# LAB WORK NO. 11
## THE USAGE OF THE MATHEMATICAL COPROCESSOR

## 1. Object of laboratory

The purpose of this lab is to familiarize the user with the mathematical coprocessor's functions, its instructions for real numbers and other coprocessor operations.

## 2. Theoretical considerations

Even tough the 8086,80286,80386 and 80486 SX processors have a series of powerful integer arithmetical they do not support floating point arithmetical operations or integer numbers represented on multiple bytes.

Because of these impediments INTEL developed the INTEL 8087 (80287, 80387) arithmetic coprocessor. As its name says the co-processor is made up of several processors that cooperate with the computer's main processor. The coprocessor can not extract by himself the instructions from the memory, this job that is done by the main processor.

### 2.1. Working principle

The coprocessor is activated at the same time as the system's general RESET signal. This signal brings the coprocessor in its initial state (with error masking, register erase, stack initialization, number rounding, etc.). After the main processor executes the first instruction the coprocessor can detect with which type of processor it has to work. Depending on the processor type the coprocessor will reconfigure itself accordingly. Obsoleted in more modern designs bye monolithic hardware structure of the two processors.

The coprocessor connects to the processors local bus through several lines: data/address, state, clock, ready, reset, test and request/grant. Being connected to the microprocessor's local bus allows the coprocessor access to all memory resources through the request/grand buss request.

The two processors are working in parallel, which implies synchronization between the code running on them.

Usually the error and instruction synchronization is done by compilers and assemblers, while the data synchronization has to be done by the user of the assembling language.

The responsibility of controlling the program belongs to the main processor. Instructions for the coprocessor start with a special ESCAPE code. The coprocessor monitors the instruction flow of the main processor. By decoding the escape code the coprocessor knows when the processor's instruction queue is loaded with instructions for the coprocessor and stores these instructions in its own queue. The ESC instruction code is the following:

| | |
|---|---|
| 1101 1xxx | mod xxx r/m |

X meaning don care. Thus all instructions that have the operation code between D8 and DF will be considered as ESC instructions. Together with the three bits from the second byte there are a total of 64 allowed instructions for the coprocessor.

The microprocessor executes the ESC instruction by computing a memory address (from mod and r/m) and executes a bus cycle, reading the data from the computed address (if mod is 11 more bus cycles will be executed). The data is not really read from memory instead a bus cycle is generated (it shows a NOP instruction) the coprocessor being activated by the ESC instruction decodes the six bits from the ESC instruction and can capture the address and/or the data from the memory selected by the instruction. This mechanism allows the programmer to treat the ESC instruction (defined by the coprocessor) as a normal instruction with all the addressing methods. If the coprocessor requires more data from memory it can request he control from the microprocessor. The main processor's registers are not accessible to the coprocessor. The coprocessor keeps the TEST line on high as long as he executes to tell the processor that he is busy.

Inside the coprocessor we have an 80 bytes memory organized as a stack of eight 10 byte elements. On these 10 bytes the floating point numbers are represented in IEEE temporary real format. The coprocessor can access the computer's memory with any addressing mode and any legal format of data. The data brought from memory is converted into the coprocessor's internal format and put on top of the stack. When writing to the main memory the internal format is converted into the format specified by the user.

The condition for executing floating point operations in the coprocessor is: the operand has to be at the top of the stack (for the

operations with two operands, at least one of them). So, with the aid of the coprocessor we can do the following operations:

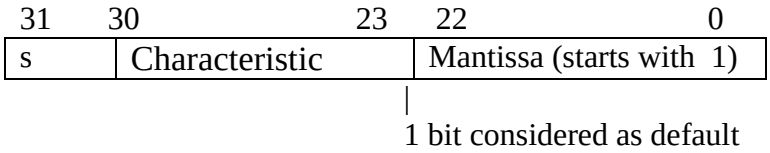
- load data into the coprocessor's internal stack from the computer's memory
- execute the necessary arithmetical operations
- store the results into the computer's memory

## 2.2. INTEL 8087 recognized data types

The great advantage of the coprocessor is that he works not only with floating point numbers but with integer numbers also and recognizes packed decimal data types. So if we have to execute a complicated integer operation and it has to be very fast it can be done with the help of the coprocessor, without having to do a time consuming conversion from integer to floating point and backwards just so that the coprocessor can work with them.

