Uniwersytet Łódzki

Wydział Matematyki i Informatyki

Informatyka

Lecture Notes in Assembly Language

Short introduction to low-level programming

Piotr Fulmański

Łódź, 2013

Spis treści

ROZDZIAŁ 1

Before we begin

1.1 Simple assembler

Before we start, I think, that it's not bad idea to practise with wery 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 4 digit instruction of the following format

```
operation code
|
xxxx
 | |
 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 BRA unconditional branche to instruction located at specified address
```

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

The number 9 is reserved for future extensions. 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.

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

Address	Value	Instruction	Accumulator
0010	1006	CPA 6	20
0011	3007	ADD 7	20+30
0012	3008	ADD 8	20+30+40
0013	2009	STO 9	no change
0014	0000	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

Address	Value	Instruction	Accumulator
0010	1006	CPA 6	3
0011	7005	MUL 5	3*2
0012	3007	ADD 7	3*2+4
0013	2004	STO 4	no change
0014	0000	HLT	

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
0010	1005	CPA 5
0011	7005	MUL 5
0012	7005	MUL 5
0013	7006	MUL 6
0014	2004	STO 4

0015	1005	CPA 5
0016	7005	MUL 5
0017	7007	MUL 7
0018	3004	ADD 4
0019	2004	STO 4
0020	1005	CPA 5
0021	7008	MUL 8
0022	3004	ADD 4
0023	2004	STO 4
0024	1009	CPA 9
0025	3004	ADD 4
0026	2004	STO 4
0027	0000	HLT

Solution 3.2

Address	Value	Instruction
0010	1005	CPA 5
0011	7005	MUL 5
0012	7005	MUL 5
0013	7006	MUL 6
0014	2100	STO 100
0015	1005	CPA 5
0016	7005	MUL 5
0017	7007	MUL 7
0018	2101	STO 101
0019	1005	CPA 5
0020	7008	MUL 8
0021	2112	STO 112
0022	1009	CPA 9
0023	3100	ADD 100
0024	3111	ADD 111
0025	3112	ADD 112

0026	2004	STO 4
0027	0000	HLT

Solution 3.3

Address	Value	Instruction	Accumulator
0010	1006	CPA 6	a
0011	7005	MUL 5	ax
0012	3007	ADD 7	ax + b
0013	7005	MUL 5	(ax + b)x
0014	3008	ADD 8	(ax+b)x+c
0015	7005	MUL 5	((ax+b)x+c)x
0016	3009	ADD 9	((ax+b)x+c)x+d
0017	2004	STO 4	no change
0018	0000	HLT	

1.1.4 Excercise 4

Calculate a to the power b.

Address	Value
0001	number 1
0002	number 2

Solution 4.1

Address	Value	Instruction
0001	xxxx	a
0002	xxxx	b
0003	0001	1
0004	xxxx	result
0005	1003	CPA 3
0006	2004	STO 4
0007	1002	CPA 2
0008	8015	BRZ 15

0009	4003	SUB 3
0010	2002	STO 2
0011	1004	CPA 4
0012	7001	MUL 1
0013	2004	STO 4
0014	8007	BRZ 7
0015	0000	HLT

Solution 4.2

Address	Value	Instruction
0001	xxxx	a
0002	xxxx	b
0003	0001	1
0004	xxxx	result
0005	1003	CPA 3
0006	2015	STO 4
0007	1002	CPA 2
0008	8014	BRZ 15
0009	4003	SUB 3
0010	2002	STO 2
0011	1015	CPA 4
0012	7001	MUL 1
0013	2015	STO 4
0014	5006	BRA 7
0015	0000	HLT

1.2 Improvements, part I

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 instuction list with two instruction

```
01xx INC address
02xx 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 (123)
| |
| |       Address     Value
3123          ...  |        |
  |         (0122) |        |
  +-------> (0123) |  0035  |
            (0124) |        |
              ...  |        |
```

In the example above instruction ADD adds value (35) from the addres 123. 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 an value use number from the address `number`, while notation

```
inst (number)
```

means: executing instruction `inst` as an value 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 (123) - value of the argument
| |
| |
3123
```

Introducing this type of addressing entails new codes for instruction because computer such as humans have to distinguisg variants of addressing

```
              Direct addressing   Immediate addressing
Human         ADD 35              ADD (5)


Computer      3035                9135

9xxx - to indicate extension of basic instruction set
x1xx - addressing mode (1 for immediate, 1 byte length)
```

```
xx3x - code for addition in basic instructions set
xxx5 - 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         9230 - add
x + 1     0128 - value for add of code 9230
```

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 5

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

1.3 Improvements, part II

- This problem seems to 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 from the address `addr`.

