop:
    div    ebx          ; dzieli eax przez 10 (wynik —> edx)
    mov    cl, [Digits + edx]    ; zapisuje wartosc ASCII w cl
    mov    [ebp], cl    ; zapisuje cl w buforze Result
    dec    ebp          ; przechodzi do kolejnego bajtu w buforze
    xor edx, edx      ; zeruje reszte z dzielenia

    inc    eax
    dec    eax
    jnz    printLoop
    jz   print        ; przechodzi do wypisywania

print:
    mov [Result+7], byte 0xA   ; dodaje znak nowej linii do wyniku

    push   0x8        ; wstawia max dlugosc (8 bajtow) na stos
    push   Result     ; wstawia wynik na stos
    push   0x1        ; FD stdout (miejsce wypisania – ekran)
    mov    eax, 0x4   ; sys_write call
    push   eax        ; Push call (BSD)
    int    0x80       ; Call
    add    esp, 0x10  ; czysci stos

; Exit
    mov    eax, 0x1   ; sys_exit call
    push   0x0        ; Exit_code 0
    int    0x80       ; Call
```

???

Solution 1.4

Different approach with a stack

../programs/nie_moje/kk_print.asm

```
; Works for numbers up to 2^64 - 1
; Author:     Konrad Kosmatka
; Assemble:   nasm -f elf64 print.asm -o print.o
; Link:       ld print.o -o print
; Run:        ./print

SECTION .data                      ; data section
char:
number: dq 18446744073709551615   ; 2^64 - 1

SECTION .text
global _start


_start:
    push rbp            ;zapis obecnego base pointer na stos
    mov rbp, rsp        ;teraz stos sie zaczyna od obecnego miejsca
loop:
    mov rdx, 0          ;zerowanie rdx
    mov rax, [number]   ;wczytanie dzielnej
    mov rbx, 10         ;wczytanie dzielnika
    div rbx             ;dzielenie
    push rdx            ;wrzucenie reszty z dzielenia na stos
    cmp rax, 0          ;czy iloraz jest rowny 0?
    je print            ;tak - wyswietl wynik (ze stosu)
    mov [number], rax   ;nie - zapis iloraz w miejsce dzielnej
    jmp loop            ;w petli

print:
    cmp rsp, rbp       ;czy na stosie jeszcze cos jest?
    je return          ;nie, koniec
    pop rbx            ;tak, pobieramy

    mov rdx, 1         ;arg3 - wypisujemy jeden znak
    mov rax, '0'       ;wczytaj znak zera
    mov [char], rax    ;zapisz znak zera do zmiennej char
    add [char], rbx    ;dodaj reszte z dzielenia
    mov rsi, char      ;arg2 - wskaznik na string
    mov rdi, 1         ;arg1 - stdout
    mov rax, 1          ;sys_write
    syscall
    jmp print           ;w petli

return:
    pop rax             ; przywracamy
    mov rbp, rax        ; stos (w sumie niepotrzebne)

    mov rdi, 0          ; return exit code (0=normal)
    mov rax, 60         ; system call number (sys_exit)
    syscall
```

???

4.2.3 add

4.2.4 sub

4.2.5 mul

4.2.6 idiv

4.2.7 imul

4.2.8 cmp

4.2.9 inc

4.2.10 dec

4.3 Logic instructions

4.3.1 and

4.3.2 or

4.3.3 not

4.3.4 xor

4.3.5 shl

4.3.6 shr

4.3.7 test

On x86, test does a binary AND between the operands, just does not save the result anywhere. cmp subtracts the second operand from the first without actually modifying the first operand.

In other words, if you wanted to check if bit 6 (01000000b = 26 = 64) is set in register ch, then you'd use test ch, 64. If you wanted to see if ch is less than/equal to/greater than 64, then you'd do cmp ch, 64.

Remember, the difference is that cmp does a subtraction, and test a binary AND operation, with the result discarded and only the flags affected. They are two very different operations.

¿ Hi, I'm a rookie in assembly language, this ¿ question just came up my mind, so I post it ¿ here in the hope that someone would explain ¿ more details about the topic. ¿ ¿ when test to see if a variable contains ¿ a zero value, people ususally use ¿ ¿ TEST reg,reg ¿ JZ Lable1 ¿ ¿ alternatively, ¿ ¿ CMP reg,0 ¿ JE Label1 ¿ ¿ is also correct, ¿ So, what's the difference ? ¿ and any other important things which pertinent ¿ is also welcome here. thanx in advance :))))

The test and cmp instructions are aliases for and and sub respectively except that test and cmp only update the flags. Therefore:

test eax, eax ; sets flags like and eax, eax jz @eax$_is_zero$

cmp eax, 0 ; sets flags like sub eax, 0 je @eax$_is_zero$

Note that je and jz are aliases for each other. It is true that x - x == 0, so if you cmp eax, eax (sub eax, eax), then the result will be 0 and the machine will set ZF (zero flag) to 1.

Here's an example that shows how these 2 instructions differ:

test eax, 1 jnz @eax$_is_oddjc@never_branchjo@never_branch$

cmp eax, 1 jnz @eax$_is_not_onejc@eax_is_zerojo@eax_is_int_min$

Here test is only checking the least significant bit, so if eax == 0 then the jnz will not be taken. If eax == 1 then the jnz will be taken. Both CF (carry flag) and OF (signed overflow flag) are cleared to 0 by test, so a jc/jo after a test will never branch.

The cmp works a bit differently as you can see. eax - 1 == 0 only if eax == 1, so jnz is taken if eax != 1. The only case where sub eax, 1 will underflow is when eax == 0, so if CF is set then we know eax was 0. Also, OF will be set if eax == -2$^31 since eax will wrap around to 2^31 - 1.$

-Matt

4.4 Jump instructions

UZUPELNIC!!! Roznice pomiedzy JNA a JBE!!!, http://www.cs.ubbcluj.ro/~dadi/ac/doc/nga904.html

4.4.1 jmp

4.4.2 call

4.4.3 JZ

Jump short if zero (ZF = 1).

4.4.4 JE

Jump short if equal (ZF=1).

4.4.5 JNZ

Jump short if not zero (ZF=0).

4.4.6 JNE

Jump short if not equal (ZF=0).

4.4.7 JA

Jump short if above (CF=0 and ZF=0).

4.4.8 JNA

Jump short if not above (CF=1 or ZF=1).

4.4.9 JB

Jump short if below (CF=1).

4.4.10 JNB

Jump short if not below (CF=0).

4.4.11 CMP, TEST and JE

Consider code like this

```
TEST eax, eax
JE   error
```

which cold be very confusing to what this does. Aren't the values in `EAX` and `EAX` the same? What is it testing? If TEST is doing the AND operation and they (both EAX) are the same values, wouldn't it just return EAX?

Once again. . .

CMP subtracts the operands and sets the *Zero Flag* if the difference is zero (which means that operands are equal).

TEST sets the *Zero Flag* if the the result of the AND operation is zero. If two operands are equal, their bitwise AND is zero if and only if both are zero. It also sets the *Sign Flag* if the top bit is set in the result, and the *Parity Flag* if the number of set bits is even.

JE (Jump if Equals)[*] tests the *Zero Flag* and jumps if the flag is set.

So simply speaking,

```
TEST eax, eax
JE   error
```

jumps to `error` if the EAX is zero.

4.4.12 LOOP

Performs a loop operation using the ECX or CX register as a counter. Each time the LOOP instruction is executed, the count register is decremented, 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 near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop. If the address-size attribute is 32 bits, the ECX register is used as the count register. Otherwise, the CX register is used.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the EIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of –128 to +127 are allowed with this instruction.

4.4.13 Jump examples

../programs/basic_cpu_instructions/jmp_loop_test1_32.asm

```
section .data
a: dq 5
b: dq 7
r: db "a␣==␣b", 10
k: db "koniec", 10
```

[*]Which is an alias of JZ (Jump if Zero).

```
section .text

global _start

_start:
   mov eax, [a]
   cmp eax, [b]
   jne dalej

   mov eax, 4
   mov ebx, 1
   mov ecx, r
   mov edx, 7
   int 0x80

dalej:
   mov eax, 4
   mov ebx, 1
   mov ecx, k
   mov edx, 7
   int 0x80

   mov eax, 1
   mov ebx, 0
   int 0x80
```

../programs/basic_cpu_instructions/jmp_loop_test2_32.asm

```
section .data
a: dq 7
b: dq 7
r: db "a␣==␣b", 10
n: db "a␣!=␣b", 10

section .text

global _start

_start:
    mov eax, [a]
    cmp eax, [b]
    jne else_

        ; if(a == b)
        push r

    jmp endif_
else_:
        ; else
        push n
endif_:

    mov eax, 4
    mov ebx, 1
    mov ecx, [esp]
    mov edx, 7
    int 0x80

    mov eax, 1
    mov ebx, 0
    int 0x80
```

../programs/basic_cpu_instructions/jmp_loop_test3_32.asm

```
section .data
a: dq 7
b: dq 5
w: db "a␣>␣b", 10
m: db "a␣<␣b", 10
r: db "a␣=␣b", 10

section .text

global _start

_start:
    mov eax, [a]
    mov ebx, [b]
    cmp eax, ebx
    jng elseif_

        ; if(a > b)
        push w

    jmp endif_
elseif_:
    ;cmp eax, ebx
    jnl else_

        ; else if(a < b)
        push m

    jmp endif_
else_:
        ; else
        push r
endif_:

    mov eax, 4
    mov ebx, 1
    mov ecx, [esp]
    mov edx, 6
    int 0x80

    mov eax, 1
    mov ebx, 0
    int 0x80
```

../programs/basic_cpu_instructions/jmp_loop_test4_32.asm

```
section .data
string:  db "tekst␣ktorego␣nie␣bedzie␣widac", 10
len:  equu $ - string

section .text

global _start
```

```
_start:
   mov ecx, string
petla:
   mov byte [ecx], '*';
   inc ecx

   cmp byte [ecx], 10
   jne petla

   mov eax, 4
   mov ebx, 1
   mov ecx, string
   mov edx, len
   int 0x80

   mov eax, 1
   mov ebx, 0
   int 0x80
```

../programs/basic_cpu_instructions/jmp_loop_test5_32.asm

```
section .data
string:  db "tego_nie_bedzie_widac␣widac␣tylko␣to", 10
len:  equ $ - string

section .text

global _start

_start:
   mov ecx, string
while_:
   cmp byte [ecx], '␣'
   je endwhile_
   cmp byte [ecx], 10
   je endwhile_

   mov byte [ecx], '*';
   inc ecx
```

```
    jmp while_
endwhile_:

    mov eax, 4
    mov ebx, 1
    mov ecx, string
    mov edx, len
    int 0x80

    mov eax, 1
    mov ebx, 0
    int 0x80
```

../programs/basic_cpu_instructions/jmp_loop_test6_32.asm

```
section .data
string:  db "jakis␣tekst", 10
len:  equ $ - string
n: dd 8

section .text

global _start

_start:
    mov ecx, 0
for_:
    cmp ecx, [n]
    jnb endfor_

    mov byte [string + ecx], '*';

    inc ecx
    jmp for_
endfor_:

    mov eax, 4
    mov ebx, 1
    mov ecx, string
    mov edx, len
    int 0x80

    mov eax, 1
    mov ebx, 0
    int 0x80
```

../programs/basic_cpu_instructions/jmp_loop_test7_32.asm

```
section .data
    string db 'a', 10

section .text

global _start

_start:
    mov ecx, 10
petla:
    inc byte [string]
    loop petla

    mov eax, 4
    mov ebx, 1
    mov ecx, string
    mov edx, 2
    int 0x80

    mov eax, 1
    mov ebx, 0
    int 0x80
```

../programs/basic_cpu_instructions/jmp_loop_test8_32.asm

```
section .data
    string db "abcdefg", 10
    len equ $ - string

section .text

global _start
```

```
_start:
   mov eax, string
   mov ecx, len - 1
petla:
   add [eax], dword 4
   inc eax
   loop petla

   mov eax, 4
   mov ebx, 1
   mov ecx, string
   mov edx, len
   int 0x80

   mov eax, 1
   mov ebx, 0
   int 0x80
```

../programs/basic_cpu_instructions/jmp_loop_test9_32.asm

```
;   LOOP
;   LOOPE - JE
;   LOOPNE    - JNE
;   LOOPZ - JZ
;   LOOPNZ    - JNZ

section .data
str1: db "to␣jest_jakis␣tekst", 10
len1: equ $ - str1
str2: db "xyzinny␣te#st...", 10
len2: equ $ - str2

section .text

global _start

_start:
   mov ecx, len2
petla:
   mov al, [str1 + ecx]
```

```
cmp al, [str2 + ecx]
loopne petla

mov byte [str1 + ecx], '*';
mov byte [str2 + ecx], '*';

mov eax, 4
mov ebx, 1
mov ecx, str1
mov edx, len1
int 0x80

mov eax, 4
mov ebx, 1
mov ecx, str2
mov edx, len2
int 0x80

mov eax, 1
mov ebx, 0
int 0x80
```

4.5 Transfer instructions

4.5.1 mov

4.5.2 call

4.5.3 push

4.5.4 pop

4.5.5 pusha

4.5.6 popa

4.5.7 xchg

ROZDZIAŁ 5

Debugging with GDB

code 2.1 from section 2.3

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 hello_64.asm -o hello_64.o
fulmanp@fulmanp-k2:~/assembler$ ld hello_64.o -o hello_64
fulmanp@fulmanp-k2:~/assembler$ ./hello_64
Hello World!
fulmanp@fulmanp-k2:~/assembler$ cat hello_64.lst
     1                                  section .data
     2
     3 00000000 48656C6C6F20576F72-     text:   db "Hello World!", 10
     4 00000009 6C64210A
     5                                  len:    equ $-text
     6
     7                                  section .text
     8
     9                                  global  _start
    10
    11                                  _start:
    12 00000000 BA0D000000                      mov     rdx, len
    13 00000005 48BE-                           mov     rsi, text
    14 00000007 [0000000000000000]
```

```
15 0000000F BF01000000                      mov     rdi, 1
16
17 00000014 B801000000                      mov     rax, 1
18 00000019 0F05                            syscall
19
20 0000001B BF00000000                      mov     rdi, 0
21 00000020 B83C000000                      mov     rax, 60
22 00000025 0F05                            syscall
23
```

ROZDZIAŁ 6

First program linked with a C library

6.1 32-bit basic program linked with a C library

6.1.1 Code for NASM

../programs/first_program/hello_c.asm

```
;  This program demonstrates basic text output to a screen.
;  It needs to be linked with a C library - pintf "C" library functions is used.
;
; assemble:      nasm -f elf hello.asm
; link:          gcc hello.o -o hello
; run:           ./hello
; output is:     Hello World

section .data                ; Data section


text    db "Hello␣World!", 10, 0   ; The string to print, 10=cr, 0=null
                             ; null terminated string have to be used
                             ; in order to use printf function


section .text                ; Code section


extern  printf               ; The C function, to be called


global  main                 ; Make label available to linker
```

```
main:                           ; Standard gcc entry point
        push    text            ; Address of control string for printf function
        call    printf          ; Call C function
        add     esp, 4          ; pop stack 1 push times 4 bytes

; Exit
        mov eax,0       ; Normal, no error, return value
        ret               ; Return
; End of the code
```

Verify correctnes of the code by assembling it

```
nasm -f elf hello_c.asm -o hello_c.o
```

linking

```
gcc hello_c.o -o hello_c
```

and finally runing

```
fulmanp@fulmanp-k2:~/assembler$ ./hello_c
Hello World!
```

If you want to know more... 6.1 (Making 32-bit program linked with a C library on 64-bit system)**.** *Making 32-bit program linked with a C library on 64-bit system requires the following commands (on my Linux, the* `gcc-multilib` *package had to be installed.)*

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf hello_c.asm -o hello_c.o
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 hello_c.o -o hello_c
fulmanp@fulmanp-k2:~/assembler$ ./hello_c
Hello World!
```

To understand this code, we have to understand calling conventions (more about this in the chapter ??).

6.1.2 GCC 32-bit calling conventions in brief

Writing assembly language functions that will link with C, and using gcc, we must obey the gcc calling conventions.

- Parameters are pushed on the stack, right to left, and **are removed by the caller** after the call.
- After the parameters are pushed, the `call` instruction is made, so when the called function gets control, the return address is at `[esp]`, the first parameter is at `[esp+4]`, etc.
- Using any of the following registers: EBX, ESI, EDI, EBP, DS, ES and SS we must save and restore their values. In other words, **these values must not change across function calls**.
- A function that returns an integer value should return it in EAX, a 64-bit integer in EDX:EAX, and a floating point value should be returned on the fpu stack top.

6.1.3 Excercise

Write in assembler an equivalent of the folowing C program running on 32-bit system

../programs/first_program/simple_printf_32.c

```
#include <stdio.h>

int main()
{
  char    char1='a';          /* Sample character */
  char    str1[]="abcdefgh";  /* Sample string */
  int     int1=123;           /* Sample integer */
  int     hex1=0x1234ABCD;    /* Sample hexadecimal */
  float   flt1=1.234e-3;      /* Sample float */
  double  flt2=-123.4e300;    /* Sample double */

  printf("printf␣test:\ncharacter=%c\nstring=%s\ninteger=%d\ninteger␣(hex)=%X\nfloat=%f\ndou
         char1, str1, int1, hex1, flt1, flt2);
  return 0;
}
```

Solution

../programs/first_program/simple_printf_32.asm

```
section .data
```

```
; Format string for printf
form_s: db "printf␣test:",10,"character=%c",10,"string=%s",10,"integer=%d",10,"integer␣(hex)=
; Other data
char1: db  'a'              ; Sample  character
str1:  db  "abcdefgh",0     ; Sample C string (needs 0)
int1:  dd  123              ; Sample integer
hex1:  dd  0x1234ABCD       ; Sample hexadecimal
flt1:  dd  1.234e-3         ; 32-bit floating point (float)
flt2:  dq  -123.4e3         ; 64-bit floating point (double)

section .bss                ; The data segment containing statically-allocated
                            ; variables - free space allocated for the future use

flttmp: resq 1              ; Statically-allocated variables without an explicit
                            ; initializer; 64-bit temporary for printing flt1

section .text               ; Code section

extern  printf              ; The C function, to be called

global  main                ; Make label available to linker

main:                       ; Standard gcc entry point

                            ; Note that printf will NOT ACCEPT single precision floats.
                            ; We have to convert them to double precision floats:
  fld   dword [flt1]        ; convert 32-bit to 64-bit via 80-bits FPU stack
  fstp  qword [flttmp]      ; Floating load makes 80-bit, store as 64-bit
                            ; Push last argument first
  push  dword [flt2+4]      ; 64 bit floating point (bottom)
  push  dword [flt2]        ; 64 bit floating point (top)
  push  dword [flttmp+4]    ; 64 bit floating point (bottom)
  push  dword [flttmp]      ; 64 bit floating point (top)
  push  dword [hex1]        ; Hex constant
  push  dword [int1]        ; Constant pass by value
  push  str1                ; "string" pass by reference
  push  dword [char1]       ; 'a'
  push  form_s              ; Address of format string
  call  printf              ; Call C function
```

```
add    esp, 36              ; Pop stack 10*4 bytes

mov    eax, 0               ; Exit code, 0=normal
ret                         ; Main returns to operating system
```

The code assembly, link and run as previously

- as a 32-bit program on 32-bit system to test and complete; now I have only 64bit system
- as a 32-bit program on 64-bit system

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf32 simple_printf_32.asm -o simple_printf_3
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 simple_printf_32.o -o simple_printf_32
fulmanp@fulmanp-k2:~/assembler$ ./simple_printf_32
printf test:
character=a
string=abcdefgh
integer=123
integer (hex)=1234ABCD
float=0.001234
double=-1.234000e+302
```

Notice that in this program a new section, the **BSS section**, was used. The name *.bss* or *bss* usually is used by compilers and linkers for a part of the data segment containing uninitialized variables statically-allocated variables represented solely by zero-valued bits initially (i.e., when execution begins). It is often referred to as the *bss section* or *bss segment*.

The BSS segment gets its name from abbreviation "Block Started by Symbol" – a pseudo-op from the old IBM 704 assembler, carried over into UNIX, and there ever since. Some people like to remember it as "Better Save Space". Since the BSS segment only holds variables that don't have any value yet, it doesn't actually need to store the image of these variables. The size that BSS will require at runtime is recorded in the object file, but BSS (unlike the data segment) doesn't take up any actual space in the object file[3].

6.2 64-bit basic program linked with a C library

6.2.1 Code for NASM

../programs/first_program/hello_c_64.asm

```
section .data             ; Data section

text:     db "Hello␣World!", 10, 0   ; The string to print, 10=cr, 0=null
                          ; null terminated string have to be used
                          ; in order to use printf function

section .text             ; Code section

extern  printf            ; The C function, to be called

global  main              ; Make label available to linker

main:                     ; Standard gcc entry point
        mov   rdi, text   ; 64-bit ABI passing order: RDI, RSI, ...
        mov   rax, 0      ; printf is varargs, so RAX counts # of non-integer
                          ; arguments being passed
        call  printf      ; The C function, to be called

; Exit
        mov   rax,0       ; Normal, no error, return value
        ret               ; Return
; End of the code
```

Verify correctnes of the code by assembling it

```
nasm -f elf64 hello_c_64.asm -o hello_c_64.o
```

linking

```
gcc hello_c_64.o -o hello_c_64
```

and finally runing

```
fulmanp@fulmanp-k2:~/assembler$ ./hello_c_64
Hello World!
```

To understand this code, we have to understand calling conventions (more about this in the chapter ??).

6.2.2 GCC 64-bit calling conventions in brief

Writing assembly language functions that will link with C, and using gcc, we must obey the gcc calling conventions. Notice that the 64-bit calling conventions differs from 32-bit calling conventions and are different for different operating systems. The most important points are (for 64-bit Linux)

- Parameters are passing from left to right and as many parameters as will fit in registers. The order in which registers are allocated, are
  - For integers and pointers: RDI, RSI, RDX, RCX, R8, R9.
  - For floating-point (float, double): XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7.
- If needed, additional parameters are pushed on the stack, right to left, and are removed by the caller after the call.
- After the parameters are pushed, the call instruction is made, so when the called function gets control, the return address is at `[ESP]`, the first memory parameter is at `[ESP + 8]`, etc.
- Variable-argument subroutines require a value in RAX for the number of vector registers used. In other words when a function taking variable-arguments is called, RAX must be set to the total number of floating point parameters passed to the function in vector registers. See below for more explanation.
- The only registers that the called function is required to preserve (the calle-save registers) are: RBP, RBX, R12, R13, R14, R15. All others are free to be changed by the called function.
- The callee is also supposed to save the control bits of the XMCSR and the x87 control word.
- Integers are returned in RAX or RDX:RAX, and floating point values are returned in XMM0 or XMM1:XMM0.