### 2.2.1. Floating point data types

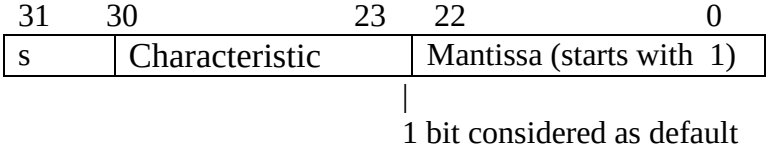
Short Real a 32 bit number represented in floating point. The number is decomposed in mantissa and characteristic. The characteristic is represented on 8 bits from which the most significant is the sign bit and it's treated differently. The physical length of the mantissa is 23 bits. The sigh of the real number is given by the most significant bit:




This floating point number representation always works with normalized numbers, meaning that the mantissa's first bit is always 1, and thus this bit is never written being considered by default. Thus the mantissa's real size is 24 bits.

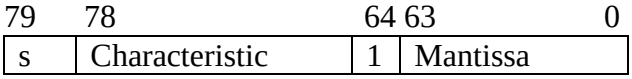
It is important for us to no the actual precision, The mantissa represents 6-7 digits, while the characteristic with its 8 bits raises their number to the order of $\sim 10^{38}$ (the exact number can not be determined because it depends on the mantissa). The highest number is approximately of $1.7 * 10^{38}$ and the lowest positive real number in around $10^{-38}$.

Long Real a 64 bit number represented in floating point. The number is decomposed in mantissa and characteristic. The characteristic is represented on 11 bits from which the most significant is the sign bit and it's treated differently. The physical length of the mantissa is 52 bits. The sigh of the real number is given by the most significant bit:




As in the previous case here also he have a default bit, thus the actual length of the mantissa in 53 bits. And these 53 bits we can represent approximately 16-17 digits, the representation of the smallest number in very precise.

Temporary real numbers an 80 bit number represented in floating point. The number is decomposed in mantissa and characteristic. The characteristic is represented on 15 bits from which the most significant is the sign bit. The physical length of the mantissa is 64 bits. The sigh of the real number is given by the most significant bit:




The temporary floating point numbers are not always normalized. The mantissa does not have to start with 1. Thus, in this case the size of the mantissa in of only 64 bits. The high precision floating point number's 64 bits normalized mantissa represents approximately 19 decimal digits. The length of the characteristic is 15 bits. Because the number is not always normalized the lowest possible number that can be represented is much smaller than we would expect: cca. $10^{-4932}$.
This representation method is highly sensitive to the possibility of the number being different or equal to 0.

### 2.2.2. Signed Integer in 2's Complement

Word, double word and quad word representations are accepted.
DW, DD and DQ.

### 2.2.3. Signed Integer in Packed BCD representation

Signed integer values represented on ten bytes can be used. DT. Numbers up to 18 decimal digits can be represented.

## 2.3. Operation errors (exceptions)

When using floating point operations we can encounter countless errors, starting from trivial logarithmic errors, to errors caused by representation limitations. These we will call exceptions. We will study these types of errors and the ways in which we can manipulate them.

When an error appears the coprocessor can manifest two behavior types. It signals the error using an interrupt if the user validates this. If not, the coprocessor will analyze the error internally and according to the signaled errors will do the following tasks. The coprocessor's designers categorized all errors in the following 6 classes:

### 2.3.1. Invalid operation

This can be: an upper or lower overflow of the coprocessor's internal stack. The lower overflow can appear when we try to access an element that doesn't exist on the stack. These are usually severe algorithmic errors; the coprocessor does not execute the operation.