```
+-code for ADD [x] ->--+
|                      +->-- finally: ADD [6] and it adds 123
|  +-operand (6) --->--+              to acumulator
|  |
|  |       Address     Value
9336           ...  |         |
   |         (0005) |         |
   +--------> (0006) |  0009   | ---+
             (0007) |         |    |
               ...  |         |    |
             (0009) |  0123   | <--+
               ...  |         |
```

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`.

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

```
Direct (one-byte) %Bezpośrednie jednobajtowe


910x INC increment  value in memory at specified address
919x DEC decrement  value in memory at specified address
1xxx CPA copy value from memory to accumulator, M -> A
912x STO copy value from accumulator to memory, A -> M
3xxx ADD add value from specified memory cell to accumulator; result is stored
      in accumulator, M + A -> A
4xxx SUB subtract from accumulator value from specified memory cell; result
      is stored in accumulator A - M -> A
915x BRA unconditional branche to instruction located at specified address
916x BRN conditional branche to instruction located at specified address if value
```

```
      stored in accumulator is negative
7xxx MUL multiply value from accumulator by value from specified memory cell;
      result is stored in accumulator M * A -> A
918x BRZ conditional branche to instruction located at specified address if value
      stored in accumulator is equal to zero

Direct (two-byte) %Bezpośednie dwubajtowe

9000 xxxx INC
9010 xxxx CPA
9020 xxxx STO
9030 xxxx ADD
9040 xxxx SUB
9050 xxxx BRA
9060 xxxx BRN
9070 xxxx MUL
9080 xxxx BRZ
9090 xxxx DEC

Immediate (one-byte) %Natychmiastowe jednobajtowe

0xxx HLT stop the cpu
01xx INC
911x CPA
2xxx STO
913x ADD
914x SUB
5xxx BRA
6xxx BRN
917x MUL
8xxx BRZ
02xx DEC
```

```
Immediate (two-byte) %Natychmiastowe dwubajtowe

9200 xxxx INC

9210 xxxx CPA

9220 xxxx STO

9230 xxxx ADD

9240 xxxx SUB

9250 xxxx BRA

9260 xxxx BRN

9270 xxxx MUL

9280 xxxx BRZ

9290 xxxx DEC

Indirect (one-byte) %Pośrednie jednobajtowe

---- INC (not applicable)

931x CPA

---- STO (not applicable)

933x ADD

934x SUB

---- BRA (not applicable)

---- BRN (not applicable)

937x MUL

---- BRZ (not applicable)

---- DEC (not applicable)

Indirect (two-byte) %Pośrednie dwubajtowe

---- xxxx INC (not applicable)

9410 xxxx CPA

---- xxxx STO (not applicable)
```

```
9430 xxxx ADD

9440 xxxx SUB

---- xxxx BRA (not applicable)

---- xxxx BRN (not applicable)

9470 xxxx MUL

---- xxxx BRZ (not applicable)

---- xxxx DEC (not applicable)
```

Notice that in instruction list some instruction are missed. Explanation for this is as folow.

```
Explain that direct addressing for jump or inc/dec is like indirect for addition.
```

Solution 5.2.1 – second approach

```
Address Value      Instruction

0001    0010       address of the first component of vector 1

0002    0020       address of the first component of vector 2

0003    0000       result

0004    0010       n - length of vector

...

0010    xxxx       first component of vector 1

...

0019    xxxx       last component of vector 1

0020    xxxx       first component of vector 2

...