RAX value for variable-argument subrutines

In the x86_64 ABI, if a function has variable arguments then AL (which is part of EAX) is expected to hold the number of vector registers used to hold arguments to that function. For example

```
printf("%d", 1);
```

has an integer argument so there's no need for a vector register, hence AL is set to 0. On the other hand, if we change this example to

```
printf("%f", 1.0f);
```

then the floating-point literal is stored in a vector register and, correspondingly, AL is set to 1

```
movsd   LC1(%rip), %xmm0
leaq    LC0(%rip), %rdi
movl    $1, %eax
call    _printf
```

As we can expect the code

```
printf("%f %f", 1.0f, 2.0f);
```

will cause the compiler to set AL to 2 since there are two floating-point arguments

```
movsd   LC0(%rip), %xmm0
movapd  %xmm0, %xmm1
movsd   LC2(%rip), %xmm0
leaq    LC1(%rip), %rdi
movl    $2, %eax
call    _printf
```

6.2.3 Excercise 2

Write a 64-bit program from excercise 6.1.3.

Solution

../programs/first_program/simple_printf_64.asm

```
section .data           ; Data section


; Format string for printf
form_s: db "printf␣test:",10,"character=%c",10,"string=%s",10,"integer=%d",10,"integer␣(hex)
; Other data
```

```
char1: db  'a'            ; Sample  character
str1:  db  "abcdefgh",0   ; Sample C string (needs 0)
int1:  dd  123            ; Sample integer
hex1:  dd  0x1234ABCD     ; Sample hexadecimal
flt1:  dd  1.234e-3       ; 32-bit floating point (float)
flt2:  dq  -123.4e3       ; 64-bit floating point (double)

section .bss              ; The data segment containing statically-allocated
                          ; variables - free space allocated for the future use

flttmp: resq 1            ; Statically-allocated variables without an explicit
                          ; initializer; 64-bit temporary for printing flt1

section .text             ; Code section

extern  printf            ; The C function, to be called

global  main              ; Make label available to linker

main:                     ; Standard gcc entry point

  fld   dword [flt1]      ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp  qword [flttmp]    ; Floating load makes 80-bit, store as 64-bit

  mov   rdi, form_s       ; 64-bit ABI passing order: rdi, rsi, ...
  mov   rsi, [char1]
  mov   rdx, str1
  mov   rcx, [int1]
  mov   r8,  [hex1]
  movsd  xmm0, [flttmp]   ; Simple movss  xmm0, [flt1] doesn't work, because
                          ; printf needs 64-bit floating-points numbers
                          ; (floats and doubles)
  movsd  xmm1,[flt2]
  mov   rax, 2            ; printf is varargs, so EAX counts # of non-integer
                          ; arguments being passed
  sub  rsp, 8             ; Tricky part. Add some stack space to frame. Why?
                          ; The stack must be 16-byte aligned.
  call  printf            ; The C function, to be called
  add  rsp, 8             ; Remove added stack space
```

```
; Exit
  mov   rax,0              ; Normal, no error, return value
  ret                      ; Return
; End of the code
```

The code assembly, link and run as previously

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 simple_printf_64.asm -o simple_printf_64.o
fulmanp@fulmanp-k2:~/assembler$ gcc simple_printf_64.o -o simple_printf_64
fulmanp@fulmanp-k2:~/assembler$ ./simple_printf_64
printf test:
character=a
string=abcdefgh
integer=123
integer (hex)=1234ABCD
float=0.001234
double=-1.234000e+302
```

Notice the tricky part of the code. Some stack space is added to frame. Why? The stack must be 16-byte aligned and is 16-byte aligned at the beginning of main(). The call instruction pushed the 8-byte return address onto the stack, which misaligns it and causes you to move RSP by some odd multiple of 8 bytes to realign it. A good question is why a misaligned stack causes a seg fault only when using a vector register (a register! not the stack!) – `hello_c_64.asm` works preety fine witthout this.

If you want to know more... 6.2 (Prying assembler code generated by GCC)**.** *Sometimes, when we drop into troubles, it's very useful to inspect code (working code) generated by some tools, like GCC. Having code as follow*

../programs/first_program/simple_printf_64.c

```
#include <stdio.h>

int main()
{
  double  flt1=1.234e-3;      /* Sample float */
  printf("printf␣float=%e\n", /* Format string for printf */
          flt1 );
  return 0;
}
```

we can type

```
fulmanp@fulmanp-k2:~/assembler$ gcc -S simple_printf_64.c -o simple_printf_64_dis.s
```

to get code we can follow (notice that presented code is compatible with AT&T syntax).

../programs/first_program/simple_printf_64_dis.s

```
    .file  "simple_printf_64.c"
    .section  .rodata
.LC1:
    .string   "printf␣float=%e\n"
    .text
    .globl    main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
    .cfi_def_cfa_register 6
    subq  $16, %rsp
    movabsq  $4563333643445681349, %rax
    movq  %rax, -8(%rbp)
    movl  $.LC1, %eax
    movsd -8(%rbp), %xmm0
    movq  %rax, %rdi
    movl  $1, %eax
    call   printf
    movl  $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

```
.LFE0:
    .size main, .-main
    .ident "GCC:␣(Ubuntu/Linaro␣4.6.3-1ubuntu5)␣4.6.3"
    .section .note.GNU-stack,"",@progbits
```

To get code compatible with Intel syntax use

```
fulmanp@fulmanp-k2:~/assembler$ gcc -S -masm=intel simple_printf_64.c -o simple_printf_64
```

../programs/first_program/simple_printf_64_dis.asm

```
    .file "simple_printf_64.c"
    .intel_syntax noprefix
    .section .rodata
.LC1:
    .string "printf␣float=%e\n"
    .text
    .globl main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    push rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    mov rbp, rsp
    .cfi_def_cfa_register 6
    sub rsp, 16
    movabs rax, 4563333643445681349
    mov QWORD PTR [rbp-8], rax
    mov eax, OFFSET FLAT:.LC1
    movsd xmm0, QWORD PTR [rbp-8]
    mov rdi, rax
    mov eax, 1
    call printf
    mov eax, 0
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

```
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
    .section .note.GNU-stack,"",@progbits
```

or having compiled file dissasembly it

```
fulmanp@fulmanp-k2:~/assembler$ gcc simple_printf_64.c -o simple_printf_64_dis
fulmanp@fulmanp-k2:~/assembler$ objdump -d --disassembler-options=intel simple_printf_64_

simple_printf_64_dis:     file format elf64-x86-64



Disassembly of section .init:

[... cut ...]

00000000004004f4 <main>:
  4004f4: 55                    push   rbp
  4004f5: 48 89 e5              mov    rbp,rsp
  4004f8: 48 83 ec 10           sub    rsp,0x10
  4004fc: 48 b8 c5 3c 2b 69 c5  movabs rax,0x3f5437c5692b3cc5
  400503: 37 54 3f
  400506: 48 89 45 f8           mov    QWORD PTR [rbp-0x8],rax
  40050a: b8 1c 06 40 00        mov    eax,0x40061c
  40050f: f2 0f 10 45 f8        movsd  xmm0,QWORD PTR [rbp-0x8]
  400514: 48 89 c7              mov    rdi,rax
  400517: b8 01 00 00 00        mov    eax,0x1
  40051c: e8 cf fe ff ff        call   4003f0 <printf@plt>
  400521: b8 00 00 00 00        mov    eax,0x0
  400526: c9                    leave
  400527: c3                    ret
  400528: 90                    nop
  400529: 90                    nop
  40052a: 90                    nop
  40052b: 90                    nop
  40052c: 90                    nop
  40052d: 90                    nop
  40052e: 90                    nop
  40052f: 90                    nop

[... cut ...]
```

6.3 Excercises

6.3.1 Excercise

Write a program calculating a dot product of two vector (of integers) of fixed size.

Solution

../programs/basic_cpu_instructions/dot_product_cpu_32.asm

```
section .data

fmt_t: db "vec1=%3d,␣vec2=%3d␣res=%3d", 10, 0
fmt_s: db "result␣is␣%d", 10, 0
vec1:  dd  1,  2,  3,  4,  5,  6,  7,  8
vec2:  dd 18, 17, 16, 15, 14, 13, 12, 11
;         18, 34, 48, 60, 70, 78, 84, 88 ; results of multiplication
res:   dd  0                             ; final result - should be 480

section .text

extern printf

global main

main:

  mov ecx, 0             ; Set counter as 0
  mov ebx, 8             ; Set number of iteration
  loop:                             ; do-while loop begin
    mov eax, [vec1 + ecx * 4]       ; Load [ecx] component of vector 1
    imul dword [vec2 + ecx * 4]     ; Multiply eax by [ecx] component of vector 2
                                    ; Result is in EDX:EAX but we take only
                                    ; bottom half of it. The question is:
                                    ; how to compute with all 64 bits?
    add [res], eax                  ; Increase final result

    push ecx                        ; Save ecx before printf call to protect them
                                    ; from destruction


    push  dword [res]               ; Constant pass by value
    push  dword [vec2 + ecx * 4]    ; Constant pass by value
    push  dword [vec1 + ecx * 4]    ; Constant pass by value
    push  fmt_t                     ; Address of format string
    call  printf                    ; Call C function
    add   esp, 16                   ; Pop stack 4*4 bytes

    pop ecx                         ; Restore ecx after printf call

    inc ecx                         ; Increase value of the counter

    cmp ecx, ebx                    ; While condition test
  jne loop                          ; do-while loop end

; Print final result
  push  dword [res]       ; Constant pass by value
  push  fmt_s             ; Address of format string
  call  printf            ; Call C function
  add   esp, 8            ; Pop stack 2*4 bytes

; Exit
  mov   eax, 0            ; Exit code, 0=normal
  ret                     ; Main returns to operating system
; End of the code
```

Compare this code with code generated from file

../programs/basic_cpu_instructions/dot_product_cpu_32.c

```
#include <stdio.h>

int main(){
  int vec1[] = {  1,  2,  3,  4,  5,  6,  7,  8};
  int vec2[] = { 18, 17, 16, 15, 14, 13, 12, 11};
  int res = 0;
  int i = 0;

  for(i=0;i<8;++i){
    res += vec1[i] * vec2[i];
    printf("vec1=%3d,␣vec2=%3d␣res=%3d\n", vec1[i], vec2[i], res);
  }

  printf("result␣is␣%d\n", res);

  return 0;
}
```

by GCC

../programs/basic_cpu_instructions/dot_product_cpu_32.s

```
    .file "dot_product_cpu_32.c"
    .intel_syntax noprefix
    .section .rodata
.LC0:
    .string  "vec1=%3d,␣vec2=%3d␣res=%3d\n"
.LC1:
    .string  "result␣is␣%d\n"
    .text
    .globl   main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    push  rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    mov   rbp, rsp
    .cfi_def_cfa_register 6
    sub     rsp, 80
    mov     DWORD PTR [rbp-80], 1
    mov     DWORD PTR [rbp-76], 2
    mov     DWORD PTR [rbp-72], 3
    mov     DWORD PTR [rbp-68], 4
    mov     DWORD PTR [rbp-64], 5
    mov     DWORD PTR [rbp-60], 6
    mov     DWORD PTR [rbp-56], 7
    mov     DWORD PTR [rbp-52], 8
    mov     DWORD PTR [rbp-48], 18
    mov     DWORD PTR [rbp-44], 17
    mov     DWORD PTR [rbp-40], 16
    mov     DWORD PTR [rbp-36], 15
    mov     DWORD PTR [rbp-32], 14
    mov     DWORD PTR [rbp-28], 13
    mov     DWORD PTR [rbp-24], 12
    mov     DWORD PTR [rbp-20], 11
    mov     DWORD PTR [rbp-8], 0
    mov     DWORD PTR [rbp-4], 0
    mov     DWORD PTR [rbp-4], 0
    jmp     .L2
.L3:
    mov     eax, DWORD PTR [rbp-4]
    cdqe
    mov     edx, DWORD PTR [rbp-80+rax*4]
    mov     eax, DWORD PTR [rbp-4]
    cdqe
    mov     eax, DWORD PTR [rbp-48+rax*4]
    imul    eax, edx
    add     DWORD PTR [rbp-8], eax
    mov     eax, DWORD PTR [rbp-4]
    cdqe
    mov     edx, DWORD PTR [rbp-48+rax*4]
    mov     eax, DWORD PTR [rbp-4]
    cdqe
    mov     esi, DWORD PTR [rbp-80+rax*4]
    mov     eax, OFFSET FLAT:.LC0
    mov     ecx, DWORD PTR [rbp-8]
    mov     rdi, rax
    mov     eax, 0
    call    printf
    add     DWORD PTR [rbp-4], 1
.L2:
    cmp     DWORD PTR [rbp-4], 7
    jle     .L3
    mov     eax, OFFSET FLAT:.LC1
    mov     edx, DWORD PTR [rbp-8]
    mov     esi, edx
    mov     rdi, rax
    mov     eax, 0
    call    printf
    mov     eax, 0
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size  main, .-main
    .ident   "GCC:␣(Ubuntu/Linaro␣4.6.3-1ubuntu5)␣4.6.3"
    .section .note.GNU-stack,"",@progbits
```

6.3.2 Excercise

Write a program to cipher data with XOR cipher.

Solution

../programs/basic_cpu_instructions/xor_cipher_32.asm

```
section .data

fmt_t: db "%3d␣%3d␣%3d␣(%c)␣xor␣%3d␣(%c)␣=␣%3d", 10, 0;
tte:   db "The␣secret␣text␣to␣encrypt", 10, 0   ; text to encrypt
ttel:  equ $ - tte - 2                          ; tte length
pass:  db "password", 10, 0
passl: equ $ - pass - 2

section .text
```

```
extern printf

global main

main:

  xor edx, edx

  mov ebx, ttel                ; Set max number of iterations
  xor ecx, ecx                 ; Set text counter as 0
rpc:                           ; Reset password counter
  xor eax, eax                 ; Set password counter as 0
loop:
    mov dl, [tte + ecx]
    xor dl, [pass + eax]




    push ecx                   ; Save ECX and EAX before printf call to protect
    push eax                   ; them from destruction

    push  dword edx            ; XOR result
    push  dword [pass + eax]   ; Second argument of XOR
    push  dword [pass + eax]   ; ASCII code of the second argument
    and   dword [esp], 000000FFh ; Cut the least significant byte
    push  dword [tte + ecx]    ; First argument of XOR
    push  dword [tte + ecx]    ; ASCII code of the first argument
    and   dword [esp], 000000FFh;
    push  dword eax
    push  dword ecx
    push  fmt_t                ; Address of format string
    call  printf               ; Call C function
    add   esp, 32              ; Pop stack 8*4 bytes

    pop eax                    ; Restore registers after printf call
    pop ecx
    inc eax
    inc ecx
    cmp eax, passl
    je rpc

    cmp ecx, ebx                ; While condition test
jne loop                        ; do-while loop end

; Exit
  mov   eax, 0                  ; Exit code, 0=normal
  ret                           ; Main returns to operating system
; End of the code
```

6.3.3 Excercise

Modify code from the last excercise to get function allows to crypr / encrypt message*.

Solution

../programs/basic_cpu_instructions/xor_cipher_32.asm

```
section .data

fmt_t: db "%3d␣%3d␣%3d␣(%c)␣xor␣%3d␣(%c)␣=␣%3d", 10, 0;
tte:   db "The␣secret␣text␣to␣encrypt", 10, 0  ; text to encrypt
ttel:  equ $ - tte - 2                          ; tte length
pass:  db "password", 10, 0
passl: equ $ - pass - 2

section .text

extern printf

global main

main:

  xor edx, edx
```

*In the XOR cipher case exactly the same code is used to crypt / encrypt message

```
  mov ebx, ttel                    ; Set max number of iterations
  xor ecx, ecx                     ; Set text counter as 0
rpc:                               ; Reset password counter
  xor eax, eax                     ; Set password counter as 0
loop:
    mov dl, [tte + ecx]
    xor dl, [pass + eax]




    push ecx                      ; Save ECX and EAX before printf call to protect
    push eax                      ; them from destruction

    push  dword edx               ; XOR result
    push  dword [pass + eax]  ; Second argument of XOR
    push  dword [pass + eax]  ; ASCII code of the second argument
    and   dword [esp], 000000FFh ; Cut the least significant byte
    push  dword [tte + ecx]   ; First argument of XOR
    push  dword [tte + ecx]   ; ASCII code of the first argument
    and   dword [esp], 000000FFh;
    push  dword eax
    push  dword ecx
    push  fmt_t                   ; Address of format string
    call  printf                  ; Call C function
    add   esp, 32                 ; Pop stack 8*4 bytes

    pop eax                       ; Restore registers after printf call
    pop ecx

    inc eax
    inc ecx
    cmp eax, passl
    je rpc

    cmp ecx, ebx                  ; While condition test
jne loop                          ; do-while loop end
```

```
; Exit
  mov    eax, 0                ; Exit code, 0=normal
  ret                          ; Main returns to operating system
; End of the code
```

ROZDZIAŁ 7

FPU

A must read document about FPU, like any other aspect of the Intel architecture, is [4]. Here only some kind of summary is given, so for detailed description see this document. To compensate this inconvenience more examples of codes would be provided.

The x87 Floating-Point Unit (FPU), also known as a co-processor, used to be an option when the first PCs came on the market. It provides high-performance floating-point processing capabilities and supports the floating-point, integer, and packed BCD integer data types together with the floating-point processing algorithms and exception handling architecture defined in the IEEE Standard 754 for Binary Floating-Point Arithmetic. Modern PCs are now all provided with a co-processor. It is worth to note that although the original PC-XT architecture (especially CPU) has evolved considerably over the years, the FPU itself has not changed apparently during that same period. The entire set of assembler instructions for the FPU is relatively small – the main difficulty is to avoid some of the pitfalls peculiar to the FPU.

The FPU executes instructions from the processor's normal instruction stream. The state of the FPU is independent from the state of the basic execution environment and from the state of SSE/SSE2/SSE3 extensions. However, the FPU and MMX instructions share state because the MMX registers are aliased to the x87 FPU data registers. Therefore, when writing code that uses FPU and MMX instructions, the programmer must explicitly manage the x87 FPU and MMX state.

7.1 FPU internals

The FPU represents a separate execution environment within the IA-32 architecture. This execution environment consists of eight 80-bit data registers (from R0 to R7[*]) and the following special-purpose registers:

- status register (16-bit),
- control register (16-bit),
- tag word register (16-bit),
- last instruction pointer register (48-bit),
- last data (operand) pointer register (48-bit),
- opcode register (11-bit).

7.1.1 FPU Data Registers

The FPU data registers consist of eight 80-bit registers. Values are stored in these registers in the double extended-precision floating-point format

```
77  66      0
98  43      0
SEEEECCCCCCCCC
| |   |
| |   significand or coefficient (64 bits)
| |
| exponent (15 bits)
|
sign (1 bit)
```

When floating-point, integer, or packed BCD integer values are loaded from memory into any of the FPU data registers, the values are automatically converted into double extended-precision floating-point format (if they are not already in that format). When computation results are subsequently

[*]Note that R0-R7 are internal names and can not be used by programmer. Instead fo this ST(0)-ST(7) are used what would be clarified further.

transferred back into memory from any of the x87 FPU registers, the results can be left in the double extended-precision floating-point format or converted back into a shorter floating-point format, an integer format, or the packed BCD integer format.

The eight FPU data registers are treated as a register stack. All addressing of the data registers is relative to the register on the top of the stack. The register number of the current top-of-stack register is stored in the TOP field in the FPU status word. The current TOP register is named as ST(0) or simply ST, and ST(i) is used to specify the i-th register from TOP in the stack where $i = \{0, \ldots, 7\}$.

```
FPU Data Register Stack


   7 xxx
   6 xxx ST(2)
   5 xxx ST(1)
   4 xxx ST(0) <--- TOP: 100
   3 xxx
   2 xxx
   1 xxx
   0 xxx

Growth stack: stack growth from higher register (R7) to lower (R0).
```

Load operations decrement TOP by one and load a value into the new top-of-stack register, and store operations store the value from the current TOP register in memory and then increment TOP by one[†]. We can think about load operation as equivalent to a push and a store operation as equivalent to a pop.

If a load operation is performed when TOP is at 0, register wraparound occurs and the new value of TOP is set to 7. The floating-point stack-overflow exception indicates when wraparound might cause an unsaved value to be overwritten. Many floating-point instructions have several addressing modes that permit the programmer to implicitly operate on the top of the stack, or to explicitly operate on specific registers relative to the TOP.

[†]Note that load and store operations are also available that do not push and pop the stack.

7.1.2 FPU Addressing Modes

- Stack mode. In this mode an instruction is written without any arguments – by default registers ST(0) and ST(1) are used. In this case both arguments are replaced by the result of instruction.

  ```
  FADD --> FADDP ST(1), ST(0) --> ST(1) + ST(0) -> ST(1) and free ST(0)
  ```

- Register mode. In this mode two arguments are used: ST(0) and ST(i).

  ```
  FADD ST(0), ST(i) --> ST(0) + ST(i) -> ST(0)
  FADD ST(i), ST(0) --> ST(i) + ST(0) -> ST(i)
  ```

- Register mode with stack pop. In this mode source argument is on the top of the stack and destination in register ST(i). When instruction is completed, source argument is poped from a stack.

  ```
  FADDP ST(i), ST(0) --> ST(i) + ST(0) -> ST(i) and free ST(0)
  ```

- Mode with memory argument. In this mode source argument is taken from memory and destination is located in ST(0).