We have an undefined result if we try to divide 0.0 by 0.0; the coprocessor is not prepared for this. Similar situations appear when we try to subtract infinite from infinite, etc. These errors (even tough they can be avoided by proper algorithms) are not as severe errors as stack overflows.

The same result will be obtained if a coprocessor function is called with wrong parameters.

If an undefined result appears the coprocessor puts into the characteristic a reserved value (bits of 0).

### 2.3.2. Overflow

The result is bigger than the largest number that can be represented. The coprocessor writes the infinite value instead of the result and moves on.

### 2.3.3. Division by zero

The divider is zero while the number to be divided is different from 0 or infinite. The coprocessor writes the infinite value instead of the result and moves on.

### 2.3.4. Underflow

The value of the result is smaller then the smallest number that can be represented. The result will be 0 and the coprocessor will move on.

### 2.3.5. De-normalized Operand

Appears if one of the operands is not normalized or the result can not be normalized (for example if it's so small that it normalization can not be done). The coprocessor moves on (the values that are not 0 will be lost, will be turned into 0).

### 2.3.6. Inexact result

The result of the operation is inaccurate due to necessary or prescribed rounding. This kind of results cad be obtained when dividing 2.0 by 3.0 and the result can be represented only as an infinite fraction. The coprocessor does the rounding and moves on.

The above were described in order of their severity. If a stack overflow appears the program is flawed and it will not be continued.

At the same time a rounding error needs not to be treated. Not even on paper can we use infinite fractions or irrational numbers as we would like. Practically speaking it is of no importance to us if we lose or not the 20th decimal of the fraction, because this is not the element that caries the important information. To solve this problem a thorough analysis of the situation and results that can appear, the representation's precision, running time, and memory size must be done. As we have seen when representing numbers, the precision from representing short real numbers is not enough for many practical applications. The precision of long real numbers is more than sufficient but occupies double memory space.

## 2.4. The coprocessors internal architecture

The coprocessor has two distinct components:

- Numerical execution unit: does the arithmetical and transfer instructions common to the coprocessor, and has an internal execution unit and a block of registers;
- Control unit: extracts from memory the instructions and operands and executes the control instructions, has a logical block, pointer and control registers;

### 2.4.1. The numerical execution unit

From the user's point of view the most important component are the general block registers that are organized as an **internal stack**. All the registers (stack elements) have 80 bits. Each operation is addressed to the element that is on top of the stack. That's why the stack's elements are named ST (0), ST (1)… etc., where ST (0) is the top of the stack, ST (1) the next element, and so on. It represents an inconvenient in assembling language programming as we have to save the stack position for each value, and when inserting a new element all previous elements' stack position is incremented.

### 2.4.2. Control unit

#### 2.4.2.1. Control Word

The control register is a 16 bit register. The user can set the value of the register and thus access a series of the coprocessors "finer" mechanisms like the rounding method, etc.

As we can se the register is divided in two, that's because the first 8 most significant bits control the processor's working strategy the other 8 bits control the interrupts when an error occurs.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x | X | X | ip | rc | rc | pc | pc | x | x | pm | um | om | zm | dm | im |

X marks an unused bit
The others are:

**IC** – Infinite control- the way in which infinite numbers are treated

When division by 0 occurs the coprocessor puts infinite in the result. Mathematically speaking we have to ways of ending the number's axis: projective and affine. The difference between the two is that the latter knows two types of infinite (positive and negative). None of the two methods is better than the other. Bit coding:

0- projective
1- affine

**RC**- Rounding control

00-round to the closest element that can be represented
01-round downwards
10-round upwards
11-trunkation

**PC-** Precision Control

In some cases we do not want to work with the result in the internal precision even tough it is always represented as such. If we use previously written procedures designed for IEEE short format, error propagation with other precision is unpredictable. So we can force results to a given precision.