0029    xxxx       last component of vector 2

0030    1004       CPA 4

0031    8040       BRZ 40

0032    9311       CPA [1]

0033    9732       MUL [2]

0034    3003       ADD 3

0035    2003       STO 3

0036    0101       INC 1

0037    0102       INC 2
```

```
0038     0204        DEC 4
0039     5030        BRA 30
0040     0000        HLT
```

1.3.1 Solution 5.2.2 – bad second approach

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 10) symbolic names are correct but their addresses are not.

```
Address Value      Instruction
0011               address of the first component of vector 1
0012               address of the first component of vector 2
0013               result
0014               n - length of vector
...
0020               first component of vector 1
...
0029               last component of vector 1
0030               first component of vector 2
...
0039               last component of vector 2
0040               CPA 14
0041               BRZ 50
0042               CPA [11]
0043               MUL [12]
0044               ADD 13
0045               STO 13
0046               INC 11
0047               INC 12
0048               DEC 14
0049               BRA 40
0050               HLT
```

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

```
Address Value      Instruction
0011    0020       address of the first component of vector 1
0012    0030       address of the first component of vector 2
0013    0000       result
0014    0010       n - length of vector
...
0020    xxxx       first component of vector 1
...
0029    xxxx       last component of vector 1
0030    xxxx       first component of vector 2
...
0039    xxxx       last component of vector 2
0040    1014       CPA 14
0041    8050       BRZ 52
0042    9410       CPA [11]
0043    0011
0044    9470       MUL [12]
0045    0012
0046    3013       ADD 13
0047    2013       STO 13
0048    0111       INC 11
0049    0112       INC 12
0050    0214       DEC 14
0051    5040       BRA 40
0052    0000       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 [11]
```

In the first case we have

```
Address Value      Instruction
x       9311       CPA [1]
```

and the second

```
Address Value      Instruction
x       9410       CPA [11]
x+1     0011
```

1.4 Improvements, part III

- 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

```
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 6

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

```
.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 od 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???

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

- That's right – we can solve the problem (1.4.1) the way we proposed, but the method used to passing argument is far from perfection. Better choice is to use data structure which help us to keep a correct order of the arguments – this is how we reach the concept of stack. Short description of the stack put here.

  Introduce stack. Notice one very important thing: stack in computers growth in direction of lower addresses. It means that if element x is above y the address of y is lower than x. To keep things working we also have to introduce two new registers in our CPU

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

  with instruction

  ```
  PUSH (rejestrowa i ewentualnie pamieciowe)
  POP
  ```

1.5.1 Excercise 6 – second approach

1.5.2 Excercise 7

Calculate the dot product of two vectors using stack.

1.5.3 Excercise 8

Find the value of the n-th element of the Fibonacci sequence.

1.6 Improvements, part V

The solution we found 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 assume that called function knows which function or part of the case was a caller – in our case, "main" code – and we hardcoded this value in our function. And what if we call function from completely different place, for example other function? We return to "main" code which wouldn't be correct.

- Introduce frame stack to keep info about ret.

```
Frame stack:

higher addresses
:    :
|  2 | [ebp + 16] (3rd function argument)
|  5 | [ebp + 12] (2nd argument)
| 10 | [ebp + 8]  (1st argument)
| RA | [ebp + 4]  (return address)
| FP | [ebp]      (old ebp value)
|    | [ebp - 4]  (1st local variable)
:    :

stack growth
```

1.6.1 Excercise 9

Funkcja dodająca dwa argumentu i zwracająca wynik.

a: 2 b: 5 wynik: 0 .code 10 BRA dodaj powrot: HLT

dodaj: CPA a ADD b STO wynik BRA powrot

teraz to samo, ale z dowma dodawaniami

rozwiazanie ze stosem

a: 2 b: 5 wynik: 0 .code 10 start: PUSH wynik PUSH a PUSH b CALL dodaj POP wynik dodaj: CPA [SP + 1] ADD [SP + 2] STO [SP + 3] RET 2

PUSH $2PUSH3$ CALL dodaj CPA [SP + 1] ADD [SP + 2] STO [SP + 2] POP STO SP RET

1.6.2 Excercise 10

Solve once again the problem from the exercise 1.5.3 using improved stack.

1.7 Other excercises

1.7.1 Excercise 11

Program ktory dzieli dwie liczby calkowite i jako wynik podaje czesc calkowita i reszte