  ```
  FADD memory --> ST(0) + memory -> ST(0)
  ```

7.1.3 FPU stack usage example

Typically the stack structure of the FPU registers and instructions are used in the following way. Assume that we want to calculate simple dot product of two vectors: $v_1 = [1.2, 3.4]$ and $v_2 = [5.6, 7.8]$ (and that TOP contains 100 which means that register R4 is the top of the stack). We can do this with code:

```
FLD  [vec1]
FMUL [vec2]
FLD  [vec1 + 8]
FMUL [vec2 + 8]
FADD ST(1)
```

- `FLD [vec1]` This instruction decrements the stack register pointer (TOP) and loads the value 1.2 from memory into ST(0) (physical register R4). At this moment all registers R0-R7, except R4, are empty.

- `FMUL [vec2]`

  The second instruction multiplies the value in ST(0) by the value 5.6 from memory and stores the result in ST(0). At this moment all registers R0-R7, except R4 in which value 6.72 is stored, are empty.

- `FLD [vec1 + 8]`

  The third instruction decrements TOP and loads the value 3.4 in ST(0). At this moment only registers R4 (in which value 6.72 is stored) and R3 (with value 3.4) are nonempty.

- `FMUL [vec2 + 8]`

  The fourth instruction multiplies the value in ST(0) by the value 7.8 from memory and stores the result in ST(0). At this moment only registers R4 (in which value 6.72 is stored) and R3 (with value 26.52) are nonempty.

- `FADD ST(1)`

  The fifth instruction adds the value from ST(0) and the value from ST(1) and stores the result in ST(0). At this moment only registers R4 (in which value 6.72 is stored) and R3 (with value 33.24) are nonempty.

  If we use for example `FADDP` (which adds ST(0) to ST(1), store result in ST(1), and pop the register stack) the only nenempty registers would be R4 (with value 33.24) referenced as ST(0).

7.2 FPU Status Register

The 16-bit FPU status register indicates the current state of the floating-point unit. The FPU sets the flags in this register to show the results of operations.

7.2.1 Exception Flags

- `IE`, Invalid Operation, bit 0

- `DE`, Denormalized Operand, bit 1

- `ZE`, Zero Divide, bit 2

- `OE`, Overflow, bit 3

- `UE`, Underflow, bit 4

- `PE`, Precision, bit 5

- `SF`, Stack Fault Flag, bit 6

  The stack fault flag indicates that stack overflow or stack underflow has occurred. The FPU explicitly sets the SF flag when it detects a stack overflow or underflow condition, but it does not explicitly clear the flag when it detects an invalid-arithmetic-operand condition. When this flag is set, the condition code flag C1 indicates the nature of the fault: overflow (C1 = 1) and underflow (C1 = 0). The SF flag is a "sticky" flag, meaning that after it is set, the processor does not clear it until it is explicitly instructed to do so (for example, by an FINIT/FNINIT).

- `ES`, Error Summary Status, bit 7

- `C0`,...,`C3`, Condition Code, bit 8, 9, 10 and 14

  The four condition code flags (C0 through C3) indicate the results of floating-point comparison and arithmetic operations. These condition code bits are used principally for conditional branching and for storage of information used in exception handling.

- `TOP`, Top of Stack (TOP) Pointer, bits 11 through 13

  TOP is a pointer to the FPU data register that is currently at the top of the FPU register stack. This pointer is a binary value from 0 to 7.

- `B`, FPU busy, bit 15

Rounding Mode	RC Field Setting (binary)	Description
Round to nearest (even)	00	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (that is, the one with the least-significant bit of zero). Default mode.
Round down	01	Rounded result is closest to but no greater than the infinitely precise result.
Round up	01	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

Tabela 7.1: Rounding Modes and Encoding of Rounding Control (RC) Field

7.3 FPU Control Register

The 16-bit control word controls the precision of the x87 FPU, rounding method used and also contains the FPU floating-point exception mask bits.

Bits 0 through 5 are exception mask bits.

The precision-control (PC) field (bits 8 and 9 of the FPU control word) determines the precision (64, 53, or 24 bits) of floating-point calculations made by the FPU. By default precision double extended precision, which uses the full 64-bit significand, is used.

The rounding-control (RC) field of the FPU control register (bits 10 and 11) controls how the results of FPU floating-point instructions are rounded (see table 7.1).

Bits 6,7 and 13-15 are not used.

7.4 FPU Tag Word Register

The 16-bit tag word indicates the contents of each the 8 registers in the FPU data-register stack (one 2-bit tag per register). The tag codes indicate whether a register contains a valid number (00), zero (01), or a special floating-point number as NaN, infinity, denormal, or unsupported format (10), or whether it is empty (11).

7.5 Examples

7.5.1 Instructions related to the FPU internals

Listing 7.1: ../programs/fpu/fpu_test_01_32.asm

```
section .data

fmt: db 10,"exception:␣%d",10,"top:␣%d",10,"R7␣%d",10,"R6␣%d",10,"R5␣%d",
        10,"R4␣%d",10,"R3␣%d",10,"R2␣%d",10,"R1␣%d",10,"R0␣%d", 10, 0

section .bss

env: resd 7                  ; We will need 28 bytes for saving the current
                             ; FPU operating environmen

section .text

extern printf

global main

main:

  finit                      ; Initialize FPU
  fld1                       ; Push +1.0 onto the FPU register stack.
  fld1
  fld1
  fld1
  call aux_print             ; Call auxiliary print code
  faddp st3, st0             ; Add ST(0) to ST(i) (in this case i=3),
                             ; store result in ST(i), and pop the
                             ; register stack.
  call aux_print

; Exit
  mov   eax, 0               ; Exit code, 0=normal
  ret                        ; Main returns to operating system

; Auxiliary print code
aux_print:
  fstenv [env]               ; Saves the current FPU operating environment
                             ; at the memory location specified with
                             ; the destination operand
```

```
 xor eax, eax
 mov ax, [env+8]              ; Copy to AX contents of the FPU tag word

 mov ecx, 0                   ; Set counter as 0

 loop:                        ; do-while loop begin
   mov ebx, eax
   and ebx, 3                 ; Extract bits 0 and 1
   shr eax, 2                 ; Shift right to extract next two bits
                              ; in next iteration
   push ebx                   ; Save extracted two bits on the stack

   inc ecx                    ; Increase value of the counter
   cmp ecx, 8                 ; While condition test
 jne loop                     ; do-while loop end



 xor eax, eax                 ; Clear eax register
 fstsw ax                     ; Save status word
 mov ebx, eax
 shr bx, 11                   ; Shift ax right by 11 to get top-of-stack
                              ; (TOP) pointer value
 and bx, 7                    ; A bit-wise AND of the two operands:
                              ; BX and binary pattern 111
 push ebx                     ; Save TOP on the stack

 mov ebx, eax                 ; Prepare to extract some exceptions flags
        ;xxxxxxxxx1xxxxx1       bit 7 - Stack Fault          (64 decimal)
                              ; bit 1 - Invalid Operation    ( 1 decimal)
 and bx, 0000000001000001b    ; A bit-wise AND of the two operands:
                              ; BX and binary pattern 1
 push ebx                     ; Save some status word's bits on the stack

 push  fmt                    ; Address of format string
 call  printf                 ; Call C function
 add   esp, 44                ; Pop stack 11*4 bytes
 ret

; End of the code
```

The code should be easy to understand thanks to comments. Bellow an extended information about some parts are presented.

- `finit`

  FINIT sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

- `fld1`

  FLDX where X is one of the following values: 1, L2T, L2E, PI, LG2, LN2, Z push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include $+1.0$ (1), $+0.0$ (Z), $log_2 10$ (L2T), $log_2 e$ (L2E), π (PI), $log_{10} 2$ (LG2), and $log_e 2$ (LN2).

- `faddp st3, st0`

  Adds the destination and source operands and stores the sum in the destination location. In this case `FADDP ST(i), ST(0)` (for i=3) add ST(0) to ST(i), store result in ST(i), and pop the register stack.

  The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format. Please check [4] for reference to other floating point add instruction (FADD/FADDP/FIADD).

- `fstenv`

  Instruction `fstenv` saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU

  – control word,

  – status word,

  – tag word,

  – instruction pointer,

– data pointer,

– and last opcode.

Figures 8-9 through 8-12 in [4], show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. According to this 14 or 28 bytes are needed to save all values.

- `mov ax, [env+8]`

  Copy to AX contents of the FPU tag word. Next we will extract every 2-bits pair and associate it with floating-point register.

7.5.2 FPU control word usage

This code also should be easy to follow. Please make some test with rounding and find some examples how it works.

Listing 7.2: ../programs/fpu/fpu_test_02_32.asm

```
section .data

fmt: db "result␣is␣%d", 10, 0
a:   dq  2.5
b:   dq  3.0

section .bss

tmp: resq 1
buf: resw 1

section .text

extern printf

global main

main:

  finit               ; Initialize FPU
  fstcw [buf]         ; Save control word
                 ;xxxx11xxxxxxxxxx
  or word [buf], 0000110000000000b ; Bits 11-10 controls rounding:
                      ; 00 round to nearst (def),
                      ; 01 round down,
                      ; 10 round up,
                      ; 11 round toward zero
  fldcw [buf]         ; Load updated control word

  fld  qword [a]      ; Load a to FPU
  fmul qword [b]      ; Multiply by b
  fist dword [tmp]    ; Cast result to int

  push  dword [tmp]
  push  fmt
  call  printf
  add   esp, 8

; Exit
  mov   eax, 0        ; Exit code, 0=normal
  ret                 ; Main returns to operating system
; End of the code
```

7.5.3 FPU status word usage

Listing 7.3: ../programs/fpu/fpu_test_03_32.asm

```
section .data

fmt: db "status␣word␣value␣%d", 10, 0
a:   dq  2.5
b:   dq  0.0

section .bss

tmp: resq 1
buf: resw 1

section .text
```

```
extern printf

global main

main:

  finit                 ; Initialize FPU
  fld   qword [a]       ; Load a to FPU
  fdiv  qword [b]       ; Divide by b

  xor eax, eax
  fstsw ax              ; Stores the current value of the FPU status word
                        ; in the destination location. The destination
                        ; operand can be either a two-byte memory location
                        ; or the AX register.

  push  eax
  push  fmt
  call  printf
  add   esp, 8

; Exit
  mov   eax, 0          ; Exit code, 0=normal
  ret                   ; Main returns to operating system
; End of the code
```

Result of execution is below

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf fpu_test_03_32.asm
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 fpu_test_03_32.o -o fpu_test_03_32
fulmanp@fulmanp-k2:~/assembler$ ./fpu_test_03_32
status word value 14340
```

Decimal value 14340 is equal to binary 11100000000100 which means that the ZE (Zero Divide) flag was set. Also we can see that TOP has decimal value 7 (111 binary).

7.5.4 FPU stack overflow

Next program tests what will happend when we try to load into FPU more than 8 numbers.

Listing 7.4: ../programs/nie_moje/kk_fpu_overflow.asm

```
; Author:      Konrad Kosmatka
; Assemble:    nasm -f elf kk_fpu_overflow.asm -o kk_fpu_overflow.o
; Link:        gcc -m32 -o kk_fpu_overflow kk_fpu_overflow.o
; Run:         ./kk_fpu_overflow

global main
extern printf

section .data
fmt: db "x=%f", 10, 0
v1:  dq  1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
len: dd  10

section .bss
tmp: resq 1

section .text

main:
  finit
  xor eax, eax
  push eax

loop:
  pop eax
  fld qword [v1+8*eax]
  cmp  eax, [len]
  je    exit
  inc  eax
  push eax
  fst qword [tmp]
  push dword [tmp+4]
  push dword [tmp]
  push fmt
  call printf
  add   esp, 12
  jmp loop

exit:
  mov eax, 0
  ret
```

The result is

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf kk_fpu_overflow.asm
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 kk_fpu_overflow.o -o kk_fpu_overflow
fulmanp@fulmanp-k2:~/assembler$ ./kk_fpu_overflow
x=1.000000
x=2.000000
x=3.000000
x=4.000000
x=5.000000
x=6.000000
x=7.000000
x=-nan
x=-nan
x=-nan
```

Note that only 7 values are stored correctly although we have 8 registers. The question is: *why*?

Below are some premises. First notice that if we replace

```
fmt: db "x=%f", 10, 0
```

by

```
fmt: db "x=%X %X", 10, 0
```

everything seems to be ok

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf kk_fpu_overflow_hex.asm -o kk_fpu_overflow_he
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 -o kk_fpu_overflow_hex kk_fpu_overflow_hex.o
fulmanp@fulmanp-k2:~/assembler$ ./kk_fpu_overflow_hex
```

```
x=0 3FF00000
x=0 40000000
x=0 40080000
x=0 40100000
x=0 40140000
x=0 40180000
x=0 401C0000
x=0 40200000
x=0 FFF80000
x=0 FFF80000
```

The same correct behaviour is when real numbers are replaced by integers. To verify this replace

```
tmp: resq 1
...
fmt: db "x=%f", 10, 0
...
  fst qword [tmp]
  push dword [tmp+4]
  push dword [tmp]
  push fmt
```

by

```
tmp: resd 1
...
fmt: db "x=%d", 10, 0
...
  fist dword [tmp]
  push dword [tmp]
  push fmt
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf kk_fpu_overflow_int.asm -o kk_fpu_overflow_in
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 -o kk_fpu_overflow_int kk_fpu_overflow_int.o
fulmanp@fulmanp-k2:~/assembler$ ./kk_fpu_overflow_int
```

```
x=1
x=2
x=3
x=4
x=5
x=6
x=7
x=8
x=-2147483648
x=-2147483648
```

What is surprising if we change the first code we used for testing stack overflow (listing 7.4) into 64-bit code

Listing 7.5: ../programs/nie_moje/kk_fpu_overflow64.asm

```
; Author:     Konrad Kosmatka
; Assemble:   nasm -f elf64 kk_fpu_overflow64.asm -o kk_fpu_overflow64.o
; Link:       gcc -o kk_fpu_overflow64 kk_fpu_overflow64.o
; Run:        ./kk_fpu_overflow64

global main
extern printf

section .data
fmt: db "x=%f", 10, 0
v1:  dq  1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0
len: dd  10

section .bss
tmp: resq 1

section .text

main:
  finit
  xor rax, rax
  push rax
```

```
loop:
  pop rax
  fld qword [v1+8*rax]
  cmp  rax, [len]
  je   exit
  inc  rax
  push rax
  fst qword [tmp]
  mov rdi, fmt
  mov rax, 1
  movq xmm0, qword [tmp]
  call printf
  jmp loop

exit:
  mov rdi, 0    ; return exit code (0=normal)
  mov rax, 60   ; system call number (sys_exit)
  syscall
```

the results are correct.

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 kk_fpu_overflow64.asm -o kk_fpu_overflow64.
fulmanp@fulmanp-k2:~/assembler$ gcc -o kk_fpu_overflow64 kk_fpu_overflow64.o
fulmanp@fulmanp-k2:~/assembler$ ./kk_fpu_overflow64
x=1.000000
x=2.000000
x=3.000000
x=4.000000
x=5.000000
x=6.000000
x=7.000000
x=8.000000
x=-nan
x=-nan
```

7.6 Excercises

7.6.1 Excercise

Write a program calculating a dot product of two fixed size vectors (of floating points components).

Solution

Listing 7.6: ../programs/fpu/dot_product_fpu_32.asm

```
section .data

fmt_t: db "vec1=%6.3f,␣vec2=%6.3f␣res=%6.3f", 10, 0
fmt_s: db "result␣is␣%6.3f", 10, 0
vec1:  dq  1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0
vec2:  dq 18.0, 17.0, 16.0, 15.0, 14.0, 13.0, 12.0, 11.0
;         18.0, 34.0, 48.0, 60.0, 70.0, 78.0, 84.0, 88.0 ; results of mul.
res:   dq  0.0            ; final result - should be 480.0

section .bss              ; The data segment containing statically-allocated
                          ; variables - free space allocated for the future use

flttmp: resq 1            ; Statically-allocated variables without an explicit
                          ; initializer; 64-bit temporary for printing flt1

section .text

extern printf

global main

main:

  mov ecx, 0                     ; Set counter as 0
  mov ebx, 8                     ; Set number of iteration

  finit                          ; Initialize FPU

  loop:                          ; do-while loop begin
    fld qword [vec1 + ecx * 8]   ; Load [ecx] component of vector 1
    fmul qword [vec2 + ecx * 8]   ; Multiply by [ecx] component of vector 2
    fadd                          ; Increase final result

    fst  qword [flttmp]           ; Floating load makes 80-bit, store as 64-bit

    push ecx                      ; Save ecx before printf call to protect them
                                  ; from destruction

                                  ; Prepare data for printing partial
                                  ; dot product results

    push  dword [flttmp+4]        ; 64 bit floating point (bottom)
    push  dword [flttmp]          ; 64 bit floating point (top)

    push  dword [vec2 + ecx * 8 + 4] ; 64 bit floating point (bottom)
    push  dword [vec2 + ecx * 8]  ; 64 bit floating point (top)

    push  dword [vec1 + ecx * 8 + 4] ; 64 bit floating point (bottom)
    push  dword [vec1 + ecx * 8]  ; 64 bit floating point (top)

    push  fmt_t                   ; Address of format string
    call  printf                  ; Call C function
    add   esp, 28                 ; Pop stack 7*4 bytes

    pop ecx                       ; Restore ecx after printf call

    inc ecx                       ; Increase value of the counter

    cmp ecx, ebx                  ; While condition test
  jne loop                        ; do-while loop end

; Print final result
  push  dword [flttmp+4]          ; 64 bit floating point (bottom)
  push  dword [flttmp]            ; 64 bit floating point (top)
  push  fmt_s                     ; Address of format string
  call  printf                    ; Call C function
  add   esp, 12                   ; Pop stack 3*4 bytes

; Exit
```

```
  mov    eax, 0                          ; Exit code, 0=normal
  ret                                    ; Main returns to operating system
; End of the code
```

ROZDZIAŁ 8

File operations

8.1 File operations with Linux system calls

Before you start, please check the table 8.1 on the page 136 where you can find important file Linux system calls for 32-bit and 64-bit x86 (notice that system call numbers are different for 32-bit and 64-bit but fortunately the order of arguments stays the same).

Now let's take a look at some source code listed in 8.1.

Listing 8.1: ../programs/files/file_base_64.asm

```
; assemble:       nasm -f elf64 file_name.asm
; link:           ld file_name.o -o file_name
; run:            ./file_name

section .data

text:     db "Running...", 10
len_text: equ $-text

err1:     db "Cannot open input file", 10, 0
len_err1: equ $-err1

err2:     db "Cannot open output file", 10, 0
len_err2: equ $-err2

file_in:  db "data_in.txt", 0
```

Name	EAX (RAX)	EBX (RDI)	ECX (RSI)	EDX (RDX)	Description
chdir	12 (80)	directory name (null terminated)			Changes the current directory.
close	6 (3)	file descriptor			Closes the given file descriptor.
lseek	19 (8)	file descriptor	offset	mode	Changes current position in the given file. The mode should be 0 for absolute positioning or 1 for relative positioning.
mkdir	39 (83)	directory name (null terminated)	permission	mode	Creates the given directory.
open	5 (2)	file name (null terminated)	option list	permission mode	Opens the given file. Returns the file descriptor or an error number.
read	3 (0)	file descriptor	buffer start	buffer size	Reads into the given buffer.
rmdir	40 (84)	directory name (null terminated)			Removes the given directory.
write	4 (1)	file descriptor	buffer start	buffer size	Writes the buffer to the file descriptor.

Tabela 8.1: Important file Linux system calls for 32-bit x86 (and 64-bit in parenthesis).

```
file_out: db "data_out.txt", 0


buffer:    db 'Hello,␣world!'
buf_size: equ $-buffer


section .bss
fd_in      resb 1
fd_out     resb 1


section .text


global  _start


_start:
```

```
; ==========================================================
; simple info print
    mov   rdx, len_text  ; arg3: length of string to print
    mov   rsi, text      ; arg2: pointer to string
    mov   rdi, 1         ; arg1: where to write, so called file handler
                         ; in this case stdout (screen)
    mov   rax, 1         ; System call number (sys_write)
    syscall              ; Call a system function

; ==========================================================
; open input file
    ;mov rdx, len         ; arg3: permission mode
                         ; mode specifies the mode to use in case a new file
                         ; is created.
                         ; This argument must be supplied when O_CREAT or O_TMPFILE is
                         ; specified in flags; if neither O_CREAT nor O_TMPFILE is
                         ; specified, then mode is ignored.
    mov rsi, 0           ; arg2: option list:
                         ; from fcntl.h
                         ; http://lxr.free-electrons.com/source/include/uapi/asm-generic/fcntl.h
                         ; IMPORTANT: use correct (for your system) fcntl.h file
                         ; O_RDONLY 00000000 /* open for reading only */
    mov rdi, file_in     ; arg1: pointer to file name
    mov rax, 2           ; System call number (sys_open)
    syscall              ; Call a system function

    cmp rax, 0           ; check if file descriptor in rax is greater than 0 (ok)
    jle error1           ; cannot open file

    mov [fd_in], rax     ; store file descriptor of input file

; ==========================================================
; open output file
    mov rdx, 0700o       ; arg3: permission mode (octal)
                         ; rwx------ = 700
                         ; The first character (0xxx) controls the SUID, SGID,
                         ; and Stickbit.
                         ; We want to set if off, so use 0.
    mov rsi, 2101q       ; arg2: option list:
```

```
                          ; from fcntl.h
                          ; O_WRONLY 00000001 /* open for writing only */
                          ; O_CREAT  00000100 /* create if nonexistant */
                          ; O_APPEND 00002000 /* set append mode */
   mov rdi, file_out      ; arg1: pointer to file name
   mov rax, 2             ; System call number (sys_open)
   syscall                ; Call a system function

   cmp rax, 0             ; check if file descriptor in rax is greater than 0 (ok)
   jle error2             ; cannot open file

   mov [fd_out], rax      ; store file descriptor of output file

; ==================================================
; write data to a file
    mov rax, 1             ; System call number (sys_write)
    mov rdi, [fd_out]      ; arg1: file descriptor
    mov rsi, buffer        ; arg2: message address
    mov rdx, buf_size      ; arg3: buffer length
    syscall

; ==================================================
; close input file
   mov rdi, [fd_in]       ; arg1: pointer to file name
   mov rax, 3             ; System call number (sys_close)
   syscall                ; Call a system function

; ==================================================
; close output file
   mov rdi, [fd_out]      ; arg1: pointer to file name
   mov rax, 3             ; System call number (sys_close)
   syscall                ; Call a system function

   jmp exit

; ==================================================
; error section
error1: ; Cannot open input file
   mov  rdx, len_err1     ; arg3: length of string to print
```

```
    mov   rsi, err1        ; arg2: pointer to string
    mov   rdi, 1           ; arg1: where to write, so called file handler
                           ; in this case stdout (screen)
    mov   rax, 1           ; System call number (sys_write)
    syscall                ; Call a system function
    jmp  exit

error2: ; Cannot open output file
    mov   rdx, len_err2  ; arg3: length of string to print
    mov   rsi, err2        ; arg2: pointer to string
    mov   rdi, 1           ; arg1: where to write, so called file handler
                           ; in this case stdout (screen)
    mov   rax, 1           ; System call number (sys_write)
    syscall                ; Call a system function
    jmp  exit

; ==================================================
; finall section - exit
exit:
    mov       rdi, 0       ; Exit code, 0=normal
    mov       rax, 60      ; System call number (sys_exit)
    syscall                ; Call a system function
; End of the code
```

How this code works now?

1. Print simple message: *Runinng...*
2. Try to open input file in read only mode. If this fails, jump to error section, print error message and finish the program.
3. Try to open output file in write only mode. If a file exists, a file pointer is set at the end to append new data. If a file not exists, should be ceated. If this procedure fails, jump to error section, print error message and finish the program.
4. Write data to a file. In this case a simple message *Hello, world!* is used.
5. Close input, outout files, jump over error section and finish th program.