Values of the PC pair of bits

00 -24 bit-short real
01 -NA
10 -53 bit-long real precision
11 -63 bit-high precision

**M. Mask** – validates or invalidates the coprocessors interrupt. When an error occurs during a floating point operation, the coprocessor sends an interrupt to the processor. On IBM-PC the interrupt created is NMI, an unmasked processor interrupt

Values on MASK

0– validates an interrupt request
1- Invalidates an interrupt request

With the following bits we specify which exception (error) really calls the interrupt. This can be useful when we are not interested in a particular exception, or if we want to control the problem by reading the coprocessors state from the program. The following bits validate the interrupt on 0 and invalidate it on 1.

**PM** Precision Mask -signals rounding interrupt
**UM** Underflow Mask - signals underflow interrupt
**OM** Overflow Mask - signals overflow interrupt
**ZM** Zero Divide Mask - signals divide by zero interrupt
**DM** De-normalized OM - signals de-normalizing interrupt
**IM** Invalid Operation Mask - signals invalid operation interrupt

### 2.4.2.2. Status Word

It's a 16 bit register. Its content is set according to the last executed operation. We can obtain form it vital information for the user. Two of the first most significant bits correspond with the zero carry bits from the I8086 processor. Because we can load any value on the last 8 STATUS bits using the instruction SAHF, after a coprocessor's operation we can read and use these bits with simple control instructions.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | C3 | St | St | St | St | C2 | C1 | C0 | Es | X | Pe | Ue | Ze | De | Ie |

In this case we have a 16 bit register split in two. The first 8 bits tell us if the operation done by the coprocessor generates an interrupt, if it does then we are told what exception generated the interrupt. The first 8 most significant bits represent the coprocessors arithmetical status.
**IR** Interrupt request- If its value is one then it means that the coprocessor requests an interrupt. In this case one of the following six bits will also be 1, and will show what class the error belongs to.
**IE** Invalid Operation Error-invalid operation
**DE** Not normalized Operand Error – error caused by an operand that was not normalized or the result can not be normalized
**ZE** Zero Divide Error – division by zero caused error
**OE** Overflow Error - overflow caused error
**UE** Underflow Error - underflow caused error
**PE** Precision Error – precision error, the result was rounded
**C0** (conditional bit 0)
**C1** (1st conditional bit)
**C2** (2nd conditional 2)
**C3** (3rd conditional 3)
**SP –** the three bits point to the top of the stack. The value 000 signals an empty stack and the first element to be loaded will be the element 0 from the stack, while the value 111 signals that the stack is full.
**B Busy** – tells us if the coprocessor is working or not. It is active on 1, so in this case we are not allowed to send other instructions to the coprocessor. This bit allows us to synchronize our program with the coprocessor.

The meaning of C0, C1, C2, and C3 is displayed in the following table. As we can see these bits are not easy to define. Practically we only need to check one or two bits, which is fairly simple if we rely on the remarks made at the start of this chapter: we can use the fact that the zero STATUS and Carry flags have the same meaning.

| C3 | C2 | C1 | C0 | Sign | Meaning |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | + | Not normalized |
| 0 | 0 | 0 | 1 | + | Not a number |
| 0 | 0 | 1 | 0 | - | Not normalized |
| 0 | 0 | 1 | 1 | - | Not a number |
| 0 | 1 | 0 | 0 | + | Normalized positive |
| 0 | 1 | 0 | 1 | + | Positive infinite |
| 0 | 1 | 1 | 0 | - | normalized |
| 0 | 1 | 1 | 1 | - | Negative infinite |

| | | | | | |
|---|---|---|---|---|---|
| 1<br>1<br>1<br>1 | 0<br>0<br>0<br>0 | 0<br>0<br>1<br>1 | 0<br>1<br>0<br>1 | +<br>empty<br>-<br>empty | Zero ( positive)<br>.....<br>zero ( negative)<br>..... |
| 1<br>1<br>1<br>1 | 1<br>1<br>1<br>1 | 0<br>0<br>1<br>1 | 0<br>1<br>0<br>1 | +<br>empty<br>-<br>empty | Invalid, Not normalized<br>.....<br>Invalid, Not normalized<br>..... |