```
dzielna:  20
dzielnik: 7
reszta:   0
wynik:    0

start:    CPA dzielna
          BRZ koniec
          BRN reszta_koniec
          INC wynik
          STO dzielna
          BRZ koniec
          BRA start

reszta_koniec:
          CPA dzielna
          STO reszta
koniec:   HLT
```

1.7.2 Excercise x

Program porządkujący liczby.

1.7.3 Excercise x

Program znajdujący najmniejszą i najwieksza sposrod 4 liczb.

1.7.4 Excercise x

1.7.5 Excercise x

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

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

```
Address Value
1000    number 1
1001    number 2
```

1.7.6 Solution x

```
Address Instruction           Accumulator
```

```
0200    1 1000
0201    4 1001      a
0202    6 0205      ax
0203    8 0212      ax+b
0204    5 0201      (ax+b)x
0205    3 1001      (ax+b)x+c
0206    2 1002      ((ax+b)x+c)x
0207    1 1001      ((ax+b)x+c)x+d
0208    2 1000
0209    1 1002
0210    2 1001
0211    5 0200
0212    0 0000
```

1.7.7 Excercise x

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

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

```
Address Instruction           Accumulator
0001    1 1000
0002    6 0004
0003    0 0000
0004    1 1001
0005    4 1000
0006    2 1000
0007    0 0000
```

ROZDZIAŁ 2

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

2.1 Assembly language

Because this book is about assembly languages, let's try to understand what an assebly language is. Simply speaking

Definition 2.1. *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.

2.2 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.

2.2.1 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... 2.1 (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.*

2.2.2 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.

2.2.3 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. As en example consider the following set of instruction

- 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.

2.2.4 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.

2.2.5 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. The Intel 80386 The Intel 80386 (GNU FDL 1.2)

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. The AMD Am486 The AMD Am486, an Intel 486 competitor (GNU FDL 1.2)

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. Pentium Pro Intel's Pentium Pro (GNU FDL 1.2)

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. AMD64 logo AMD64, a rebranding of x86-64

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.

2.2.6 Late-x86 age – stone age devices

tutu

- 2003: AMD introduces the x86-64, a 64-bit superset of the x86 instruction set.

  2004: AMD demonstrates an x86 dual-core processor chip. Pentium D Intel's first dual-core chip, the Pentium D

  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.

2.3 An overview of the x86 architecture

2.3.1 Basic properties of the architecture

tutu

2.3.2 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 3

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

3.1 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... 3.1 (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... 3.2 (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... 3.3 (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.*

3.2 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 varius 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... 3.4 (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.*

3.3 x86 registers

3.3.1 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 3.1 and 3.2 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 3.1: 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 3.2: Meaning of the Intel x86 FLAGS register.

3.3.2 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 3.3 and 3.4 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, a 32-bit Streaming SIMD Extensions (SSE) control/status register (MXCSR) and eight 128-bit SSE floating point registers (XMM0 to XMM7) were added.

3.3.3 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. The least significant 32 bits of these registers

[¶]Being more precisely, registers: ST(0) to ST(7) works as an "aliases" for directyle 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 3.3: 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 3.4: Meaning of the Intel x86 EFLAGS register (high half).

are available via a D suffix (R8D through R15D), the least significant 16 bits via a W suffix (R8W through R15W), and the least significant 8 bits via a B suffix (R8B through R15B).

3.3.4 Miscellaneous/special purpose registers

1. 128-bit SIMD registers XMM0 - XMM15
2. 256-bit SIMD registers YMM0 - YMM15
3. 512-bit SIMD registers ZMM0 - ZMM31
4. control registers (CR0 through 4, CR8 for 64-bit only) 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. Nas jednak będą interesowały szczególnie 6 bitów tego rejestru - dolne 5 (od PE do ET) oraz najwyższy bit (PG). Tabelka przedstawia rejestr CR0 (domyślnie dana operacja jest włączona gdy bit jest ustawiony, czyli ma wartość 1): Bit Nazwa Nazwa angielska Opis 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 CR1 Ten rejestr jest zarezerwowany i nie mamy do niego żadnego dostępu. 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. 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). CR4 Używany w trybie chronionym w celu kontrolowania operacji takich jak wsparcie wirtualnego 8086, technologii stronicowania pamięci, kontroli błędów sprzętowych i innych. Bit Nazwa Nazwa angielska Opis 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

5. debug registers (DR0 through 3, plus 6 and 7)

6. test registers (TR3 through 7; 80486 only)

7. descriptor registers (GDTR, LDTR, IDTR)

8. task register (TR)

ROZDZIAŁ 4

Memory

4.1 Itroduction

4.1.1 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.

4.1.2 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... 4.1 (Paging). *tutu - uzupelnic*

[†]Another method is paging; both methods can be combined.

4.1.3 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
```

4.2 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 4.1 (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.