```
fulmanp@fulmanp-k2:~/assembler$ ls -l | grep txt
```

```
fulmanp@fulmanp-k2:~/assembler$ echo 'Test message in input file.' > data_in.txt
fulmanp@fulmanp-k2:~/assembler$ ls -l | grep txt
-rw-rw-r-- 1 fulmanp fulmanp   28 maj 31 10:54 data_in.txt
fulmanp@fulmanp-k2:~/assembler$ cat data_in.txt
Test message in input file.
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 file_base_64.asm
fulmanp@fulmanp-k2:~/assembler$ ld file_base_64.o -o file_base_64
fulmanp@fulmanp-k2:~/assembler$ ./file_base_64
Running...
fulmanp@fulmanp-k2:~/assembler$ ls -l | grep data
-rw-rw-r-- 1 fulmanp fulmanp   28 maj 31 10:54 data_in.txt
-rwx------ 1 fulmanp fulmanp   13 maj 31 12:26 data_out.txt
fulmanp@fulmanp-k2:~/assembler$ ./file_base_64
Running...
fulmanp@fulmanp-k2:~/assembler$ ls -l | grep data
-rw-rw-r-- 1 fulmanp fulmanp   28 maj 31 10:54 data_in.txt
-rwx------ 1 fulmanp fulmanp   26 maj 31 12:26 data_out.txt
fulmanp@fulmanp-k2:~/assembler$ cat data_out.txt
Hello, world!Hello, world!
```

How this code should work? Note that although we have created input file, this file is not used – see excercise section below for fix.

8.1.1 Excercise

Based on code from listing 8.1 write a program which copy contents of input file to output file.

8.2 File operations with C functions

8.3 Command Line Parameters

Listing 8.2: ../programs/files/command_line_32.asm

```
section      .data
```

```
fmt_argc:  db  "Number␣of␣arguments:␣%d", 10, 0
fmt_argv:  db  "Argument␣number:␣%3d␣%s", 10, 0

section     .bss
argc: resd 1
argv: resd 1

section     .text

extern      printf
global      main

main:
    mov ecx, [esp+4]
    mov edx, [esp+8]

    mov [argc], ecx
    mov [argv], edx

    push dword [argc]
    push fmt_argc
    call  printf
    add esp, 8

    mov ecx, [argc]
    mov ebx, 0
print_argv:
    push ecx

    mov   eax, [argv]
    mov   edx, [eax + 4*ebx]
    push  edx
    push  ebx
    push  fmt_argv
    call  printf
    add esp, 12

    add ebx, 1
    pop ecx
    loop  print_argv

    ret
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf command_line_32.asm -o command_line_32.o
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 -o command_line_32 command_line_32.o
fulmanp@fulmanp-k2:~/assembler$ ./command_line_32 foo bar
Number of arguments: 3
Argument number:   0 ./command_line_32
Argument number:   1 foo
Argument number:   2 bar
```

Listing 8.3: ../programs/files/command_line_64.asm

```
section      .data

fmt_argc:   db   "Number␣of␣arguments:␣%d", 10, 0
fmt_argv:   db   "Argument␣number:␣%3d␣%s", 10, 0

section      .bss
argc: resq 1
argv: resq 1

section      .text

extern       printf
global       main

main:
    mov [argc], rdi
    mov [argv], rsi

    mov   rsi, [argc]

    mov   rdi, fmt_argc
    mov   rax, 0
    call  printf
    xor rcx, rcx
    mov rcx, [argc]
    mov rbx, 0
print_argv:

    push rcx

    mov   rax, [argv]
    mov   rdx, [rax + 8*rbx]
    mov   rsi, rbx

    mov   rdi, fmt_argv
    mov   rax, 0
    call  printf

    add rbx, 1
    pop rcx
    loop print_argv

    ret
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 command_line_64.asm  -o command_line_64.o
fulmanp@fulmanp-k2:~/assembler$ gcc -o command_line_64 command_line_64.o
fulmanp@fulmanp-k2:~/assembler$ ./command_line_64 foo bar
Number of arguments: 3
Argument number:   0 ./command_line_64
Argument number:   1 foo
Argument number:   2 bar
```

8.4 Auxiliary code

While working with files with numbers we can treat them as text or binary files. Text are more useful for humans but binary are more handy for computers (and assemblers). To convert text file to binary we can use simple C program.

Listing 8.4: ../programs/files/converter.c

```
#include <stdio.h>

int main(int argc, char ** argv){
  double x;
  FILE *fin, *fout;

  fin = stdin;    //fopen(argv[1],"rt");
  fout = stdout; //fopen(argv[2],"rt");

  if (fin && fout){
    if(!feof(fin))
    while(1){
      fscanf(fin, "%lf", &x);
      if(feof(fin))
        break;
      fwrite (&x , sizeof(double), 1, fout);
    }
  }
}
```

If we assume that we have source file `converter_in.txt`

```
65 66 67
```

then we can run our converter as showned below

```
fulmanp@fulmanp-k2:~/assembler$ cat converter_in.txt | ./converter > converter_out.txt
fulmanp@fulmanp-k2:~/assembler$ xxd converter_out.txt
0000000: 0000 0000 0040 5040 0000 0000 0080 5040  .....@P@......P@
0000010: 0000 0000 00c0 5040                      ......P@
```

The result is a binary file with 24 bytes:

- Bytes 0-7: 0000 0000 0040 5040 represents 64-bit value 6.500000e+01,
- Bytes 8-15: 0000 0000 0080 5040 represents 64-bit value 6.600000e+01,
- Bytes 16-23: 0000 0000 00c0 5040 represents 64-bit value 6.700000e+01.

Now we can read easily such a file directly in our assembler.

8.5 Records

Need examples... See: `http://mirror.easyname.at/nongnu//pgubook/ProgrammingGroundUp-1-0-booksi`
`pdf`, chapter 6

8.6 Excercises

8.6.1 Excercise

Write a program similar to program from excercise 7.6.1 (listing 7.6) for calculating a dot product of two fixed size vectors (of floating points components) but data should be read from a file.

ROZDZIAŁ 9

MMX

9.1 Introduction

The one think we can say about MMX (Multi-Media eXtensions) is that this is not a multipurposes technology. Being more precisely, the set of instruction is very specyfic and is optimized for special type of applications – MMX is useles in other types of programms. For example among 24* instructions defined by MMX there are only three, very specific types of multiplication represented by PMADDWD, PMULHW, PMULLW. Reasons for that a very well explained in [5].

The definition of MMX technology resulted from a joint effort between Intel's microprocessor architects and software developers. A wide range of software applications was analyzed, including graphics, MPEG video, music synthesis, speech compression, speech recognition, image processing, games, video conferencing and more. These applications were broken down to identify the most compute-intensive routines, which were then analyzed in details using advanced computer-aided engineering tools. The results of this extensive analysis showed many common, fundamental characteristics across these diverse software categories. The key attributes of these applications were:

- *Small integer data types (for example: 8-bit graphics pixels, 16-bit audio samples)*
- *Small, highly repetitive loops*
- *Frequent multiplies and accumulates*
- *Compute-intensive algorithms*

*57 taking into account all variants: for example there is PADD mnemonic with three different sufixes – B, W and D, so we have different mnemonic and sometimes different opcodes for the same mnemonic.

- *Highly parallel operations*

MMX technology is designed as a set of basic, general purpose integer instructions that can be easily applied to the needs of the wide diversity of multimedia and communications applications[†]*. The highlights of the technology are*

- *Single Instruction, Multiple Data (SIMD) technique*
- *Eight 64-bit wide MMX registers*
- *Four new data types*
- *24 new instructions*

9.2 Single Instruction, Multiple Data (SIMD) technique

According to Flynn's taxonomy SIMD (Single instruction, multiple data) is one of basic computer architectures. It describes computers with multiple processing elements that perform **the same operation on multiple data points simultaneously**.

```
source1: a3      a2      a1      a0
source2: |   b3  |   b2  |   b1  |   b0
         |   |   |   |   |   |   |   |
         +(o)+   +(o)+   +(o)+   +(o)+
           |       |       |       |
        a3 o b3 a2 o b2 a1 o b1 a0 o b0


Notice that we have different arguments (a0-a3 and b0-b3)
but the same operation o.
```

Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio. Modern CPU designs include SIMD instructions in order to improve the performance of

[†]Generality of this approach is, in my opinion, questionable. For example, MMX support packed doubleword type but either it's impossible to implement dot product on 4-byte integers (very, very possible) or I dont't know how to do it (much less possible).

multimedia use. MMX technology uses the single instruction, multiple data technique for performing arithmetic and logical operations on bytes, words, or doublewords packed into MMX registers. For example, the PADDSW instruction adds 4 signed word integers from one source operand to 4 signed word integers in a second source operand and stores 4 word integer results in a destination operand.

9.3 Eight 64-bit wide MMX registers

The MMX register set consists of eight 64-bit registers, that are used to perform calculations on the MMX packed integer data types (see next section to read about new data types). What is worth to note is that MMX registers are not a new and separate set of registers. As it was mentioned in chapter 7 the MMX and FPU instructions share state because the MMX registers are aliased to the x87 FPU data registers. The most frequently explanation for this design choice are [9]

- *MMX had to substantially improve the performance of multimedia, communications, and other numeric intensive applications*
- *MMX had to be kept independent of the current microarchitectures, so that it would scale easily with future advanced microarchitecture techniques and higher processor frequencies in future Intel processors.*
- *MMX processors had to retain backwards compatibility with non-MMX processors. Software must run without modification on a processor with MMX technology.*
- *They had to ensure the coexistence of of existing applications and new applications using MMX technology.*

This last point is important. Modern processors and operating systems can run multiple applications simultaneously. New applications which used the new MMX instructions had to be able to multitask with any other applications. This put some constraints on the MMX technology definition. They couldn't create a new MMX state or mode (in other words, no new registers) because then operating systems would have needed to be modified to take care of these new additions. The main technique for maintaining compatibility of MMX technology was to "hide" it inside the existing floating-point state and registers (current operating systems and applications are designed to work with the floating-point state). An operating system doesn't need to know if MMX technology is present, since it's hidden in the floating-point state. Applications have to check for the presence of MMX technology, and if it's built into the processor they use the new instructions.

Saying the truth, explanation as mentioned above does not convince me. Notice that SSE instruction set (see chapter 10), which is floating point equivalent of MMX, introduced a physicaly new set of registers. Backward compatibility was an Intel's excuse for inconvinients they gave to programmers. The truth was that with such a design goals a new technology was introduced at the lowest cost.

9.4 New data types

MMX technology introduced the following 64-bit data types to the IA-32 architecture:

- 64-bit packed byte integers — eight packed bytes (eight 8-bit integers)
- 64-bit packed word integers — four packed words (four 16-bit integers)
- 64-bit packed doubleword integers — two packed doublewords (two 32-bit integers)
- 64-bit quadword — one quadword

When performing computer arithmetic, an operation may result in an out-of-range condition, where the true result cannot be represented in the destination format. The MMX technology provides three ways of handling out-of-range conditions:

Wraparound arithmetic With wraparound arithmetic, a true out-of-range result is truncated (that is, the carry or overflow bit is ignored and only the least significant bits of the result are returned to the destination). Wraparound arithmetic is suitable for applications that control the range of operands to prevent out-of-range results. If the range of operands is not controlled, however, wraparound arithmetic can lead to large errors. For example, adding two large signed numbers can cause positive overflow and produce a negative result.

Signed saturation arithmetic With signed saturation arithmetic, out-of-range results are limited to the representable range of signed integers for the integer size being operated on. For example, if positive overflow occurs when operating on signed word integers, the result is "saturated" to 7FFFH, which is the largest positive integer that can be represented in 16 bits; if negative overflow occurs, the result is saturated to 8000H.

Unsigned saturation arithmetic With unsigned saturation arithmetic, out-of-range results are limited to the representable range of unsigned integers for the integer size. So, positive overflow when operating on unsigned byte integers results in FFH being returned and negative overflow results in 00H being returned.

Saturation arithmetic provides an answer for many overflow situations. For example, in color calculations, saturation causes a color to remain pure black or pure white without allowing inversion. It also prevents wraparound artifacts from entering into computations when range checking of source operands it not used. MMX instructions do not indicate overflow or underflow occurrence by generating exceptions or setting flags in the EFLAGS register.

9.5 New instructions

Generaly speaking MMX introduced 24 new instructions, grouped into the following categories:

- Data transfer
- Arithmetic
- Comparison
- Conversion
- Unpacking
- Logical
- Shift
- Empty MMX state instruction (EMMS)

Bellow we show few examples of MMX instructions to give a brief overview of the ideas behind them.

9.5.1 Add packed integers with PADDW

This example shows a packed add word with wrap around.

```
   a3   |    a2   |    a1   | a0=FFFFh|    movq mm0, [ edx ]
   b3   |    b2   |    b1   | b0=0003h|    movq mm1, [ esi ]
 a3 + b3 | a2 + b2 | a1 + b1 | a0 + b0=|    paddw mm0, mm1
        |         |         |  0002h  |
```

It performs four additions of the eight, 16-bit elements, with each addition independent of the others and in parallel. In this case, the rightmost result exceeds the maximum value representable in 16-bits thus it wraps around. FFFFh + 0003h would be a 17-bit result of value 10002. The 17th bit is lost because of wrap around, so the result is 0002.

9.5.2 Multiply and Add Packed Integers with PMADDWD

This example shows the instruction used for multiply-accumulate operations, which is fundamental to many algorithms based on matrix (vectors) multiplication.

```
    a3    |    a2    |    a1    |    a0    |    movq mm0, [ edx ]
    b3    |    b2    |    b1    |    b0    |    movq mm1, [ esi ]
 a3 * b3 + a2 * b2 | a1 * b1 + a0 * b0 |    pmaddwd mm0, mm1
```

PMADDWD multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source (b0 and b1) and destination (a0 and a1) operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0).

9.5.3 Compare packed signed integers for greater than with PCMPGTW

PCMPGTW performs a signed compare for the greater value of the packed word integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

```
    1    |    4    |    5    |    7    |    movq mm0, [ edx ]
    2    |    3    |    6    |    7    |    movq mm1, [ esi ]
  0000h  |  FFFFh  |  0000h  |  0000h  |    pcmpgtw mm0, mm1
```

9.5.4 Pack with signed saturation with PACKSSWB

Converts packed signed word integers into packed signed byte integers (PACKSSWB), using saturation to handle overflow conditions. Converts 4 packed signed word integers from the destination

Instruction	Extension	Description
`PMADDWD mm, mm/m64`	MMX	Multiply the packed words in mm by the packed words in mm/m64, add adjacent doubleword results, and store in mm.
`PMADDWD xmm1, xmm2/m128`	SSE2	Multiply the packed word integers in xmm1 by the packed word integers in xmm2/m128, add adjacent doubleword results, and store in xmm1.
`VPMADDWD xmm1, xmm2, xmm3/m128`	AVX	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1.
`VPMADDWD ymm1, ymm2, ymm3/m256`	AVX2	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1.

Tabela 9.1: PMADDWD variants

operand (first operand) and 4 signed word integers from the source operand (second operand) into 8 packed signed byte integers and stores the result in the destination operand.

```
    a3   |    a2   |    a1   |    a0   |    movq mm0, [ edx ]
    b3   |    b2   |    b1   |    b0   |    movq mm1, [ esi ]
b3' |b2' |b1' |b0' |a3' |a2' |a1' |a0' |    packsswb mm0, mm1
```

If a signed word integer value is beyond the range of a signed byte integer (that is, greater than 7FH for a positive integer or greater than 80H for a negative integer), the saturated signed byte integer value of 7Fh or 80h, respectively, is stored in the destination.

Notice that althought we are talking now about MMX instructions all of them are also a part of further extensions like SSE or AVX. For example PMADDWD has variants described in the table 9.1.

9.6 Examples

On the listing 9.1 usage of the instructions from section 9.5 is presented.

Listing 9.1: ../programs/mmx/mmx_basic_example_64.asm

```
section     .data


vec1:  dw    65534,     2, 65534,     4
vec2:  dw        1, 65534,     3,     4
;          65535,     0,     1,     8 ; results of paddw
; 65534 = 1111 1111 1111 1110 (2) = −2(U2 16−bit)
;              −2,     −4,     −6,     16 ; results of pmaddwd − partial
;                      −6,             10 ; results of pmaddwd − final
;               0, 65535,      0,      0 ; results of pcmpgtw
; 65534 = 1111 1111 1111 1110 (2) = −2(U2 16−bit)
;                     1111 1110 (2) = −2(U2 8−bit)
;          0..x56..0 1111 1110 (2) = 254(U2 64−bit)
; or
;          1..x56..1 1111 1110 (2) = −2(U2 64−bit)
; −2 2 −2 4 1 −2 3 4                  ; packsswb
; but display
; −2 2 −2 4 1 4 3 −2
; because of calling convention

fmt2:   db    "Result:␣%6d,␣%6d", 10, 0
fmt4:   db    "Result:␣%6d,␣%6d,␣%6d,␣%6d", 10, 0
fmt8:   db    "Result:␣%6d,␣%6d,␣%6d,␣%6d,␣%6d,␣%6d,␣%6d,␣%6d", 10, 0

section     .text

extern      printf
global      main

main:

    movq    mm0, [vec1]
    paddw   mm0, [vec2] ; Add Packed Integers

    call printf_4

    movq    mm0, [vec1]
    pmaddwd mm0, [vec2] ; Multiply and Add Packed Integers

    call printf_2

    movq    mm0, [vec1]
    pcmpgtw mm0, [vec2] ; Compare Packed Signed Integers for Greater Than

    call printf_4
    movq      mm0, [vec1]
    packsswb mm0,[vec2]  ; Pack with Signed Saturation

    call  printf_8

    ret

printf_2:
    xor       rax, rax
    movd      eax, mm0
    mov       rsi, rax

    psrlq     mm0, 32

    xor       rax, rax
    movd      eax, mm0
    mov       rdx, rax

    mov       rdi, fmt2
    mov       rax, 0
    call      printf
    ret

printf_4:
    xor       rbx, rbx
    or        rbx, 0FFFFh

    movq      rax, mm0
    and       rax, rbx
    mov       rsi, rax
    psrlq     mm0, 16

    movq      rax, mm0
    and       rax, rbx
    mov       rdx, rax
    psrlq     mm0, 16

    movq      rax, mm0
    and     rax, rbx
    mov     rcx, rax
    psrlq   mm0, 16

    movq    rax, mm0
    and     rax, rbx
    mov     r8,  rax

    mov     rdi, fmt4
    mov     rax, 0
    call    printf
    ret

printf_8:
    xor     rbx, rbx
    or      rbx, 0FFh

    movq    rax, mm0
    and     rax, rbx
    cbw
    cwde
    cdqe
    mov     rsi, rax
    psrlq   mm0, 8

    movq    rax, mm0
    and     rax, rbx
    cbw
    cwde
    cdqe
    mov     rdx, rax
    psrlq   mm0, 8

    movq    rax, mm0
    and     rax, rbx
    cbw
    cwde
    cdqe
    mov     rcx, rax
```

```
psrlq    mm0, 8

movq     rax, mm0
and      rax, rbx
cbw
cwde
cdqe
mov      r8,  rax
psrlq    mm0, 8

movq     rax, mm0
and      rax, rbx
cbw
cwde
cdqe
mov      r9,  rax
psrlq    mm0, 8

; End of registers - stack part begins
; Now args are from right to left

movq     rax, mm0
and      rax, rbx
cbw
cwde
cdqe
push     rax
psrlq    mm0, 8

movq     rax, mm0
and      rax, rbx
cbw
cwde
cdqe
push     rax
psrlq    mm0, 8

movq     rax, mm0
and      rax, rbx
```

```
    cbw
    cwde
    cdqe
    push    rax
    psrlq   mm0, 8

    mov     rdi, fmt8
    mov     rax, 0
    call    printf
    add rsp, 24
    ret
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 mmx_basic_example_64.asm -o mmx_basic_examp
fulmanp@fulmanp-k2:~/assembler$ gcc -o mmx_basic_example_64 mmx_basic_example_64.o
fulmanp@fulmanp-k2:~/assembler$ ./mmx_basic_example_64
Result:  65535,      0,      1,      8
Result:     -6,     10
Result:      0,  65535,      0,      0
Result:     -2,      2,     -2,      4,      1,      4,      3,     -2
```

9.7 Excercise

Write a program calculating a dot product of two vector (of 16-bit integers) of fixed size.

9.7.1 Solution

Taking into account everything we know about MMX, it is not possible to write with MMX equivalent of the code 7.6 from chapter 7 or this equivalen would be very impractical. That's why MMX implementation of dot product would be „tuned" for MMX instruction set and works only for 16-bit integers.

../programs/mmx/dot_product_mmx_32.asm

```
section .data

fmt_t: db "MMX=%d,␣rest=%d", 10, 0
fmt_p_mmx: db "partial␣result␣of␣mmx␣part␣%3d", 10, 0
```

```
fmt_p: db "partial␣result␣of␣non␣mmx␣part␣%3d", 10, 0
fmt_f: db "final␣result␣%3d", 10, 0
vec1:  dw  1,  2,  3,  4,  5,  6,  7,  8,  9, 10
vec2:  dw 18, 17, 16, 15, 14, 13, 12, 11, 10,  9
;         18, 34, 48, 60, 70, 78, 84, 88, 90, 90 ; results of mul.
res:   dd  0      ; final result - should be 660

section .text

extern printf

global main

main:

  mov edx, vec1
  mov esi, vec2
  mov ecx, 10       ; ecx = the number of 16-bit integers

  mov ebx, ecx      ; Copy ecx to ebx
  and ebx, 3        ; We are going to take four 16-bit integers at once
                    ; so we need the number of integers left (remainder
                    ; of division ecx/4) i.e. ebx = ebx % 4
  shr ecx, 2        ; Division by 4 - integer part of division: ecx/4

  push  edx         ; Print integer part and remainder
  push  ecx
  push  ebx
  push  ecx
  push  fmt_t
  call  printf
  add   esp, 12
  pop   ecx
  pop   edx

loop_mmx:
  movq mm0, [edx]      ; Copy four 16-bit integers into MM0 register
  pmaddwd mm0, [esi]
  movd eax, mm0
  psrlq mm0, 32
  movd edi, mm0
  add  eax, edi
  add [res], eax
  add edx, 8        ; Four 16-bit integers = 4 * 2 byte = 8 byte
  add esi, 8

  push  esi         ; Print partial result of MMX part
  push  edx
  push  ecx
  push  ebx
  push  eax
  push  fmt_p_mmx
  call  printf
  add   esp, 8
  pop   ebx
  pop   ecx
  pop   edx
  pop   esi

  loop loop_mmx

  cmp ebx, 0
  je end_nonmmx_part ; if ebx = 0 then jump end_nonmmx_part

  mov ecx, ebx
loop_nonmmx:
  xor eax, eax
  push edx   ; Save EDX to prevent it from destruction by IMUL
  mov ax, [edx]
  imul word [esi] ; Result is in DX:AX
  add [res], eax
  pop edx
  add edx, 2
  add esi, 2

  push  esi             ; Print partial result of non MMX part
  push  edx
  push  ecx
  push   eax
  push   fmt_p
  call   printf
  add    esp, 8
  pop    ecx
  pop    edx
  pop    esi

  loop loop_nonmmx

end_nonmmx_part:

  push   dword [res] ; Print final result
  push   fmt_f
  call   printf
  add    esp, 8