The value of C3 is 0 if the result of the operation (normalized of not normalized) is not 0. If the bit's value is 1 then the result is either 0 or invalid or the corresponding stack element is empty. It can be said that C3 greatly resembles the zero STATUS bit. The value of C2 depends on C3. If the value of C3 in 0, C2 points out the normalized result on 1, and a non-normalized result on 0. If the value of C33 is 1 (the result is 0 or the element is empty) then it's the opposite, C2 set on 1 points to a invalid number, while 0 points to zero.

As we have seen in the previous table C1 points to the number's sign. If the result is negative then C1 is 1 if not C1 is 0. C0 tells us if the result is valid or not. If C) is 0 then there are no severe errors, but if C0 is 1 the result is invalid (the result is infinite or other special value).

If other types of operations are being done, like comparisons then their meaning changes:

| C3 | C2 | C1 | C1 | Meaning |
|---|---|---|---|---|
| 0<br>0<br>1<br>1 | 0<br>0<br>0<br>0 | X<br>X<br>X<br>X | 0<br>1<br>0<br>1 | ST (0) >"op"<br>ST (0) <"op"<br>ST (0) ="op"<br>ST (0) and "op" can not be compared |

If C3 is 0 then the result of the comparison will be read from C0. If C3 is 1 then C2 will point out if the numbers are equal or not. Or ST (0) can not be compared (void or infinite).

After the partial remainder operation these bits have other meaning. In this case C0, C1, C2 (from top to bottom in this order) will keep the one, two or three bits of the result when the division has a remainder. The value of C2 is 0 after the creation of a partial remainder and 1 in case of error. The meaning is (about the creation of a partial remainder will discus at the instruction description):

| Divider/Divided | C3 | C1 | C0 |
|---|---|---|---|
| Divide>Divided /2 | X | X | Bit 0 |
| Divider>Divided /4 | X | Bit 1 | Bit 0 |
| Divider<=Divided /4 | Bit 2 | Bit 1 | Bit 0 |

X signifies that in that case the bits keep their previous value. For example, if the number to be divided is smaller than the number to be divided by then the remainder is equal to the divider and the result is 0. In this case C3 and C1 keep their value, and C0 will be 0 to signal that the result will be zero. And if the divider is smaller than half the number to be divided, but larger then the quarter of the number to be divided then the result will be 2 or 3; we have this number in C1, C0 and C3 keeps his previous value.

#### 2.4.2.3. Tag Word

| 15 14 | 13 12 | 11 10 | 9 8 | 7 6 | 5 4 | 3 2 | 0 1 |
|---|---|---|---|---|---|---|---|
| R7 | R6 | R5 | R4 | R3 | R2 | R1 | R0 |

The bit pairings describe from top to bottom the stack registers 0, 1... Etc.
A bit pairing can have the following meanings:

| **Value** | **Stack registers bits meaning** |
|---|---|
| 00 | The corresponding element contains a valid data |
| 01 | The corresponding element contains zero |
| 10 | The corresponding element contains a special value |
| 11 | The corresponding element is empty |

The term special value means that the stack element contains infinite of for some reason the result of an operation is invalid.

#### 2.4.2.4. Instruction Pointer

The instruction pointer contains the physical address and the coprocessor's last operation code.

This register helps us when we are writing interrupt routines for catching errors that appear during coprocessor operations. In these cases it is useful to know the operation's code and the physical address (the internal memory location where it can be found). We can easily see its importance when we realize that the program does not have to wait for the coprocessor, while this is working the processor can execute other tasks. We must consider the coprocessor only when its result is needed or when we need to do another operation.