4.2.1 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]
  ```

4.3 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.

4.4 Virtual memory

ROZDZIAŁ 5

First program

It should be familiar after reading

5.1 32-bit basic stand alone program

5.1.1 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=cr
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 stdou
        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 runing

```
./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... 5.1 (Making 32-bit code on 64-bit system with NASM)**.** *When you try to make 32-bit program on 64-bit system assembling 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
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... 5.2 (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  ................
```

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.
- `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 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
  ```
- `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.

[*]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.

- `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 function.

  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 function. Values in registers RCX and R11 are destroyed.

- `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

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

List file tutu

For the above code, the content of listfile is generated as follow

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

                                 section .data              ; Data section

00000000 48656C6C6F20576F72-    text    db "Hello World!", 10  ; The string to pr
00000009 6C64210A
                                 len     equ $-text         ; "$" means "here"
                                                            ; len is a value, not

                                 section .text              ; Code section

                                 global  _start             ; Make label available
                                                            ; We must export the e
                                                            ; loader. They convent
                                                            ; entry point. Use ld

                                 _start:                    ; Standard  ld  entry
00000000 66BA0D000000                    mov     edx,len    ; arg3: length of stri
00000006 66B9[00000000]                  mov     ecx,text   ; arg2: pointer to str
0000000C 66BB01000000                    mov     ebx,1      ; arg1: where to write
00000012 66B804000000                    mov     eax,4      ; System call number (
00000018 CD80                            int     0x80       ; Interrupt 80 hex, ca

                                 ; Exit
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 say that tutu

5.1.2 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=cr
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
```

```
        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... 5.3 (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
```

The main reason for this is different syntax used by NASM (Intel syntax) and GNU AS (AT&T syntax). See next section for more details; now only the most conspicuous differences would be comment.

- 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..

[†]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`

- 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.

5.1.3 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

```
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

  ```
  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]
  ```

5.2 64-bit basic stand alone program

5.2.1 Code for NASM

../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. (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=cr
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 stdou
        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
```

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 5.1.

Notice that taking code from section 5.1 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 5.1 any of the programs is not truly 64-bit.

5.3 32-bit basic program linked with a C library

5.3.1 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... 5.4 (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 **??**).

5.3.2 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 `[esp4]`+, 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.

5.3.3 Excercise

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

../programs/first_program/simple_printf_32.c

```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
- 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].

5.4 64-bit basic program linked with a C library

5.4.1 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 EAX 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 ??).

5.4.2 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.
- 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.

5.4.3 Excercise

Write a 64-bit program from excercise 5.3.3.

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. Why a misaligned stack causes a seg fault only when using a vector register (a register! not the stack!) isn't entirely clear to me. Probably a lack of understanding of how varargs work. . .

If you want to know more. . . 5.5 (Prying assembler code generated by GCC)**.** *Sometimes, when we drop into troubles, it's very useful to inspect (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 ...]
```

ROZDZIAŁ 6

Basic CPU instructions

../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

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:  equ $ - 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
```

6.0.4 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 destruct

    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 10*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
```

6.0.5 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
```

6.0.6 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

  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]
```

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

```
    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 – to be stack, or not to be a stack, that is the question

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 showned.