; Exit
  mov    eax, 0          ; Exit code, 0=normal
  ret                    ; Main returns to operating system
; End of the code
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf dot_product_mmx_32.asm -o dot_product_mmx_32.
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 -o dot_product_mmx_32 dot_product_mmx_32.o
fulmanp@fulmanp-k2:~/assembler$ ./dot_product_mmx_32
MMX=2, rest=2
partial result of mmx part 160
partial result of mmx part 320
partial result of non mmx part  90
partial result of non mmx part  90
final result 660
```

There is no difference between 32-bit and 64-bit code; the following code for simplicity focus only on MMX part.

../programs/mmx/dot_product_mmx_64.asm

```
section     .data
```

```
vec1:   dw    1, 2, 3, 4, 5, 6, 7, 8
vec2:   dw    9, 8, 7, 6, 5, 4, 3, 2
;           9,16,21,24,25,24,21,16 ; results of mul.
res:    dd 0 ; final result - should be 159
fmt:    db    "Result:␣%3d", 10, 0


section       .text

extern        printf
global        main

main:
    mov       rcx, 2      ; Integer part of division: len(vec1)/4
                          ; For simplicity we assume no fractional part
    mov       r11, vec1
    mov       r12, vec2

loop_mmx:
    movq      mm0, [r11]
    pmaddwd   mm0, [r12]
    movd      eax, mm0
    psrlq     mm0, 32
    movd      ebx, mm0
    add       eax, ebx
    add       [res], eax
    add       r11, 8
    add       r12, 8
    loop      loop_mmx

    mov       rsi, [res]
    mov       rdi, fmt
    mov       rax, 0
    call      printf
    ret
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 dot_product_mmx_64.asm -o dot_product_mmx_6
fulmanp@fulmanp-k2:~/assembler$ gcc -o dot_product_mmx_64 dot_product_mmx_64.o
```

```
fulmanp@fulmanp-k2:~/assembler$ ./dot_product_mmx_64
Result: 156
```

Better solution (faster) of this excercise could be found in [7]. To verify if it's realy better, reader could use RDTS instruction – see chapter 11.

ROZDZIAŁ 1

SSE

10.1 Streaming SIMD Extensions

Like MMX is tuned for working with bytes or words (8 or 16-bit integers) the SSE (Streaming SIMD Extensions) is tuned for working with single-precision floating-point values.

10.2 Example

Let's take a look into simple SSE instructions example.

../programs/sse/sse_example_01_64.asm

```
global _start

section .data
     fmt:    db "%6.3f %6.3f %6.3f %6.3f", 10, 0
    vec1: dd 1.2, 3.4, 5.6, 7.8 ; packed single-precision floating-point values
    vec2: dd 8.7, 6.5, 4.3, 2.1
; mulps   10.44, 22.1, 24.08, 16.38
; addps   19.14, 28.6, 28.38, 18.48

section .bss
    align 16 ; Need by fxsave because: "The destination operand [of FXSAVE]
             ; contains the first byte of the memory image, and it must
             ; be aligned on a 16-byte boundary. A misaligned destination
             ; operand will result in a general-protection (#GP) exception
```

```
              ; being generated (or in some cases, an alignment check
              ; exception [#AC])."
    fxsave_area: resb 512 ; free space for fxsave structure
    flttmp : resq 1 ; temporary value for conversion from single
                     ; to double precision floating-point number
section .text

extern      printf
global      main

main:

; Move Unaligned Packed Single-Precision Floating-Point Values
    movups xmm0, [vec1]
    call print_xmm
    movups xmm1, [vec2]
; Multiply Packed Single-Precision Floating-Point Values
    mulps xmm0, xmm1
    call print_xmm
; Add Packed Single-Precision Floating-Point Values
    addps xmm0, xmm1
    call print_xmm

    ret

print_xmm:
    fxsave [fxsave_area]
    ; get first 4 bytes from 160-175 bytes of FXSAVE area to get
    ; first single-precision floating-point number from xmm0 register
    fld dword [fxsave_area + 160]
    fstp qword [flttmp]
    movsd xmm0, [flttmp]
    ; get next 4 bytes
    fld dword [fxsave_area + 164]
    fstp qword [flttmp]
    movsd xmm1, [flttmp]
    fld dword [fxsave_area + 168]
    fstp qword [flttmp]
    movsd xmm2, [flttmp]
    fld dword [fxsave_area + 172]
    fstp qword [flttmp]
    movsd xmm3, [flttmp]
    mov   rdi, fmt
    mov rax, 4
    call printf
    fxrstor [fxsave_area]
    ret
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf64 sse_example_01_64.asm -o sse_example_01_64.
fulmanp@fulmanp-k2:~/assembler$ gcc -o sse_example_01_64 sse_example_01_64.o
fulmanp@fulmanp-k2:~/assembler$ ./sse_example_01_64
 1.200  3.400  5.600  7.800
10.440 22.100 24.080 16.380
19.140 28.600 28.380 18.480
```

10.3 Excercise

Write a program calculating a dot product of two vectors (of floating points) of fixed size.

Solution

../programs/sse/dot_product_sse_32.asm

```
section .data

fmt_t: db "SSE=%d,␣rest=%d", 10, 0
fmt_p_sse: db "partial␣result␣on␣sse␣%8.3f␣%8.3f␣%8.3f␣%8.3f", 10, 0
fmt_p: db "partial␣result␣on␣fpu␣%8.3f", 10, 0
fmt_f_sse: db "final␣result␣on␣sse␣␣␣%8.3f␣%8.3f␣%8.3f␣%8.3f", 10, 0
fmt_f: db "final␣result␣%8.3f", 10, 0
vec1:   dd  1.0,  2.0,  3.0,  4.0,  5.0,  6.0,  7.0,  8.0,  9.0, 10.0
vec2:   dd 18.0, 17.0, 16.0, 15.0, 14.0, 13.0, 12.0, 11.0, 10.0,  9.0
; results of mul
;          18.0, 34.0, 48.0, 60.0, 70.0, 78.0, 84.0, 88.0, 90.0, 90.0
res:    dd  0.0     ; final result - should be 660.0

section .bss
```

```
flttmp: resq 1
buf_p: resd 4
buf_s: resd 4

section .text

extern printf

global main

main:

  mov edx, vec1
  mov esi, vec2
  mov ecx, 10       ; ecx = the number of 32-bit floating-point (FP) values

  mov ebx, ecx      ; Copy ecx to ebx
  and ebx, 3        ; We are going to take four 32-bit FP at once so we
                    ; need the number of FP left (remainder of division ecx/4)
                    ; i.e. ebx = ebx % 4
  shr ecx, 2        ; Division by 4 - integer part of division: ecx/4

  push   edx        ; Print integer part and remainder
  push   ecx
  push   ebx
  push   ecx
  push   fmt_t
  call   printf
  add    esp, 12
  pop    ecx
  pop    edx

  xorps xmm7, xmm7
loop_sse:
  movups xmm0, [edx]   ; Copy four 32-bit floating-point values from
                       ; vector 1 into XMM0 register.
  movups xmm1, [esi]   ; Copy four 32-bit floating-point values from
                       ; vector 2 into XMM1 register.
  mulps xmm0, xmm1     ; Multiply of the four packed single-precision
                       ; floating-point values.
  addps xmm7, xmm0     ; Add to final four 32-bit floating-point values
  add edx, 16          ; Four 32-bit floats = 4 * 4 byte = 16 byte
  add esi, 16

  movups [buf_p], xmm0  ; Write back the result of partial multiplication
  movups [buf_s], xmm7  ; Write back the result of accumulated sum

  push edx
  push ecx
; Print partial result of SSE part
; The contents of the XMM registers are printed, so the order (direction) is from
; the right to the left which is a reverse order of the components in our vectors
; (from the left to the right).
; Fourth argument
  fld    dword [buf_p]      ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp   qword [flttmp]
  push   dword [flttmp+4]   ; 64 bit floating point (bottom)
  push   dword [flttmp]     ; 64 bit floating point (top)
; Third argument
  fld    dword [buf_p+4]    ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp   qword [flttmp]
  push   dword [flttmp+4]   ; 64 bit floating point (bottom)
  push   dword [flttmp]     ; 64 bit floating point (top)
; Second argument
  fld    dword [buf_p+8]    ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp   qword [flttmp]
  push   dword [flttmp+4]   ; 64 bit floating point (bottom)
  push   dword [flttmp]     ; 64 bit floating point (top)
; First argument
  fld    dword [buf_p+12]   ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp   qword [flttmp]
  push   dword [flttmp+4]   ; 64 bit floating point (bottom)
  push   dword [flttmp]     ; 64 bit floating point (top)
  push   fmt_p_sse
  call   printf
  add    esp, 36
; Print accumulated sum
; Fourth argument
  fld    dword [buf_s]      ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp   qword [flttmp]
  push   dword [flttmp+4]   ; 64 bit floating point (bottom)
  push   dword [flttmp]     ; 64 bit floating point (top)
; Third argument
  fld    dword [buf_s+4]    ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp   qword [flttmp]
  push   dword [flttmp+4]   ; 64 bit floating point (bottom)
  push   dword [flttmp]     ; 64 bit floating point (top)
; Second argument
  fld    dword [buf_s+8]    ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp   qword [flttmp]
  push   dword [flttmp+4]   ; 64 bit floating point (bottom)
  push   dword [flttmp]     ; 64 bit floating point (top)
; First argument
  fld    dword [buf_s+12]   ; Convert 32-bit to 64-bit via 80-bits FPU stack
  fstp   qword [flttmp]
  push   dword [flttmp+4]   ; 64 bit floating point (bottom)
  push   dword [flttmp]     ; 64 bit floating point (top)
  push   fmt_f_sse
  call   printf
  add    esp, 36

  pop ecx
  pop edx
  ;loop loop_sse ; Only the offsets of -128 to +127 are allowed
                 ; with loop instruction.
  dec ecx
  jnz loop_sse

  fldz                          ; Set FPU to 0
  cmp ebx, 0
  je end_nonsse_part            ; if ebx = 0 then jump end_nonsse_part

  mov ecx, ecx
loop_nonsse:
  fld dword [edx + ecx * 4]     ; Load component of vector 1
  fmul dword [esi + ecx * 4]    ; Multiply by component of vector 2
  fadd                          ; Increase partial fpu result

  fst  qword [flttmp]           ; Floating load makes 80-bit, store as 64-bit

  push ecx                      ; Save registers before printf call to protect
  push edx                      ; them from destruction
  push esi

  push  dword [flttmp+4]        ; 64 bit floating point (bottom)
  push  dword [flttmp]          ; 64 bit floating point (top)

  push  fmt_p                   ; Address of format string
  call  printf                  ; Call C function
  add   esp, 12                 ; Pop stack 7*4 bytes

  pop esi                       ; Restore registers after printf call
  pop edx
  pop ecx

  inc ecx                       ; Increase value of the counter

  cmp ecx, ebx                  ; While condition test
  jne loop_nonsse               ; do-while loop end

end_nonsse_part:

; Combine final result from SSE and FPU part
  fld dword [buf_s]             ; Load component from XMM register bits  0- 31
  fld dword [buf_s+4]           ; Load component from XMM register bits 32- 63
  fld dword [buf_s+8]           ; Load component from XMM register bits 64- 95
  fld dword [buf_s+12]          ; Load component from XMM register bits 96-127
  fadd
  fadd
  fadd
  fadd

  fst  qword [flttmp]           ; Floating load makes 80-bit, store as 64-bit

  push  dword [flttmp+4]        ; 64 bit floating point (bottom)
  push  dword [flttmp]          ; 64 bit floating point (top)

  push  fmt_f                   ; Address of format string
  call  printf                  ; Call C function
  add   esp, 12                 ; Pop stack 7*4 bytes



; Exit
  mov   eax, 0          ; Exit code, 0=normal
  ret                   ; Main returns to operating system
; End of the code
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf dot_product_sse_32.asm -o dot_product_sse_32.
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 -o dot_product_sse_32 dot_product_sse_32.o
fulmanp@fulmanp-k2:~/assembler$ ./dot_product_sse_32
SSE=2, rest=2
partial result on sse   60.000   48.000   34.000   18.000
final result on sse     60.000   48.000   34.000   18.000
partial result on sse   88.000   84.000   78.000   70.000
final result on sse    148.000  132.000  112.000   88.000
partial result on fpu   90.000
partial result on fpu  180.000
final result  660.000
```

When I was preparing this program I encountered the following problem

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf dot_product_sse_32.asm -o dot_product_sse_32.
dot_product_sse_32.asm:96: error: short jump is out of range
```

Why? The SSE loop (starting at `loop_sse:`) is very long – there are many instructions. Intel documentation about LOOP instruction (eg. [4], page 891) says

Each time the LOOP instruction is executed, the count register is decremented, 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 near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of -128 to +127 are allowed with this instruction.

That's why code

```
label:
  loop-body
  loop label
```

works fine, but code

```
label:
  loop-body
  more-code-added
  loop label
```

does not work and error "*short jump out of range*" appears. The solution is obvious. Because the LOOP instruction can't jump to a distance of more than 127 bytes we need to change code to use DEC ECX with JNZ instructions. For example

```
  mov ecx, 10
label:
  loop-body
  loop label
```

become

```
  mov ecx, 10
label:
  loop-body
  more-code-added
  dec ecx
  jnz loop
```

ROZDZIAŁ 1

RDTS – measure what is unmeasurable

11.1 Read time-stamp counter

The Time Stamp Counter (TSC) is a 64-bit register which counts the number of cycles since reset. The instruction RDTSC returns the TSC in EDX:EAX. In x86-64 mode, RDTSC also clears the higher 32 bits of RAX. Its opcode is 0F 31.

Notice that the time-stamp counter measures "cycles" and not "time". For example, two bilions cycles on a 2 GHz processor is equivalent to one second of real time, while the same number of cycles on a 1 GHz processor is two second of real time. Thus, comparing cycle counts only makes sense on processors of the same speed. To compare processors of different speeds, the cycle counts should be converted into time units

$$s = \frac{c}{f}$$

where s is time in seconds, c is the number of cycles and f is the frequency.

11.2 Usage of the RDTS

Prevent from out-of-order execution

Out-of-order execution (see) is a nice feature but impede any optimization activities. We may encounter this problem trying to measure time. That is why the obvious approach showned on listing

Speed [GHz]	Max time for 32-bit counter [s]	Max time for 64-bit counter [days] ([years])
0.5	8.5899	427008 (1169.88)
1	4.2949	213504 (584.942)
1.5	2.8633	142336 (389.962)
2	2.1474	106752 (292.471)
2.5	1.7179	85401 (233.977)
3	1.4316	71168 (194.981)
1	a	b

Tabela 11.1: Maximum TSC value and real time for selected frequencies.

11.1 is not good.

Listing 11.1: ../programs/rdtsc/01.asm

```
rdtsc             ; Read time stamp counter
mov [time], eax   ; Copy counter into variable
...               ; Do something
rdtsc             ; Read time stamp
sub eax, [time]   ; Find the difference
```

Instead of this we have to follow the pattern showned on listing 11.2 where CPUID instruction is used. *CPUID can be executed at any privilege level to serialize instruction execution. Serializing instruction execution guarantees that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed.* ([4], CPUID description). See also [4], "Serializing Instructions" in chapter 8, volume 3A.

Listing 11.2: ../programs/rdtsc/02.asm

```
cpuid             ; Force all previous instructions to complete
rdtsc             ; Read time stamp counter
mov [time], eax   ; Copy counter into variable
...               ; Do something
cpuid             ; Wait for [something] to complete before RDTSC
rdtsc             ; Read time stamp counter
sub eax, [time]   ; Find the difference
```

Now the RDTSC instructions will be guaranteed to complete at the desired time in the execution stream. Of course this approach take into account the cycles it takes for the CPUID instruction to complete, so the programmer must subtract this from the recorded number of cycles. A must know think about the CPUID instruction is that it can take longer to complete the first couple of times it is called. Thus, the best policy is to call the instruction three times, measure the elapsed time on the third call, then subtract this measurement from all future measurements[8].

Caching data nad code

11.2.1 Usage example

Now we will try measure execution time (number of cycles) for base arithmetical instructions for integers.

../programs/rdtsc/rdtsc_ex_02.asm

```
section .data

fmt: db "subtime=%d,␣add=%d␣sub=%d␣mul=%d␣div=%d", 10, 0
x:   dd 6
y:   dd 3

section .bss

subtime: resd 1
t_add: resd 1
t_sub: resd 1
t_mul: resd 1
t_div: resd 1

section .text

extern printf

global main

main:

  ; Make three warm-up passes through the timing routine to make
  ; sure that the CPUID and RDTSC instruction are ready

  cpuid
  rdtsc
 mov [subtime], eax
```

```
  cpuid
  rdtsc
  sub eax, [subtime]
  mov [subtime], eax

  cpuid
  rdtsc
  mov [subtime], eax
  cpuid
  rdtsc
  sub eax, [subtime]
  mov [subtime], eax

  cpuid
  rdtsc
  mov [subtime], eax
  cpuid
  rdtsc
  sub eax, [subtime]
  mov [subtime], eax

  ; Only the last value of subtime is kept
  ; subtime should now represent the overhead cost of the
  ; MOV and CPUID instructions

; ADD
  mov ecx, [x]
  mov ebx, [y]
  cpuid
  rdtsc
  mov [t_add], eax
  add ecx, ebx
  cpuid
  rdtsc
  sub eax, [t_add]
  mov [t_add], eax

; SUB
  mov ecx, [x]
  mov ebx, [y]
  cpuid
  rdtsc
  mov [t_sub], eax
  sub ecx, ebx
  cpuid
  rdtsc
  sub eax, [t_sub]
  mov [t_sub], eax

; MUL
  mov ecx, [x]
  mov ebx, [y]
  cpuid
  rdtsc
  mov [t_mul], eax
  imul ecx, ebx
  cpuid
  rdtsc
  sub eax, [t_mul]
  mov [t_mul], eax

; DIV
  xor edx, edx
  mov ecx, [x]
  mov ebx, [y]
  cpuid
  rdtsc
  mov [t_div], eax
  mov eax, ecx
  ; idiv ebx ; If this line is uncommented have
             ; B  d w obliczeniach zmiennoprzecinkowych (core dumped)
             ; No idea why?!
  cpuid
  rdtsc
  sub eax, [t_div]
  mov [t_div], eax

; Print results
```

```
    push   dword [t_div]
    push   dword [t_mul]
    push   dword [t_sub]
    push   dword [t_add]
    push   dword [subtime]
    push   fmt                   ; Address of format string
    call   printf                ; Call C function
    add    esp, 24               ; Pop stack 7*4 bytes

; Exit
    mov    eax, 0                ; Exit code, 0=normal
    ret                          ; Main returns to operating system
; End of the code
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf rdtsc_ex_02.asm -o rdtsc_ex_02.o
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 rdtsc_ex_02.o -o rdtsc_ex_02
fulmanp@fulmanp-k2:~/assembler$ ./rdtsc_ex_02
subtime=259294, add=2660 sub=5562 mul=2848 div=43133

...

subtime=9274, add=2228 sub=2160 mul=2119 div=1904
subtime=7803, add=2390 sub=2403 mul=2268 div=1782
subtime=8735, add=2403 sub=2363 mul=2362 div=1687
subtime=7655, add=2241 sub=2200 mul=2444 div=1755
subtime=11313, add=2403 sub=2349 mul=2336 div=1782
subtime=7587, add=2241 sub=2228 mul=2228 div=1728
subtime=14823, add=2943 sub=3038 mul=3159 div=2309
```

The same test for floating point numbers.