Now, (because our program has passed the instruction that caused an error) we can't find out which was the last instruction sent to the coprocessor. The space reserved for the instruction code is larger then the true size of the code so the code appears as aligned to the right.

**2.4.2.5. Data Pointer**

The data pointer contains the physical address of the last external data utilized in the last floating point operation.

Like the previous this register uses interrupts for catching errors that may appear during the execution of coprocessor instructions. In this case we must know the data's physical address (external for the coprocessor) used by the last instruction that caused the error.

For the user that that write programs in assembling language it is important to know the coprocessor's condition, more exactly the coprocessor's environment that defines the working conditions at a particular time. This environment is defined by known elements.

**2.5. Coprocessor's environment**

The internal registers that the user can be access. The mathematical coprocessor has a set of registers of 14 bits organized like this:

| | |
|---|---|
| | COMAND REGISTER |
| | STATUS REGISTER |
| | STACK REGISTER |
| A19-16 | INSTRUCTIONS REGISTER (A15-0)<br>\| 0 \| OPERATION CODE ( bits 10-0) |
| A19-16 | DATA REGISTER (A15-0)<br>\| 0. . .. . . . . . . . . . .. . . . . ... . . . . . .. . . 0 |

**2.6. Coprocessor's instruction set**

The coprocessor can be programmed in assembly language using the ESC instruction. This instruction sends a 6 bit operation code on the data bus and if necessary also sends on the data bus a memory address. The coprocessor sees and takes the instruction sent to him and executes it. There are two ways of a new synchronization between the coprocessor and the processor, both attributed to the processor:

- the processor tests the coprocessor's status
- the processor calls a WAIT instruction

### 2.6.1. Data transfer instructions

Data transfer instructions ensure the exchange of data between the computers memory and the coprocessor's stack. They can be classified like this:

#### 2.6.1.1. LOAD Instructions

**FILD adr** - Loads on the stack the integer variable located at address „adr". The variable stored in memory of type (DB, DW, and DD) is converted to the coprocessors internal format at load.

**FLD adr** - Loads on the stack the real variable (long or short) located at address „adr". The variable stored in memory of type (DD, DQ, and DT) is converted to the coprocessors internal format at load

**FBLD adr** - Loads on the stack the packed decimal variable located at address „adr". The variable stored in memory of type (DT) is converted to the coprocessors internal format at load.

#### 2.6.1.2. STORE Instructions

**FIST adr** -Stores at the address „adr" the value located on the stack (ST (0)) as a number. The stored value can be only an integer represented on one byte or a short integer, corresponding to the data stored at address „adr" (DW or DD). The stack pointer remains unchanged after the data is stored. The conversion is done during the store process.

**FISTP adr** - Stores at the address „adr" the value located on the stack (ST (0)) as an integer number. The stored value can be any integer (byte integer, short integer, long integer, corresponding to the data stored at address „adr" (DW, DD or DQ). The conversion is done during the store process. The instruction changes the stack: ST (0) is deleted by decrementing the stack pointer.

**FST adr** - Stores at the address „adr" the value located on the stack (ST (0)) as an integer number. The stored value can be an integer short integer or in double precision, corresponding to the data stored at address „adr" (DD or DQ). The stack pointer and the data on the stack remain unchanged after the data is stored. The conversion is done during the store process.

**FSTP adr** - Stores at the address „adr" the value located on the stack (ST (0)) as a floating point number. The value can be a short real with double or extended precision, corresponding to the data stored at address „adr" (DD, DQ or DT). The conversion is done during the store process from the internal format. The instruction changes the stack: ST (0) is deleted by decrementing the stack pointer.

**FBSTP adr** - Stores at the address „adr" the value located on the stack (ST (0)) as a packed decimal number (defined at "adr" with DT). The stack pointer is decremented. The conversion is done during the store process from the internal format.