7.1 FPU internals

7.2 Instructions related to the FPU internals

../programs/fpu/fpu_test_01_32.asm

```
section .data

fmt: db 10,"overflow:␣%d",10,"top:␣%d",10,"R7␣%d",10,"R6␣%d",10,"R5␣%d",10,"R4␣%d",10,"R3␣%d

section .bss

env: resd 7

buf: resw 1

section .text

```

```
extern printf

global main

main:

  fld1
  fld1
  fld1
  fld1
  call ptw
  faddp st3, st0
  call ptw

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

; Auxiliary print code
ptw:
  fstenv [env]                  ; saving fpu state
  xor eax, eax
  mov ax, [env+8]

  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

    push ebx

    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 pointer value
  and bx, 7                       ; A bit-wise AND of the two operands:
                                  ; ax and binary pattern 111
  push ebx

  mov ebx, eax
         ;xxxxxxxxx1xxxxx1   64 - Stack Fault + 1 - Invalid Operation
  and bx, 0000000001000001b ; A bit-wise AND of the two operands:
                                  ; ax and binary pattern 1
  push ebx

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

; End of the code
```

../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:
  fstcw [buf]          ; Save control word
                    ;xxxx11xxxxxxxxxx
  or word [buf], 0000010000000000b ; 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
```

7.2.1 Excercise

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

Solution

../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

  fldz

  loop:                              ; do-while loop begin
    fld qword [vec1 + ecx * 8]       ; Load [ecx] component of vector 1
    fmul qword [vec2 + ecx * 8]      ; Multiply eax 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

    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

MMX

8.1 Multi-Media eXtensions

The one think we can say about MMX is that this is not a multipurposes tehnology. 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.

- *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*
- *57 new instructions*

8.1.1 Single Instruction, Multiple Data (SIMD) technique

tutu

8.1.2 Eight 64-bit wide MMX registers

MMX had a couple of design goals which are very important. For the most part they were listed earlier, but I'm going to list them again, since they really are important. 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 (aka multitasking). 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.

[†]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).

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.

8.1.3 Four new data types

tutu

8.1.4 24 new instructions

tutu

8.1.5 Excercise

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

Solution

Taking into account all the above, it is not possible to write with MMX equivalent of the code 7.2.1 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 32-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
                     ; 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
```

Better solution (faster) of this excercise could be found in [6]. To verify if it's realy better, reader could use RDTS instruction – see chapter 10.

ROZDZIAŁ 9

SSE

9.1 Streaming Simd Extensions

Like MMX is tuned for working with bytes or words (8 or 16-bit integers) the SSE is tuned for working with single-precision floating-point values. If you need doubles, read next chapter.

9.1.1 Excercise

Write a program calculating a dot product of two vector (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
;         18.0, 34.0, 48.0, 60.0, 70.0, 78.0, 84.0, 88.0, 90.0, 90.0 ; results of mul.
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 regis
  movups xmm1, [esi]   ; Copy four 32-bit floating-point values from vector 2 into XMM1 regis
  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 them
  push edx                         ; 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
```

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

10.1 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

$$s = fraccf$$

where s is time in seconds, c is the number of cycles and f is the frequency.

10.2 Usage of the RDTS

Prevent from out-of-order execution

../programs/rdtsc/01.asm

```
rdtsc              ; Read time stamp counter
```

Speed [GHz]	Max time for 32-bit counter [s]	Max time for 64-bit counter [s]
0.5	8.5899	
1	4.2949	
1.5	2.8633	
2	2.1474	
2.5	1.7179	
3	1.4316	
1	a	b

Tabela 10.1: Maximum TSC value and real time for selected frequencies.

```
mov [time], eax ; Copy counter into variable
...             ; Do something
rdtsc           ; Read time stamp
sub eax, [time] ; Find the difference
```

../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[7].

Caching data nad code

10.2.1 Usage example

../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

; Floating point test start

; 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
```

../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

; Floating point test start

; 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
  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
```

10.2.2 Excercise

Write a program calculating a dot product of two vector (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
;          18.0, 34.0, 48.0, 60.0, 70.0, 78.0, 84.0, 88.0, 90.0, 90.0 ; results of mul.
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 regis
  movups xmm1, [esi]   ; Copy four 32-bit floating-point values from vector 2 into XMM1 regis
  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 them
  push edx                       ; 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
```

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] *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.

[7] *Using the RDTSC Instruction for Performance Monitoring*, Intel Corporation, 1997, retrived on 2013-04-29, from http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf.

Spis rysunków

Spis tabel

Skorowidz