../programs/rdtsc/rdtsc_ex_01.asm

```
section .data

fmt: db "subtime=%d,␣add=%d␣sub=%d␣mul=%d␣div=%d", 10, 0
```

```
x:    dq 6.0
y:    dq 3.0

section .bss

subtime: resd 1
t_add: resd 1
t_sub: resd 1
t_mul: resd 1
t_div: resd 1

section .text

extern printf

global main

main:

  ; Make three warm-up passes through the timing routine to make
  ; sure that the CPUID and RDTSC instruction are ready

  cpuid
  rdtsc
  mov [subtime], eax
  cpuid
  rdtsc
  sub eax, [subtime]
  mov [subtime], eax

  cpuid
  rdtsc
  mov [subtime], eax
  cpuid
  rdtsc
  sub eax, [subtime]
  mov [subtime], eax

  cpuid
```

```
    rdtsc
    mov [subtime], eax
    cpuid
    rdtsc
    sub eax, [subtime]
    mov [subtime], eax

    ; Only the last value of subtime is kept
    ; subtime should now represent the overhead cost of the
    ; MOV and CPUID instructions

; ADD
    fld qword [x]
    fld qword [y]
    cpuid
    rdtsc
    mov [t_add], eax
    fadd
    cpuid
    rdtsc
    sub eax, [t_add]
    mov [t_add], eax

; SUB
    fld qword [x]
    fld qword [y]
    cpuid
    rdtsc
    mov [t_sub], eax
    fsub
    cpuid
    rdtsc
    sub eax, [t_sub]
    mov [t_sub], eax

; MUL
    fld qword [x]
    fld qword [y]
    cpuid
```

```
  rdtsc
  mov [t_mul], eax
  fmul
  cpuid
  rdtsc
  sub eax, [t_mul]
  mov [t_mul], eax

; DIV
  fld qword [x]
  fld qword [y]
  cpuid
  rdtsc
  mov [t_div], eax
  fdiv
  cpuid
  rdtsc
  sub eax, [t_div]
  mov [t_div], eax

; Print results

  push  dword [t_div]
  push  dword [t_mul]
  push  dword [t_sub]
  push  dword [t_add]
  push  dword [subtime]
  push  fmt                  ; Address of format string
  call  printf               ; Call C function
  add   esp, 24              ; Pop stack 7*4 bytes

; Exit
  mov   eax, 0               ; Exit code, 0=normal
  ret                        ; Main returns to operating system
; End of the code
```

```
fulmanp@fulmanp-k2:~/assembler$ nasm -f elf rdtsc_ex_01.asm -o rdtsc_ex_01.o
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 rdtsc_ex_01.o -o rdtsc_ex_01
```

```
fulmanp@fulmanp-k2:~/assembler$ ./rdtsc_ex_01
subtime=29133, add=28849 sub=29592 mul=29862 div=29255

...

subtime=111618, add=96714 sub=95675 mul=38502 div=29592
subtime=8788, add=6277 sub=6372 mul=124983 div=102708
subtime=7439, add=5697 sub=5724 mul=6561 div=6750
subtime=10058, add=7776 sub=8154 mul=8667 div=8316
subtime=7533, add=5845 sub=5845 mul=5468 div=5697
subtime=13217, add=7722 sub=7749 mul=7776 div=10004
subtime=9963, add=5845 sub=5926 mul=8316 div=5724
```

In both cases the results are difficult to interpretation. I will appreciate any help in this field. This explanation seems to be reliable [10]: *The Time Stamp Counter has, until recently, been an excellent high-resolution, low-overhead way of getting CPU timing information. With the advent of multi-core/hyper-threaded CPUs, systems with multiple CPUs, and hibernating operating systems, the TSC cannot be relied on to provide accurate results — unless great care is taken to correct the possible flaws: rate of tick and whether all cores (processors) have identical values in their time-keeping registers. There is no promise that the timestamp counters of multiple CPUs on a single motherboard will be synchronized. In such cases, programmers can only get reliable results by locking their code to a single CPU. Even then, the CPU speed may change due to power-saving measures taken by the OS or BIOS, or the system may be hibernated and later resumed (resetting the TSC). In those latter cases, to stay relevant, the counter must be recalibrated periodically (according to the time resolution the application requires).*

On an older machine (Asus Eee PC 900HD) we have reproducible results for both integerers

```
fulmanp@fulmanp-eee-900hd:~/assembler$ nasm -f elf rdtsc_ex_02.asm -o rdtsc_ex_02.o
fulmanp@fulmanp-eee-900hd:~/assembler$ gcc rdtsc_ex_02.o -o rdtsc_ex_02
fulmanp@fulmanp-eee-900hd:~/assembler$ ./rdtsc_ex_02
subtime=214, add=213 sub=213 mul=214 div=166

...
```

```
subtime=214, add=213 sub=213 mul=214 div=166
```

and floating point numbers

```
fulmanp@fulmanp-eee-900hd:~/assembler$ nasm -f elf rdtsc_ex_01.asm -o rdtsc_ex_01.o
fulmanp@fulmanp-eee-900hd:~/assembler$ gcc rdtsc_ex_01.o -o rdtsc_ex_01
fulmanp@fulmanp-eee-900hd:~/assembler$ ./rdtsc_ex_01
subtime=214, add=213 sub=213 mul=215 div=247

...

subtime=214, add=213 sub=213 mul=215 div=247
```

11.2.2 Excercise

Use RDTSC instruction to compare dot product programs from previous sections.

Solution

ROZDZIAŁ 1

Inline assembler

Every time we want to get code like

../programs/inline/idea_01.c

```
// Some C
// language code

int foo_g;
int fooFunction(int foo, int bar) {
    ...
    // Assembler part inserted here
    ...
}

// Some C
// language code
```

we have to ralize of the three fundamental problems

1. How to "insert" assembler code to high level language.
2. How to pass variables to an assembler or the same in other words: how to enable that low level language – assembler – access variables created by high levele languages – C/C++. This problem could be divided into two subproblems according to the type of variable
   - global variable,

- local variable.

3. How to return something from low level to high level language.

12.1 First fundamental problem

Assembler part in C high level language begins keyword `asm`* followed by left bracket (and end by right bracket)

../programs/inline/idea_02.c

```
// C code
asm ( <assembler routine> );
// C code

<assembler routne> ::= {"␣assembler␣instrction␣"}*
```

This basic form could be replaced by more sophisticated (extended)

Listing 12.1: ../programs/inline/idea_03.c

```
// C code
asm ( <assembler routine> : output : input : modify);
// C code
```

where the data that will be used as input, output for the asm are specified as well as which registers or memory will be modified. No particular input/output/modify field is compulsory. Regardless of the form, every single assembler instruction have to be followed by new line sequence
`n`. Add example for code without new-line char You can also use the keyword `volatile`† after `asm` which prevent an assembler instruction from being deleted, moved significantly, or combined.

../programs/inline/idea_03_02.c

```
// C code
asm volatile ( <assembler routine> : output : input : modify);
// C code
```

*Or `__asm__` in case of conflict with `asm`.
†Or `__volatile__` in case of conflict with `volatile`.

12.2 Second fundamental problem

12.2.1 Global variables

Basic example of inline assembler could be as follow

../programs/inline/example_01.c

```
#include <stdio.h>

int foo = 0;

void incFoo() {

  asm (
    "mov $foo,%rax\n"
    "add $1,(%rax)\n"
  );
}


int main() {

    incFoo();
    incFoo();
    incFoo();
    printf("Variable \"foo\" after three calls: %d\n", foo);

    return 0;
}
```

compiled and run in usual way

```
fulmanp@fulmanp-k2:~/assembler$ gcc example_01.c -o example_01
fulmanp@fulmanp-k2:~/assembler$ ./example_01
Variable "foo" after three calls: 3
```

This code shows how to get an access to global high level language's variable from inline assembler – simply use the name of this variable in your assembler code. Other thing is assembler syntax – as you can notice, AT&T syntax is used[‡]. For global variables we can set register constraints on variable

[‡]AT&T syntax is default but switch to Intel syntax is also possible.

declaration or simply speaking tie variables to certain hardware registers. This is done at the variable declaration. The following example ties the variable `foo` to register RBX throughout the life of the program

../programs/inline/idea_04.c

```
int register foo asm("rax")=0;
```

../programs/inline/example_02.c

```
#include <stdio.h>

register int foo asm("ebx");

void incFoo() {

   asm (
     "mov␣%ebx,%eax\n"
     "add␣$1,(%eax)\n"
      "mov␣%eax,%ebx\n"
   );
}


int main() {
   foo = 0;
  incFoo();
  incFoo();
  incFoo();
  printf("Variable␣\"foo\"␣after␣three␣calls:␣%d\n", foo);

  return 0;
}
```

When the variable type is not matched with the type of target hardware register, you will receive a compilation error notice. Need example for this. After a variable is tied to a specific register, it is not possible to use another register to hold the same variable. Need example for this.

12.2.2 Local variables

The most intuitive would be to use global variables approach.

../programs/inline/example_07.c

```c
#include <stdio.h>



void incFoo() {
   int foo = 0;

   asm (
     "mov␣$foo,%rax\n"
     "add␣$1,(%rax)\n"
   );
}


int main() {

    incFoo();
    incFoo();
    incFoo();
    printf("Variable␣\"foo\"␣after␣three␣calls:␣%d\n", foo);

    return 0;
}
```

Unfortunately this is not a corret solution

```
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 example_07.c -o example_07
example_07.c: In function ‘main’:
example_07.c:20:56: error: ‘foo’ undeclared (first use in this function)
example_07.c:20:56: note: each undeclared identifier is reported only once for each funct
```

Access to local variables uses extended form of inline assembler (see listing 12.1). The output and input fields must consist of an operand constraint string followed by a C expression enclosed in parentheses. The output operand constraints must be preceded by an = which indicates that it is an output. There may be multiple outputs, inputs, and modified registers. Each "entry" should be separated by commas (,) and there should be no more than 10 entries total. The operand constraint string may either contain the full register name, or an abbreviation.

Letter	Meaning
a	%rax / %eax
b	%rbx / %ebx
c	%rcx / %ecx
d	%rdx / %edx
S	%rsi / %esi
D	%rdi / %edi
I	constant
m	memory
q or r	Allows GCC to assign (select) register.
g	Variable is located in memory or register.
A	long long variable (64bit) is loaded to EAX:EDX.
0, 1,. . . ,9	Reuse previously ”binded” variable.

Tabela 12.1: The operand constraint.

Let's study the first example.

Listing 12.2: ../programs/inline/example_04.c

```
#include <stdio.h>

void foo() {
   int bar;

   printf("Value␣before␣assembler␣section:␣bar=%d\n",bar);
   asm (
      "movl␣$1,%0"
      :               // output
      : "r" (bar)     // input
                      // modify
   );
   printf("Value␣after␣assembler␣section:␣bar=%d\n",bar);

}

int main() {

   foo();

   return 0;
}
```

Notice that

- Variable bar is a type of input variable and is binded to register selected by GCC.
- Notation %0 is used to refer to the first variable defined in assembler part – in this case bar is the first variable.
- The result

  ```
  fulmanp@fulmanp-k2:~/assembler$ ./example_04
  Value before assembler section: bar=0
  Value after assembler section: bar=0
  ```

  is not exactly what we wanted to get; the variable bar is defined as input and that's why cannot be changed.

Next example

Listing 12.3: ../programs/inline/example_05.c

```
#include <stdio.h>

void foo() {
  int bar0, bar1;

  printf("Value␣before␣assembler␣section:␣bar0=%d,␣bar1=%d\n",bar0, bar1);
  asm (
    "movl␣$1,%1\n"
    "movl␣%1,%0\n"
    : "=r" (bar0) // output
    : "r" (bar1)  // input
                  // modify
  );
  printf("Value␣after␣assembler␣section:␣bar0=%d,␣bar1=%d\n",bar0, bar1);
}

int main() {

  foo();
```

```
   return 0;
}
```

and result of running it

```
fulmanp@fulmanp-k2:~/assembler$ gcc -m32 example_05.c
fulmanp@fulmanp-k2:~/assembler$ ./a.out
Value before assembler section: bar0=-601972, bar1=-144661459
Value after assembler section: bar0=1, bar1=-144661459
```

shows some aspects of input and output type variables. As we can see, variable declared in input section could be used in assembly code: sequence

```
movl $1, %1
movl %1, %0
```

copy value 1 to GCC selected register number 1 (%1) which represents variable `bar1` and then copy value from register %1 to GCC selected register number 0 which represents variable `bar0`. Final result is correct, so we conclude that intermediate use of register %1 was correct — the value 1 was transferred to `bar0` via `bar1` — but we cannot see any changes in `bar1` because it wasn't declared as "viewable" (output) type.

Now we can present fixed code from listing 12.2 and 12.3.

Listing 12.4: ../programs/inline/example_04_fix.c

```
#include <stdio.h>

void foo() {
   int bar;

   printf("Value␣before␣assembler␣section:␣bar=%d\n",bar);
   asm (
      "movl␣$1,%0"
      : "=r" (bar)  // output
      :             // input
      :             // modify
   );
   printf("Value␣after␣assembler␣section:␣bar=%d\n",bar);
}

int main() {

   foo();

   return 0;
}
```

```
fulmanp@fulmanp-k2:~/assembler$ ./example_04_fix
Value before assembler section: bar=0
Value after assembler section: bar=1
```

../programs/inline/example_05_fix.c

```
#include <stdio.h>

void foo() {
   int bar0, bar1;

   printf("Value before assembler section: bar0=%d, bar1=%d\n", bar0, bar1);
   asm (
      "movl $1,%1\n"
      "movl %1,%0\n"
      : "=r" (bar0), "=r" (bar1) // output
      :          // input
      :          // modify
   );
   printf("Value after assembler section: bar0=%d, bar1=%d\n", bar0, bar1);
}

int main() {

   foo();

   return 0;
}
```

```
fulmanp@fulmanp-k2:~/assembler$ ./example_05_fix
Value before assembler section: bar0=-3987300, bar1=-144612307
Value after assembler section: bar0=1, bar1=1
```

Next example shows two more things.

../programs/inline/example_06.c

```
#include <stdio.h>

int weightedSum(foo1, weight1, foo2, weight2) {
  int sum;
  asm(
     "mull  %%ebx\n"
     "movl  %%eax, %%ecx\n"
     "movl  %3, %%eax\n"
     "mull  %4\n"
     "addl  %%ecx, %%eax\n"
     "movl  %%eax, %0\n"
     :"=d" (sum)                              // output
     // input: eax:=foo1, ebx=weight1, ?=foo2, ?=weight2
     :"a" (foo1), "b" (weight1), "r" (foo2), "r" (weight2)
     :                                        // modify
  );
  return sum;
}

int main() {
   int  res = weightedSum(3,5,7,11);
   printf("Result = %d\n",res);
   return 0;
}
```

You may have noticed that registers are now prefixed with %% rather than %. This is necessary when using the output/input/modify fields because register aliases (numbers from %0 to %9) based on the extra fields can also be used. Intention of this code should be clear: we want to calculate weighted sum of two variables. Unfortunately the code doesn't work

```
fulmanp@fulmanp-k2:~/assembler$ ./example_06
Result = 180
```

Clobber list

Some instructions clobber some hardware registers. We have to list those registers in the clobber-list, i.e. the modify field after the third : in the assembler code. This is to inform GCC that we will use and modify them ourselves. So GCC will not assume that the values it loads into these registers will be valid. We shoudn't list the input and output registers in this list. Because, GCC knows that assembler uses them (because they are specified explicitly as constraints). If the instructions use any other registers, implicitly or explicitly (and the registers are not present either in input or in the output constraint list), then those registers have to be specified in the clobbered list.

In our code clobbered register is ECX. We use it, for example in line

```
movl  %%eax, %%ecx
```

That is why we have to inform GCC about that in modify field – see fixed version of this code.

../programs/inline/example_06_fix.c

```c
#include <stdio.h>

int weightedSum(foo1, weight1, foo2, weight2) {
  int sum;
  asm(
     "mull␣␣%%ebx\n"
     "movl␣␣%%eax,␣%%ecx\n"
     "movl␣␣%3,␣%%eax\n"
     "mull␣␣%4\n"
     "addl␣␣%%ecx,␣%%eax\n"
     "movl␣␣%%eax,␣%0\n"
     :"=d" (sum)                              // output
     :"a" (foo1), "b" (weight1), "r" (foo2), "r" (weight2)  // input: eax:=foo1, ebx=weight1
     : "ecx"                                  // modify
  );
  return sum;
}

int main() {
   int  res = weightedSum(3,5,7,11);
   printf("Result␣=␣%d\n",res);
   return 0;
}
```

```
fulmanp@fulmanp-k2:~/assembler$ ./example_06_fix
Result = 92
```

12.3 Third fundamental problem

This problem was solved in previous sections. As we have seen, we can return value using variables binded to registers declared in *output* section of inline code.

ROZDZIAŁ 1

Introduction

In the beginning, Intel created the 8086
and its first 16-bit microprocessor.
And Intel said, Let there be x86: and there
was x86.
And Intel saw the x86, that it was good.

`http://www.maximumpc.com/article/features/cpu_retrospective_the_life_and_times_x86`

13.1 Assembly language

Because this book is about assembly languages, let's try to understand what an assebly language is. Simply speaking

Definition 13.1. *an **assembly language** is a low-level programming language for a computer, microcontroller, or other programmable device, in which each statement corresponds to a single machine code instruction.*

According to this definition it is not surprising, that each assembly language is specific to a particular computer architecture which stays in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an **assembler**; the conversion process is referred to as **assembly**, or **assembling** the code. There is usually a one-to-one correspondence between simple assembly statements and machine language instructions. In everyday language an assembly languages is very often refered as assembler, but it's good to distinguish between these concepts.

The most natural language for every processor is a sequence or stream of bits. For example, the instruction

```
10110000 01100001
```

tells an x86/IA-32 processor to move an immediate 8-bit value into a register. The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the AL register is 000, so the following machine code loads the AL register with the data 01100001.

Although this type of language is most natural for computers, it is completelu useless for human. This binary computer code can be made more human-readable by expressing it in hexadecimal as follows

```
B0 61
```

Here, B0 means *Move a copy of the following value into AL*, and 61 is a hexadecimal representation of the value 01100001, which is 97 in decimal. A little bit beter but still far from perfection, mainly because one number expressed many things like typ of operation (copy, 5 bits) and location (AL register, 3 bits) in above example. The key idea behind assembly language is to

- separate all parts of instruction to make them independent from other,
- replace some binary sequences, like 10110, by something which is easier to remember or which help human to figure out what are they represents.

Continuing our example, Intel assembly language provides the mnemonic MOV, which is an abbreviation of move, for instructions such as this, so the machine code above can be written as follows in assembly language

```
MOV AL, 61h        ; Load AL with 97 decimal (61 hex)
```

and this is much easier to read and to remember, even without an explanatory comment after the semicolon. What is more important, in many cases the same mnemonic such as MOV may be used for a family of related instructions even thought that are represented by different binary sequences. For example the Intel uses opcode 10110000 (B0) to copy an 8-bit value into the AL register, while 10110001 (B1) to move it into CL.

```
MOV AL, 1h        ; Load AL with immediate value 1
MOV CL, 2h        ; Load CL with immediate value 2
```

In each case, the MOV mnemonic is translated directly into an opcode by an assembler, and the programmer does not have to know or remember which.

Each computer architecture has its own machine language. Computers differ in the number and type of operations they support, in the different sizes and numbers of registers, and in the representations of data in storage. While most general-purpose computers are able to carry out essentially the same functionality, the ways they do so differ; the corresponding assembly languages reflect these differences.

13.2 Pre-x86 age – historical background

- **1947**: The transistor is invented at Bell Labs.
- **1965**: Gordon Moore at Fairchild Semiconductor observes that the number of transistors on a semiconductor chip doubles every year*. For microprocessors, it will double about every two years for more than three decades.
- **1968**: Gordon Moore, Robert Noyce and Andy Grove found Intel Corp. to make the business of "INTegrated ELectronics."
- **1969**: Intel announces its first product, the world's first metal oxide semiconductor (MOS) static RAM, the 1101. It signals the end of magnetic core memory.
- **1971**: Intel launches the world's first microprocessor, the 4-bit 4004, designed by Federico Faggin. The 2,000-transistor chip is made for a Japanese calculator, but Intel calls it "a micro-programmable computer on a chip."
- **1972**: Intel announces the 8-bit 8008 processor. Teenagers Bill Gates and Paul Allen try to develop a programming language for the chip, but it is not powerful enough.
- **1974**: Intel introduces the 8-bit 8080 processor, with 4,500 transistors and 10 times the performance of its predecessor.

*ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf

- **1975**: The 8080 chip finds its first PC application in the Altair 8800, launching the PC revolution. Gates and Allen succeed in developing the Altair Basic language, which will later become Microsoft Basic, for the 8080.
- **1976**: The x86 architecture suffers a setback when Steve Jobs and Steve Wozniak introduce the Apple II computer using the 8-bit 6502 processor from MOS Technology. PC maker Commodore also uses the Intel competitor's chip.
- **1978**: Intel introduces the 16-bit 8086 microprocessor – a new age begins.

13.2.1 Intel 4004

The Japanese company Busicom had designed special purpose chipset for use in their Busicom 141-PF calculator and commissioned Intel to develop it for production. However, Intel determined it was too complex and would use non-standard packaging and so it was proposed that a new design produced with standard 16-pin DIP packaging and reduced instruction set be developed. This resulted in the 4004, released by Intel Corporation in 1971, which was part of a family of chips, including ROM, DRAM and serial to parallel shift register chips. The Intel 4004 was a 4-bit central processing unit (CPU). It was the second complete CPU on one chip (only preceded by the TMS 1000), and also the first commercially available (sold as a component) microprocessor.

Technical specifications.

- Approximately 2,300 transistors
- Maximum clock speed was 740 kHz
- Instruction cycle time: 10.8 μs (8 clock cycles / instruction cycle)
- Instruction execution time 1 or 2 instruction cycles (10.8 or 21.6 μs), 46300 to 92600 instructions per second
- Separate program and data storage. Contrary to Harvard architecture designs, however, which use separate buses, the 4004, with its need to keep pin count down, used a single multiplexed 4-bit bus for transferring:
  - 12-bit addresses
  - 8-bit instructions

  - 4-bit data words

- Instruction set contained 46 instructions (of which 41 were 8 bits wide and 5 were 16 bits wide)
- Register set contained 16 registers of 4 bits each
- Internal subroutine stack 3 levels deep.

If you want to know more... 13.1 (Harvard architecture)**.** *The term originated from the Harvard Mark I computer, employed* ***entirely separate memory systems*** *to store instructions and data. The CPU fetched the next instruction and loaded or stored data simultaneously and independently. This is in contrast to a Von Neumann architecture computer, in which both instructions and data are stored in the same memory system and must be accessed in turn. The true distinction of a Harvard machine is that instruction and data memory occupy different address spaces. In other words, a memory address does not uniquely identify a storage location (as it does in a Von Neumann machine); you also need to know the memory space (instruction or data) to which the address belongs.*

13.2.2 Intel 8008

Originally known as the 1201, the Intel 8008 chip – early byte-oriented microprocessor introduced in April 1972 – was commissioned by Computer Terminal Corporation (CTC) to implement an instruction set of their design for their Datapoint 2200 programmable terminal. Intel didn't believe there really was a significant market for a general-purpose microcomputer-on-a-chip – John Frassanito recalls that *"Bob Noyce said it was an intriguing idea, and that Intel could do it, but it would be a dumb move. He said that if you have a computer chip, you can only sell one chip per computer, while with memory, you can sell hundreds of chips per computer."*[2]. What's more, if Intel introduced their own processor, they might be seen as a competitor, and their customers might look elsewhere for memory. As the chip was delayed and did not meet CTC's performance goals, the 2200 ended up using CTC's own TTL based CPU instead. An agreement permitted Intel to market the chip to other customers after Seiko expressed an interest in using it for a calculator. Cooperation with CTC explains the reason Intel to this day uses LSB/MSB byte order: because the Type 1 2200 used a serial shift register memory, and that allowed propagating carries from LSB to MSB without requiring the memory recirculate around to the previous byte.

Technical specifications.

- 8-bit CPU with an external 14-bit address bus that could address 16KB of memory. The chip (limited by its 18-pin DIP packaging) had a single 8-bit bus and required a significant amount of external support logic. To verify
- Initial versions of the 8008 could work at clock frequencies up to 0.5 MHz, this was later increased in the 8008-1 to a specified maximum of 0.8 MHz.
- Instructions took between 5 and 11 T-states where each T-state was 2 clock cycles.
- Register-register loads and ALU operations took 5T (20 μs at 0.5 MHz), register-memory 8T (32 μs), while calls and jumps (when taken) took 11 T-states (44 μs).
- The 8008 was a little slower in terms of instructions per second (36,000 to 80,000 at 0.8 MHz) than the 4-bit Intel 4004 and Intel 4040,[6] but the fact that the 8008 processed data eight bits at a time and could access significantly more RAM still gave it a significant speed advantage in most applications.
- The 8008 had 3,500 transistors.

13.2.3 Intel 8080

The Intel 8080 was the second 8-bit microprocessor designed and manufactured by Intel and was released in April 1974. It was an extended and enhanced variant of the earlier 8008 design, *with assembly-language compatibility although without binary compatibility*[†]. It used the same basic instruction set as the 8008 and added some handy 16-bit operations to the instruction set as well. Larger 40-pin DIP packaging allowed to provide a 16-bit address bus and an 8-bit data bus.

Architecture details and technical specifications.

- With 16-bit address bus, the Intel 8080 allowing an access to 64 KiB of memory.
- The processor had seven 8-bit registers (A, B, C, D, E, H, and L) where A was the 8-bit accumulator and the other six could be used as either byte-registers or as three 16-bit register pairs (BC, DE, HL) depending on the particular instruction. Some instructions also enabled HL to be used as a (limited) 16-bit accumulator, and a pseudoregister, M, could be used almost anywhere that any other register could be used and referred to the memory address pointed to

[†]This sentence is very important and emphasizes differences between assembler (assembly-language) and binary code – the same assembler may result in different binary code.

by HL. It also had a 16-bit stack pointer to memory (replacing the 8008's internal stack), and a 16-bit program counter.

- The processor maintains internal flag bits which show results of artithmetic and logical functions. The flags are:
  - **sign** – set 1 if result is negative,
  - **zero** – set if the accumulator register is zero,
  - **parity** – set 1 if the number of 1 bits in the accumulator is even,
  - **carry** – set if the last add operation resulted in a carry, or if the last subtraction operation did not require a borrow,
  - **auxiliary carry** – used for binary-coded decimal arithmetic.

  The purpose of flag bits is that it simplify some operation – conditional branch instructions could test the various flag status bits (set after last operation) and based on it decide to make or not a jump. To better understand this please read section 1.5.

- All the Intel 8080's instructions were encoded in a single byte (including register-numbers, but excluding immediate data), for simplicity. Some of them were followed by one or two bytes of data, which could be an immediate operand, a memory address, or a port number. Like larger processors, it had automatic CALL and RET instructions for multi-level procedure calls and returns (which could even be conditionally executed, like jumps) and instructions to save and restore any 16-bit register-pair on the machine stack. There were also eight one-byte call instructions (RST) for subroutines located at the fixed addresses 00h, 08h, 10h,. . . ,38h. These were intended to be supplied by external hardware in order to invoke a corresponding interrupt-service routine, but were also often employed as fast system calls.

- Although the 8080 was generally an 8-bit processor, it also had limited abilities to perform 16-bit operations. For example any of the three 16-bit register pairs (BC, DE, HL) or SP could be loaded with an immediate 16-bit value (using LXI), incremented or decremented (using INX and DCX), or added to HL (using DAD).

- The Intel 8080 provided a separate stack space. One of the bits in the processor state word indicates that the processor is accessing data from the stack. Using this signal, it was possible to implement a separate stack memory space. However, this feature was seldom used.
- The 8080 was manufactured in a silicon gate process using a minimum feature size of 6 μm.
- Approximately 6,000 transistors were used and the die size was approximately 20 mm^2.
- The initial specified clock frequency limit was 2 MHz with common instructions having execution times of 4, 5, 7, 10 or 11 cycles.

Influence on industry

Until the 8080 was introduced, computer systems were usually created by computer manufacturers as the entire computer, including processor, terminals, and system software such as compilers and operating system and all other stuff. The 8080 has sometimes been labeled "*the first truly usable microprocessor*", although earlier microprocessors were used for calculators and other applications. The 8080 was actually designed for just about **any application**.

The 8080 and 8085 gave rise to the 8086, which was designed as a source compatible (although not binary compatible) extension of the 8085. This design, in turn, later spawned the x86 family of chips, the basis for most CPUs in use today. Many of the 8080's core machine instructions and concepts, for example, registers named A, B, C and D, as well as many of the flags used to control conditional jumps, are still in use in the widespread x86 platform. 8080 Assembler code can still be directly translated into x86 instructions; all of its core elements are still present.

13.2.4 An early x86 age – accidental birth of a standard

- **1975**: Intel sarted project iAPX 432.
- **1978**: Intel introduces the 16-bit 8086 microprocessor.
- **1979**: Intel introduces a lower-cost version of the 8086, the 8088, with an 8-bit bus.
- **1980**: Intel introduces the 8087 math co-processor.
- **1981**: IBM picks the Intel 8088 to power its PC.
- **1982**: IBM signs Advanced Micro Devices as second source to Intel for 8086 and 8088 microprocessors.

In 1975 Intel started project iAPX 432 (short for ***i**ntel **A**dvanced **P**rocessor archite**c**ture*[‡]. This project, if successfully implemented, would became a point in computer history when completely new quality arise.

The preceding 8-bit microprocessors' instruction sets were too primitive to support compiled programs and large software systems. Intel now aimed to build a sophisticated complete system in a few LSI chips, that was functionally equal to or better than the best 32-bit minicomputers and mainframes requiring entire cabinets of older chips. This system would support multiprocessors, modular expansion, fault tolerance, advanced operating systems, advanced programming languages, very large applications, ultra reliability, and ultra security. Many advanced multitasking and memory management features were implemented in hardware, leading to the design being referred to as a Micromainframe. Because the 432 had no software compatibility with existing software the architects had total freedom to do a novel design from scratch, using whatever techniques they guessed would be best for large-scale systems and software. They applied fashionable computer science concepts from universities, particularly capability machines, object-oriented programming, high-level CISC machines, Ada, and densely encoded instructions. This ambitious mix of novel features made the chip larger and more complex. The chip's complexity limited the clock speed and lengthened the design schedule. Not far from the beginning of the project it became clear that it would take several years and many engineers to design all this. Meanwhile, Intel urgently needed a **simpler interim product to meet the immediate competition** from Motorola, Zilog, and National Semiconductor. So Intel began a rushed project to design the **8086 as a low-risk incremental evolution from the 8080**, using a separate design team. The mass-market 8086 shipped i8. As it turned out, despite the fact of substitutional nature of 8086, it was good enough to begin the IBM PC age. When introduced (1981), the 432 ran many times slower than contemporary conventional microprocessor designs such as the Motorola 68010 and Intel 80286. Slow, uncompatible with existing software and technicaly very complicated – this is not a recipe for success.

13.2.5 Mid-x86 age – conquest of the market

- 1982: Intel introduces the 16-bit 80286 processor with 134,000 transistors.
- 1984: IBM develops its second-generation PC, the 80286-based PC-AT. The PC-AT running MS-DOS will become the de facto PC standard for almost 10 years.

[‡]This project was initially named the 8800, as next step beyond the existing Intel 8008 and 8080 microprocessors.

- 1985: Intel exits the dynamic RAM business to focus on microprocessors, and it brings out the 80386 processor, a 32-bit chip with 275,000 transistors and the ability to run multiple programs at once.
- 1986: Compaq Computer leapfrogs IBM with the introduction of an 80386-based PC.
- 1987: VIA Technologies is founded in Fremont, Calif., to sell x86 core logic chip sets.
- 1989: The 80486 is launched, with 1.2 million transistors and a built-in math co-processor. Intel predicts the development of multicore processor chips some time after 2000.
- Late 1980s: The complex instruction set computing (CISC) architecture of the x86 comes under fire from the rival reduced instruction set computing (RISC) architectures of the Sun Sparc, the IBM/Apple/Motorola PowerPC and the MIPS processors. Intel responds with its own RISC processor, the i860.
- 1990: Compaq introduces the industry's first PC servers, running the 80486.
- 1993: The 3.1 million transistor, 66-MHz Pentium processor with superscalar technology is introduced.
- 1994: AMD and Compaq form an alliance to power Compaq computers with Am486 microprocessors.
- 1995: The Pentium Pro, a RISC slayer, debuts with radical new features that allow instructions to be anticipated and executed out of order. That, plus an extremely fast on-chip cache and dual independent buses, enable big performance gains in some applications.
- 1997: Intel launches its 64-bit Epic processor technology. It also introduces the MMX Pentium for digital signal processor applications, including graphics, audio and voice processing.
- 1998: Intel introduces the low-end Celeron processor.
- 1999: VIA acquires Cyrix Corp. and Centaur Technology, makers of x86 processors and x87 co-processors.
- 2000: The Pentium 4 debuts with 42 million transistors.

13.2.6 Late-x86 age – stone age devices

- 2003: AMD introduces the x86-64, a 64-bit superset of the x86 instruction set.
- 2004: AMD demonstrates an x86 dual-core processor chip.
- 2005: Intel ships its first dual-core processor chip.
- 2005: Apple announces it will transition its Macintosh computers from PowerPCs made by Freescale (formerly Motorola) and IBM to Intel's x86 family of processors.
- 2005: AMD files antitrust litigation charging that Intel abuses "monopoly" to exclude and limit competition. (The case is still pending in 2008.)
- 2006: Dell Inc. announces it will offer AMD processor-based systems.

13.3 An overview of the x86 architecture

13.3.1 Basic properties of the architecture

tutu

13.3.2 Operating modes

Real mode

Real mode is an operating mode of 8086 and all later x86-compatible CPUs. Real mode is characterized by

- a **20 bit segmented memory address space** (only 1 MiB of memory can be addressed),
- direct software access to BIOS routines and peripheral hardware,
- lack of memory protection or multitasking at the hardware level.

All x86 CPUs compatible processors start up in real mode at power-on.

Protected mode

The Intel 80286, in addition to real mode, introduced to support protected mode, where

- addressable physical memory was expanded to 16 MB and addressable virtual memory to 1 GB,
- provide protected memory, which prevents programs from corrupting one another.

The Intel 80386 introduced to support in protected mode for paging – a mechanism making it possible to use paged virtual memory. This extension allows to develop many modern opeating systems like Linux or Windows NT and in consequence the 386 architecture became the basis of all further development in the x86 series.

Upon power-on, the processor initializes in real mode, and then begins executing instructions. Operating system boot code may place the processor into the protected mode to enable more advanced features. The instruction set in protected mode is backward compatible with the one used in real mode.

Virtual 8086 mode

The virtual 8086 mode is a sub-mode of operation in 32-bit protected mode. This is a hybrid operating mode that allows real mode programs and operating systems to run under the control of a protected mode supervisor operating system. This allows to running both protected mode programs and real mode programs simultaneously. This mode is exclusively available for the 32-bit version of protected mode; virtual 8086 mode does not exist in the 16-bit version of protected mode, or in long mode.

Long mode

The 32-bit address space of the x86 architecture was limiting its performance in applications requiring large data sets. When designed a 32-bit address space would allow the processor to directly address, unimaginably large in those days, data – 4 GiB, but relativeli fast this size was surpassed by applications such as video processing and database engines. Using 64-bit addresses, one can directly address 16 EiB (or 16 billion GiB) of data, although most 64-bit architectures don't support access to the full 64-bit address space (AMD64, for example, supports only 48 bits, split into 4 paging levels, from a 64-bit address).

AMD developed the 64-bit extension of the 32-bit x86 architecture that is currently used in x86 processors, initially calling it x86-64, later renaming it AMD64. The Opteron, Athlon 64, Turion 64, and later Sempron families of processors use this architecture. The success of the AMD64 line of processors coupled with the lukewarm reception of the IA-64 architecture forced Intel to release its own implementation of the AMD64 instruction set. This was the first time that a major extension of the x86 architecture was initiated and originated by a manufacturer other than Intel. It was also the first time that Intel accepted technology of this nature from an outside source.

Long mode is mostly an extension of the 32-bit instruction set, but unlike the 16 to 32-bit transition, many instructions were dropped in the 64-bit mode. This does not affect actual binary backward compatibility (which would execute legacy code in other modes that retain support for those instructions), but it changes the way assembler and compilers for new code have to work.

Intel branded its implementation of AMD64 as EM64T, and later re-branded it Intel 64. In its literature and product version names, Microsoft and Sun refer to AMD64/Intel 64 collectively as x64 in the Windows and Solaris operating systems respectively. Linux distributions refer to it either as "x86-64", its variant "x86_64", or "amd64". BSD systems use "amd64" while Mac OS X uses "x86_64".

ROZDZIAŁ 1

Registers

Computer Science is no more about computers than astronomy is about telescopes.
Edsger W. Dijkstra

The computer was born to solve problems that did not exist before.
Bill Gates

14.1 General information

A **processor register** is a small amount of storage available as part of a CPU or other digital processor. Registers are typically at the top of the memory hierarchy, and provide the fastest way to access data[*].

If you want to know more. . . 14.1 (Out-of-order execution)**.** *In computer engineering, **out-of-order execution** (OoOE or OOE) is a paradigm to make use of instruction cycles that would otherwise be wasted by a certain type of costly delay. In this paradigm, a processor executes instructions in an order governed by the* availability *of input data, rather than by their original*

[*]The term normally refers only to the group of registers that are directly encoded as part of an instruction, as defined by the instruction set. However, modern high performance CPUs often have duplicates of these "architectural registers" in order to improve performance via **register renaming**, allowing parallel and **speculative execution**.

order *in a program. In doing so, the processor can avoid being idle while data is retrieved for the next instruction in a program, processing instead the next instructions which are able to run immediately. For instance, a processor may be able to execute hundreds of instructions while a single load from main memory is in progress. Shorter instructions executed while the load is outstanding will finish first, thus the instructions are finishing out of the original program order.*

Ta cecha powoduje jednak, że mikroprocesor musi pamiętać rzeczywistą kolejność (zwykle posiada wiele kopii rejestrów, niewidocznych dla programisty) i uaktualniać stan w oryginalnym porządku, ale także anulować (wycofywać) zmiany, w przypadku gdy wystąpił jakiś błąd podczas wykonywania wcześniejszej instrukcji. Ilustracja dla hipotetycznego mikroprocesora z dwiema jednostkami wykonawczymi:

```
1. a = b + 1
2. c = a + 2
3. d = e + 1
4. f = d + 2
```

Instrukcja nr 2 nie może wykonać się przed pierwszą, bowiem jej argument zależy od wyniku instrukcji 1., podobnie instrukcja 4. zależy od 3. Bez zmiany kolejności procesor wykonałby szeregowo 4 instrukcje w założonym porządku, wykorzystując jednak tylko jedną jednostkę wykonawczą:

```
czas . . . . . .
     1
       2
         3
           4
```

Jednak można wykonać równolegle niezależne od siebie instrukcje 1. i 3., następnie również równolegle instrukcje 2. i 4. — w ten sposób wykorzystane zostaną obie jednostki wykonawcze, także czas wykonywania będzie 2 razy mniejszy:

```
czas . . . .
     1
     3
       2
```

```
    4
```

If you want to know more... 14.2 (Register renaming)**.** *In computer architecture, register renaming refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations. Consider this piece of code running on an out-of-order CPU*

```
1. a = b
2. a = a + 1
3. b = a
4. a = c
5. a = a + 2
6. c = a
```

Instructions 1, 2, and 3 are independent of instructions 4, 5, and 6, but the processor cannot finish 4 until 3 is done, because 3 would then write the wrong value. Fortunately, we can eliminate this restriction by changing the names *of some of the registers making this code possible to be executed as out-of-order*

```
1. a = b
2. a = a + 1
3. b = a
4. d = c
5. d = d + 2
6. c = d
```

or the same but more clearly

```
1. a = b        4. d = c
2. a = a + 1    5. d = d + 2
3. b = a        6. c = d
```

Now instructions 1, 2, and 3 can be executed in parallel with instructions 4, 5, and 6. When possible, the compiler would detect the distinct instructions and try to assign them to a different register. However, there is a finite number of register names that can be used in the assembly *code. This is why many high performance CPUs have more physical registers than may be named directly in the instruction set, so they rename registers in hardware to achieve additional parallelism.*

If you want to know more... 14.3 (Speculative execution)**.** *Speculative execution in computer systems is doing work, the result of which may not be needed. This performance optimization technique is very often used in pipelined processors and other systems. The main idea is to do work before it is known whether that work will be needed at all, so as to prevent a delay that would have to be incurred by doing the work after it is known whether it is needed. If it turns out the work wasn't needed after all, the results are simply ignored. The target is to provide more concurrency if extra resources are available. For instance, modern pipelined microprocessors use speculative execution to reduce the cost of conditional branch instructions.*

14.2 Categories of registers

The most coarse division of registers based on the number of bits they can hold. We have, for example, a set of an "8-bit registers" or a "32-bit registers". More precise classification based on registrs' content or instructions that operate on them[†].

- **User-accessible registers** – registers to which a user have an access to freely read and write. The most common division of user-accessible registers is into data registers and address registers.
    - **Data registers** can hold various kind of data: numeric such as integer and floating-point, characters, small bit arrays etc. In some older and low end CPUs, a special data register, known as the accumulator, is used implicitly for many operations.
    - **Address registers** hold addresses and are used by instructions that indirectly access main memory (sometimes called *primary memory* when we consider the whole hierarchy of computer's memory)[‡].
- **General purpose registers (GPRs)** – can store both data and addresses, i.e., they are combined data/address registers.

[†]Please note that some registers belongs to more than one category.

[‡]Nothe that some processors contain registers that may only be used to hold an address or only to hold numeric values (in some cases used as an index register whose value is added as an offset from some address); others allow registers to hold either kind of quantity.

- **Floating point registers (FPRs)** – in many architectures dedicated registers to store floating point numbers.
- **Special purpose registers (SPRs)** – hold program state; they usually include the **program counter** (aka **instruction pointer**) and **status register** (aka **processor status word (PSW)**). Processor status word is a register used as a vector of bits representing Boolean values to store and control the results of operations and the state of the processor. Sometimes the **stack pointer** is also included in this group. The very special kind of this type of registers is an **instruction register (IR)**. An instruction register stores the instruction currently being executed or decoded. In simple processors each instruction to be executed is loaded into the instruction register which holds it while it is decoded, prepared and finally executed, which can take several steps. Some of the complicated processors use a pipeline of instruction registers where each stage of the pipeline does part of the decoding, preparation or execution and then passes it to the next stage for its step (see *Instruction pipeline* notes below).
- **Control and status registers** – there are three types: **program counter**, **instruction registers** and **processor status word**.
- **Vector registers** hold data for vector processing done by SIMD instructions (Single Instruction, Multiple Data).
- Embedded microprocessors can also have registers corresponding to specialized hardware elements.

If you want to know more... 14.4 (Instruction pipeline). *An **instruction pipeline** is a technique used to increase the number of instructions that can be executed by CPU in a unit of time (refers as instruction throughput). Note, that **pipelining does not reduce the time to complete an instruction, but increases the number of instructions that can be processed at once.***

In this technique each instruction is split into a sequence of independent steps. Taking into account e.g. the basic five-stage pipeline in a RISC machine the following steps are distinguished

- *Instruction Fetch (IF),*
- *Instruction Decode and register fetch (ID),*
- *Execute (EX),*

- *Memory access (MEM),*
- *Register write back (WB).*

Pipelining let the processor work on as many instructions as there are independent steps. This approach is similar to an assembly line where many vehicles are build at once, rather than waiting until one vehicle has passed through the whole line before admitting the next one. As the goal of the assembly line is to keep each assembler productive at all times, pipelining seeks to use every part of the processor busy with some instruction. Pipelining lets the computer's cycle time be the time of the slowest step, and ideally lets one instruction complete in every cycle.

Pipelining, among many benefits, leads also to problem known as a ***hazard****. It arise because a human programmer writing an assembly language program assumes the sequential-execution model – model when each instruction completes before the next one begins. Unfortunately this assumption is not true on a pipelined processor. Imagine the following two register instructions to a hypothetical RISC processor that has the 5, aforementioned, steps*

```
1. Add R1 to R2.
2. Move R2 to R3.
```

Instruction 1 would be fetched at time t_1 and its execution would be complete at t_5. Instruction 2 would be fetched at t_2 and would be complete at t_6. The first instruction might deposit the incremented number into `R2` *as its fifth step (register write back) at t_5. But the second instruction might get the number from* `R2` *(to move to* `R3`*) in its second step at time t_3. The problem is that the first instruction would not have incremented the value by then. Such a situation where the expected result is problematic is a* hazard. *A human programmer writing in a compiled language might not have these concerns, as the compiler could be designed to generate machine code that avoids hazards.*

14.3 x86 registers

14.3.1 16-bit architecture

The original Intel 8086 and 8088 have fourteen 16-bit registers.

- Four of them (AX, BX, CX, DX) are general-purpose registers (GPRs)[§]. Each can be divided into two parts accessed independently as two separate bytes – for example high byte (or MSB – most significant byte) of AX can be accessed as AH while low byte (or LSB – least significant byte) as AL. Despite the generality of those registers, all of them have "predefined" meaning
  - AX is an accumulator register used in arithmetic operations.
  - BX is a base register used as a pointer to data (located in segment register DS, when in segmented mode).
  - CX is a counter register used in shift/rotate instructions and loops.
  - DX is a data register used in arithmetic operations and I/O operations.
- There are two pointer registers: SP (stack pointer register) which points to the top of the stack and BP (stack base pointer register used to point to the base of the stack.
- Two registers (SI and DI) are for array indexing. SI is a source index register used as a pointer to a source in stream operations. DI is a destination index register used as a pointer to a destination in stream operations.
- Four segment registers (SS, CS, DS and ES) are used to form a memory address.
  - SS – stack sgment – pointer to the stack.
  - CS – code segment – pointer to the code.
  - DS – data segment – pointer to the data.
  - ES – extra segment – pointer to extra data ('E' stands for 'Extra').
- The FLAGS register used as processor status word contains – see table 14.1 and 14.2 for description of the meaning of a bits.
- The instruction pointer (IP) points to the next instruction that will be fetched from memory and then executed (if no branching is done). This register cannot be directly accessed (read or write) by a program.

[§]Although each may have an additional purpose: for example only CX can be used as a counter with the loop instruction.

Bit	Abbreviation	Description	Category
0	CF	Carry flag	Status
1	1	Reserved	
2	PF	Parity flag	Status
3	0	Reserved	
4	AF	Adjust flag	Status
5	0	Reserved	
6	ZF	Zero flag	Status
7	SF	Sign flag	Status
8	TF	Trap flag (single step)	System
9	IF	Interrupt enable flag	Control
10	DF	Direction flag	Control
11	OF	Overflow flag	Status
12-13	IOPL	I/O privilege level (286+ only), always 1 on 8086 and 186	System
14	NT	Nested task flag (286+ only), always 1 on 8086 and 186	System
15	0	Reserved, always 1 on 8086 and 186, always 0 on later models	

Tabela 14.1: Intel x86 FLAGS register.

Flag	Set when...
AF	Carry of Binary Code Decimal (BCD) numbers arithmetic operations.
CF	Set if the last arithmetic operation carried (addition) or borrowed (subtraction) a bit beyond the size of the register. This is then checked when the operation is followed with an add-with-carry or subtract-with-borrow to deal with values too large for just one register to contain.
DF	Stream direction. If set, string operations will decrement their pointer rather than incrementing it, reading memory backwards.
IF	Set if interrupts are enabled.
IOPL	I/O Privilege Level of the current process.
OF	Set if signed arithmetic operations result in a value too large for the register to contain.
NT	Controls chaining of interrupts. Set if the current process is linked to the next process.
PF	Set if the number of set bits in the least significant byte is a multiple of 2.
SF	Set if the result of an operation is negative.
TF	Set if step by step debugging.
ZF	Set if the result of an operation is Zero (0).

Tabela 14.2: Meaning of the Intel x86 FLAGS register.

14.3.2 32-bit architecture

The 80386 extended the set of registers to 32 bits while retaining all of the 16-bit and 8-bit names that were available in 16-bit mode. The new extended registers are denoted by adding an E (for Extended) prefix; thus the core eight 32-bit registers are named EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP. The original 8-bit and 16-bit register names map into the least significant portion of the 32-bit registers. There are two new segment registers

- FS – F segment – pointer to more extra data ('F' comes after 'E' used to denote 16-bit extra segment register ES).
- GS – G segment – pointer to still more extra data ('G' comes after 'F').

What is important, all segment regiters were still 16-bit. The low half of the extenden 32-bit flag register EFLAGS stay unchanged and is identical to FLAGS. New bits are introduced in high half of the flag register – see table 14.3 and 14.4 for description of the meaning of a bits. Above mentioned extension was natural and was not connected with any significant improvements in CPU architecture. Later, 32-bit architecture were upgraded with new functionality significantly improve the performance.

1. With the 80486 a floating-point processing unit (FPU) was added, with eight 80-bit wide registers: ST(0) to ST(7)[¶].
2. With the Pentium MMX, eight 64-bit MMX integer registers were added (MMX0 to MMX7, which share lower bits with the 80-bit-wide FPU stack).
3. With the Pentium III, an eight 128-bit SSE floating point registers (XMM0 to XMM7) were added. There is also a new 32-bit control/status register, MXCSR. Please read chapter 10 for more details.
4. In March 2008 Intel proposed Advanced Vector Extensions (AVX) instruction set which was first supported by Intel with the Sandy Bridge processor shipping in Q1 2011 and later on by AMD with the Bulldozer processor shipping in Q3 2011. AVX provides new features, new instructions and a new coding scheme. With this also a new set of registers were introduced: the width of the SIMD register file is increased from 128 bits to 256 bits, and renamed from XMM0–XMM7 to YMM0–YMM7 (an existig SSE registers (XMM0–XMM7) are mapped to lower 128-bits of YMM0–YMM7 registers). Please read chapter ?? for more details.

[¶]Being more precisely, registers: ST(0) to ST(7) works as an "aliases" for directly unaccessible registers R0-R7.

Bit	Abbreviation	Description	Category
16	RF	Resume Flag (386+ only)	System
17	VM	Virtual-8086 Mode (386+ only)	System
18	AC	Alignment Check (486SX+ only)	System
19	VIF	Virtual Interrupt Flag (Pentium+)	System
20	VIP	Virtual Interrupt Pending flag (Pentium+)	System
21	ID	Identification Flag (Pentium+)	System

Tabela 14.3: Intel x86 EFLAGS register (high half). Those bits that are not listed are reserved by Intel.

Flag	Set when...
AC	Alignment Check. Set if alignment checking of memory references is done.
ID	Identification Flag. Support for CPUID instruction if can be set.
RF	Response to debug exceptions.
VIF	Virtual Interrupt Flag. Virtual image of IF.
VIP	Virtual Interrupt Pending flag. Set if an interrupt is pending.
VM	Virtual-8086 Mode. Set if in 8086 compatibility mode.

Tabela 14.4: Meaning of the Intel x86 EFLAGS register (high half).

14.3.3 64-bit architecture

Starting with the AMD Opteron processor, the x86 architecture extended the 32-bit registers into 64-bit registers in a way similar to how the 16 to 32-bit extension took place – an R prefix identifies the 64-bit registers (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, RFLAGS, RIP). Additional eight 64-bit general registers (R8-R15) were introduced. It also introduces a new naming convention:

- R0 is RAX.
- R1 is RCX.
- R2 is RDX.
- R3 is RBX.
- R4 is RSP.
- R5 is RBP.
- R6 is RSI.
- R7 is RDI.

- R8, R9, R10, R11, R12, R13, R14, R15 are the new registers and have no other names.
- R0D–R15D are the lowermost 32 bits of each register. For example, R0D is EAX.
- R0W–R15W are the lowermost 16 bits of each register. For example, R0W is AX.
- R0B–R15B are the lowermost 8 bits of each register. For example, R0B is AL.

SSE instruction set, as we mentioned in section 14.3.2, originally added eight new 128-bit registers known as XMM0 through XMM7. The AMD64 extensions from AMD (originally called x86-64) added a further eight registers XMM8 through XMM15, and this extension is duplicated in the Intel 64 architecture. The registers XMM8 through XMM15 are accessible only in 64-bit operating mode.

In x86-64 mode we have more AVE registers named YMM0 through YMM15.

ZMMX0-ZMMX31

14.3.4 Miscellaneous/special purpose registers

There are registers on the 80386 and higher processors that are not well documented by Intel. These are divided in control registers, debug registers, test registers and protected mode segmentation registers.

1. `CR0` Ten rejestr ma długość 32 bitów na procesorze 386 lub wyższym. Na procesorze x86-64 analogicznie rejestr ten jak i inne kontrolne ma długość 64 bitów. CR0 ma wiele różnych flag, które mogą modyfikować podstawowe operacje procesora. Tabela 14.5 przedstawia rejestr CR0 (domyślnie dana operacja jest włączona gdy bit jest ustawiony, czyli ma wartość 1):
2. `CR1` Ten rejestr jest zarezerwowany i nie mamy do niego żadnego dostępu.
3. `CR2` CR2 zawiera wartość będącą błędem w adresowaniu pamięci (ang. Page Fault Linear Address). Jeśli dojdzie do takiego błędu, wówczas adres miejsca jego wystąpienia jest przechowywany właśnie w CR2.
4. `CR3` Używany tylko jeśli bit PG w CR0 jest ustawiony. CR3 umożliwia procesorowi zlokalizowanie położenia tablicy katalogu stron dla obecnego zadania. Ostatnie (wyższe) 20 bitów tego rejestru wskazują na wskaźnik na katalog stron zwany PDBR (ang. Page Directory Base Register).

Bit	Flag	Name	Description
31	PG	Paging Flag	Jeśli ustawiony na 1, stronicowanie włączone. Jeśli bit ma wartość 0 to wyłączone
30	CD	Cache disable	Wyłącz pamięć cache
29	NW	Not Write-Through	Zapis do pamięci, czy przez cache
18	AM	Aligment Mask	Maska wyrównania. Aby ta opcja działała musi być ustawiona na 1, bit AC z rejestrów flag procesora również musi mieć wartość 1 oraz poziom uprzywilejowania musi wynosić 3.
16	WP	Write Protection	Ochrona zapisu
5	NE	Numeric Error	Numeryczny błąd, włącza wewnętrzne raportowanie błędów FPU gdy jest ten bit ustawiony
4	ET	Extension Type	Typ rozszerzenia. Ta flaga mówi nam jaki mamy koprocesor. Jeśli 0 to 80287, gdy 1 to 80387
3	TS	Task switched	Przełączanie zadań, pozwala zachować zadania x87
2	EM	Emulate Flag	Jeśli jest ustawiona nie ma żadnego koprocesora. W przeciwnym wypadku jest obecność jednostki x87
1	MP	Monitor Coprocessor	Monitor Koprocesora, kontroluje instrukcje WAIT/FWAIT
0	PE	Protection Enabled	Jeśli 1 system jest w trybie chronionym. Gdy PE ma wartość 0 procesor pracuje w trybie rzeczywistym

Tabela 14.5: CR0 register flags

5. CR4 Używany w trybie chronionym w celu kontrolowania operacji takich jak wsparcie wirtualnego 8086, technologii stronicowania pamięci, kontroli błędów sprzętowych i innych. Tabela 14.6 przedstawia rejestr CR4.
6. debug registers (DR0 through 3, plus 6 and 7)
7. test registers (TR3 through 7; 80486 only)
8. descriptor registers (GDTR, LDTR, IDTR)
9. task register (TR)

Bit	Flag	Name	Description
13	VMXE	Enables VMX	Włącza operacje VMX
10	OSXMMEXCPT	Operating System Support for Unmasked SIMD Floating-Point Exceptions	Wsparcie systemu operacyjnego dla niemaskowalnych wyjątków technologii SIMD
9	OSFXSR	Operating system support for FXSAVE and FXSTOR instructions	Wsparcie systemu operacyjnego dla instrukcji FXSAVE i FXSTOR
8	PCE	Performance-Monitoring Counter Enable	Licznik monitora wydajności. Jeśli jest ustawiony rozkaz RDPMC może być wykonany w każdym poziomie uprzywilejowania. Zaś jeśli wartość tego bitu wynosi 0, rozkaz może być wykonany tylko w trybie jądra (poziom 0)
7	PGE	Page Global Enabled	Globalne stronicowanie
6	MCE	Machine Check Exception	Sprawdzanie błędów sprzętowych jeśli bit ten ma wartość 1. Dzięki temu możliwe jest wyświetlenie przez system operacyjny danych na temat tego błędu jak np w systemie Windows na ”błękintym ekranie śmierci”
5	PAE	Physical Address Extension	Jeśli bit jest ustawiony to zezwalaj na użycie 36-bitowej fizycznej pamięci
4	PSE	Page Size Extensions	Rozszerzenie stronicowania pamięci. Jeśli 1 to stronice mają wielkość 4 MB, w przeciwnym przypadku 4 KB
3	DE	Debugging Extensions	Rozszerzenie debugowania
2	TSD	Time Stamp Disable	Jeśli ustawione, rozkaz RDTSC może być wykonany tylko w poziomie uprzywilejowania 0 (czyli w trybie jądra), zaś gdy równe 0 w każdym poziomie uprzywilejowania
1	PVI	Protected Mode Virtual Interrupts	Jeśli ustawione to włącza sprzętowe wsparcie dla wirtualnej flagi przerwań (VIF) w trybie chronionym
0	VME	Virtual 8086 Mode Extensions	Podobne do wirtualnej flagi przerwań

Tabela 14.6: CR4 register flags

ROZDZIAŁ 1

Memory

15.1 Itroduction

15.1.1 Data representation – endianness

x86 architecture use the little-endian format to store bytes of multibyte values. Oznacza to, że wielobajtowe wartości są zapisane w kolejności od najmniej do najbardziej znaczącego (patrząc od lewej strony), bardziej znaczące bajty będą miały "wyższe" (rosnące) adresy. Notice, that the order of bytes is reversed but not bits. Zatem 32-bitowa wartość B3B2B1B0 mogłaby by na procesorze z rodziny x86 być zaprezentowana w ten sposób: Reprezentacja kolejności typu little-endian Byte 0 Byte 1 Byte 2 Byte 3 Przykładowo 32-bitowa wartość 1BA583D4h (literka h w Asemblerze oznacza liczbę w systemie szesnastkowym, tak jak 0x w C/C++) mogłaby zostać zapisana w pamięci mniej więcej tak: Przykład D4 83 A5 1B Zatem tak wygląda nasza wartość (0xD4 0x83 0xA5 0x1B) gdy zrobimy zrzut pamięci.

15.1.2 Memory segmentation

Memory segmentation is the division of computer's primary memory into segments or sections. The size of a memory segment is generally not fixed[*] and may be even as small as a single byte. Segments usually represent natural divisions of a program such as individual routines, data tables or simply data and execution code part so concept of segmentation is not abstract idea to the programmer. With every segment there are some basic information associated with it

[*]In a sense, that differnt segments could have different lengt.

- length of the segment,
- set of permissions,
- information indicates where the segment is located in memory,
- flag indicating whether the segment is present in main memory or not.

A process is allowed to make a reference into a segment if the type of reference is allowed by the permissions, and the offset within the segment is within the range specified by the length of the segment. Otherwise, a hardware exception such as a *segmentation fault* is raised. That is why memory segmentation is one of the methods of implementing memory protection[†]. The information about location in memory might be the address of the first location in the segment, or the address of a page table for the segment if the segmentation is implemented with paging. When a reference to a location within a segment is made

- the offset within the segment will be added to address of the first location in the segment to give the address in memory of the referred-to item (the first case);
- the offset of the segment is translated to a memory address using the page table (the second case).

If an access is made to the segment that is not present in main memory, an exception is raised, and the operating system will read the segment into memory from secondary storage. The part of CPU responsible for translating a segment and offset within that segment into a memory address, and for performing checks to make sure the translation can be done and that the reference to that segment and offset is permitted is called a memory management unit (MMU).

With memory segmentation a linear address is obtained combining (typically by addition) the **segment address** with **offset** (within this segment). For instance, the segmented address ABCDh:1234h has a segment selector of ABCDh, representing a segment address of ABCDh, to which we add the offset, yielding the linear address 06EF0h + 1234h = 08124h.

If you want to know more... 15.1 (Paging)**.** *tutu - uzupelnic*

[†]Another method is paging; both methods can be combined.

15.1.3 Addressing mode

The addressing mode indicates the manner in which the operand is presented. There is a nice analogy from real live. Generaly the following addressing mode could be considered.

- Immediate. In this type of addressing opperands are dostepne immediately after instruction is read, because actual values are stored in the field.

  ```
  For example:

  xx - instruction code
  aaa - field for operand 1
  bbb - field for operand 2

  xxaaabbb - binary sequence representing instruction

  aaa - actual value of the operand 1
  bbb - actual value of the operand 2
  ```

- Direct. In this type of addressing addresses of actual values are stored in the operand fields of instruction

  ```
  For example:
              Address  Value
  xxaaabbb       1001  0010
     |  |        1010  0011
     |  +------> 1011  0100
     |           1100  0101
     +---------> 1101  0110

  Actual value of the operand 1 (0100) is uder address aaa (1011)
  Actual value of the operand 2 (0110) is uder address bbb (1101)
  ```

- Indirect.

```
For example:

xx - instruction code
aaa - space for operand 1
bbb - space for operand 2

xxaaabbb - binary sequence representing instruction

aaa - actual value of the operand 1
bbb - actual value of the operand 2
```

The registers used for indirect addressing are BX, BP, SI, DI

- Base-index Considering an array, for example, BX contains the address of the beginning of the array, and DI contains the index into the array.

```
For example:

xx - instruction code
aaa - space for operand 1
bbb - space for operand 2

xxaaabbb - binary sequence representing instruction

aaa - actual value of the operand 1
bbb - actual value of the operand 2
```

15.2 Real mode

During the late 1970s it became clear that the 16-bit 64-KiB address limit of minicomputers would not be enough in the future. The 8086 prcessor was developed from the simple 8080 microprocessor and primarily aiming at very small, inexpensive computers and other specialized devices. Thus simple segment registers, enabling memory segmentation, were adopted which increased the memory address width by (only) 4 bits. The effective 20-bit address space of real mode limits the addressable memory to 2^{20} bytes, or 1,048,576 bytes. The number 20 is derived directly from the hardware design of the Intel 8086, which had exactly 20 address pins.

Each segment begins at a multiple of 16 bytes, from the beginning of the linear (flat) address space resulting in 16 byte intervals. The actual location of the beginning of a segment in the linear address space can be calculated with multiplying segment number by 16. For example a segment value of `000Ah` (10) would give an linear address at `00A0h` (160) in the linear address space. Then the address offset can be added to the segment address: `000Ah:0000Bh` (10:11) would be interpreted as `000Ah + 0000Bh = ABh` $(10 \cdot 16 + 11 = 171)$ where `ABh` is the linear address[‡]. Since all segments are 64 KiB long $(65536 \cdot 16 = 1,048,576)$, a single linear address can be mapped to up to 4096 distinct `segment:offset` pairs. For example, the linear address `01234h` (4660) can have the segmented addresses `0000h:01234h` $(0 \cdot 16 + 4660 = 0 + 4660)$, `0123h:0004h` $(291 \cdot 16 + 46 = 4656 + 4)$, `00ABh:0784h` $(171 \cdot 16 + 46 = 2736 + 1924)$, etc. The 16-bit segment selector is interpreted as the most significant 16 bits of a linear 20-bit address (called a segment address) of which the remaining four least significant bits are all zeros. The segment address is always added with a 16-bit offset to yield a linear address, which is the same as physical address in this mode (see image ??).

rysunek

`rysunek`

Now there is a tricky part. The last segment, FFFFh (65535) as we use 16 bits as a segment selector, begins at linear address FFFF0h (1048560) – this is 16 bytes before the end of the 20 bit address space range from 0 to 1,048,576. Thus with an offset of up to 65,536 bytes, one can access, up to 65,520 (65,536-16) bytes past the end of the 20 bit 8088 address space. On the 8088, these address accesses were wrapped around to the beginning of the address space such that `FFFFh:00010h` (65535:16) would access address 0 and `FFE8h:` (65512:80) would access address 304 of the linear address space.

Remark 15.1 (Segment length in real mode)**.** *Real mode segments are always 64 KiB long – in practice it means only that* no segment can be longer than 64 KiB *than that* every segment must be 64 KiB long. *Because in real mode there is no protection or privilege limitation, any program can always access any memory (since it can arbitrarily set segment selectors to change segment addresses with absolutely no supervision). Even if a segment could be defined to be smaller than 64 KiB, it would still be entirely up to the programs to coordinate and keep within the bounds of*

[‡]Such address translations are carried out by the segmentation unit of the CPU.

their segments. Therefore, real mode can just as well be imagined as having a variable length for each segment, in the range 1 to 65536 bytes, that is just not enforced by the CPU.

15.2.1 Addressing modes

In real mode there are several addressing modes.

- Register addressing

  ```
  mov ax, bx  ; moves contents of register bx into ax
  ```

- Immediate

  ```
  mov ax, 1   ; moves value of 1 into register ax
  ```

- Direct memory addressing

  ```
  mov ax, [102h] ; Actual address is DS:0 + 102h
  ```

- Direct offset addressing

  ```
  byte_tbl db 12,15,16,22,..... ; Table of bytes
  mov al,[byte_tbl+2]
  mov al,byte_tbl[2] ; same as the former
  ```

- Register Indirect

  ```
  mov ax,[di]
  ```

  The registers used for indirect addressing are BX, BP, SI, DI

- Base-index

  ```
  mov ax,[bx + di]
  ```

  Considering an array, for example, BX contains the address of the beginning of the array, and DI contains the index into the array.

- Base-index with displacement

  ```
  mov ax,[bx + di + 10]
  ```

15.3 Protected mode

In protected mode, a segment register no longer contains the physical address of the beginning of a segment, but contain a "selector" that points to a system-level structure called a segment descriptor. A segment descriptor contains the physical address of the beginning of the segment, the length of the segment, and access permissions to that segment. The offset is checked against the length of the segment, with offsets referring to locations outside the segment causing an exception. Offsets referring to locations inside the segment are combined with the physical address of the beginning of the segment to get the physical address corresponding to that offset. The segmented nature can make programming and compiler design difficult because the use of near and far pointers affects performance.

15.4 Virtual memory

DODATEK A

???

NASM

32-bit program on 32-bit system

```
nasm -f elf hello.asm
ld hello.o -o hello
```

32-bit program on 64-bit system

```
nasm -f elf hello.asm
ld -m elf_i386 hello.o -o hello
```

32-bit program on 64-bit system (but it's not true 64-bit program)

```
nasm -f elf64 hello.asm
ld hello.o -o hello
```

64-bit program on 64-bit system

```
nasm -f elf64 hello_64.asm -o hello_64
ld hello_64.o -o hello_64
```

32-bit program linked with a C library on 32-bit system

```
nasm -f elf hello_c.asm -o hello_c.o
gcc hello_c.o -o hello_c
```

32-bit program linked with a C library on 64-bit system

```
nasm -f elf32 simple_printf_32.asm -o simple_printf_3
gcc -m32 simple_printf_32.o -o simple_printf_32
```

64-bit program linked with a C library on 64-bit system

```
nasm -f elf64 hello_c_64.asm -o hello_c_64.o
gcc hello_c_64.o -o hello_c_64
```

GNU AS

32-bit program on 32-bit system

```
as hello.s -o hello.o
ld hello.o -o hello
```

32-bit program on 64-bit system

```
as --32 hello.s -o hello.o
ld -m elf_i386 hello.o -o hello
```

Bibliografia

[1] David Salomon, *Assemblers and Loaders*, http://www.davidsalomon.name/assem.advertis/asl.pdf, retrived 2013-01-17.

[2] Lamont Wood, *Forgotten PC history: The true origins of the personal computer*, August 8, 2008 (Computerworld), http://www.computerworld.com/s/article/print/9111341/Forgotten_PC_history_The_true_origins_of_the_personal_computer, retrived on 2013-03-13.

[3] Peter van der Linden, *Expert C Programming: Deep C Secrets*, Prentice Hall 1994, p. 141, (retrived on 2013-04-22, http://books.google.pl/books?id=4vm2xK3yn34C&pg=PA141&redir_esc=y#v=onepage&q&f=false)

[4] *Intel® 64 and IA-32 Architectures. Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, retrived on 2013-04-05.

[5] *Intel MMX^{TM} Technology Overview*, March 1996, retrived on 2013-05-09 from http://www.zmitac.aei.polsl.pl/Electronics_Firm_Docs/MMX/overview/24308102.pdf.

[6] *The NASM Language* retrived on 2015-03-13 from http://www.nasm.us/doc/nasmdoc3.html

[7] *Using MMX^{TM} Instructions to Compute a 16-Bit Vector*, March 1996, retrived on 2013-05-01 from http://software.intel.com/sites/landingpage/legacy/mmx/MMX_App_Compute_16bit_Vector.pdf.

[8] *Using the RDTSC Instruction for Performance Monitoring*, Intel Corporation, 1997, retrived on 2013-04-29, from http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

[9] *MMX technology*, retrived on 2013-05-09, from http://web.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p3/mmx.htm.

[10] *Time Stamp Counter*, retrived on 2015-05-10, from http://en.wikipedia.org/wiki/Time_Stamp_Counter.

Spis rysunków

Spis tabel

Skorowidz