**NOTE**: You must remember that any type of data can be loaded. When we try to store we have two possibilities: If the data from the stack is to be eliminated we can use the 7 data types. But if we want to keep the stored value on the stack only the 4 basic typed are allowed.

### 2.6.2. Internal data transfer instructions

**FLD** ST (i) Put value from ST (i) on the stack. Thus the value from ST (i) will be found twice: in ST (0) and ST (i+1).

**FST** ST (i) The value from ST (0) is copied in the stack's "i" element. The old ST (i) is lost.

**FSTP** ST (i) The value from ST (0) is copied in the stack's "i" element. The old ST (i) is lost. ST (0) is eliminated by decrementing the stack pointer.

**FXCH** ST (i) swap between ST (0) and ST (i).

### 2.6.3. Constants loading instruction

**FLDZ** Loads 0 at the top of the stack

**FLD1** Loads 1.0 at the top of the stack

**FLDPI** Loads"pi" at the top of the stack

**FLDL2T** Loads log (10) at the top of the stack

**FLDL2E** Loads log (e) at the top of the stack

**FLDLG2** Loads log (2) at the top of the stack

**FLDLN2** Loads ln (2) at the top of the stack

## 2.7. Arithmetical and comparison instructions

Arithmetical instructions usually have 2 operands. One of them is always at the top of the stack, and usually this is also the place where the result is be generated. Basic operations can be executed without restrictions with the following methods
-the instruction's mnemonic is being written without an operand. In this case the operands are ST (0) and ST (1).
-the instruction's mnemonic and the operand. The operand can be a memory address or a stack element (ST (1) is also usable but it's not very use full)
-the instruction's mnemonic is written and 2 operands: the first is a stack element (not ST (0)) and the second is ST (0). In this case the result will be put in the place of the first operand and ST (0) will be deleted from the stack. (In the instruction's mnemonic the letter P appears).

### 2.7.1. Arithmetical instructions

**FADD** ST (0) ←ST (0) +ST (1) FADD **op** ST (0) ←ST (0) +"op" from memory or stack.
Floating point operation.
**FADD op** ST (0) ←ST (0) +"op" from memory or stack.
Integer operation.
**FADD ST (i), ST (0)** ST (i) ←ST (i) +ST (0); ST (0) eliminated
**FSUB** ST (0) ←ST (0) -ST (1) FSUB **op** ST (0) ←ST (0) –„op" from memory or stack.
Floating point operation.
**FISUB op** ST (0) ←ST (0) -"op" from memory or stack.
Integer operation.
**FSUB ST (i), ST (0)** ST (i) ←ST (i) -ST (0); ST (0) eliminated
**FSUBR ST (i)** ST (i) ←ST (i) -ST (0) ; FSUB ST (i) opposite instruction
**FMUL** ST (0) ←ST (0) XST (1)
**FMUL op** ST (0) ←ST (0) x "op" from memory or stack.

Floating point operation.
**FIMUL op** ST (0) ←ST (0) x ”op” from memory or stack.
Integer operation.
**FMULP ST (i), ST (0)** ST (i) ←ST (i) x ST (0); ST (0) eliminated
**FDIV** ST (0) ←ST (0): ST (1) FDIV **op** ST (0) ←ST (0):”op” from memory or stack.
Floating point operation.
**FDIV op** ST (0) ←ST (0):”op” from memory or stack.
Integer operation.
**FDIVP ST (i), ST (0)** ST (i) ←ST (i): ST (0); ST (0) eliminated
**FDIVR ST (i)** ST (i) ←ST (i): ST (0); FDIV ST (i) opposite instruction.

### 2.8.2. Number comparison instructions

| | |
|---|---|
| **FCOM** | The values from ST (0) and ST (1) are compared and C3, C2 and C0 indictors are set |
| **FCOM op** | The values ST from (0) and memory or stack (floating point variable) are compared and C3, C2 and C0 indictors are set |
| **FICOM op** | The values ST from (0) and memory or stack (floating point variable) are compared and C3, C2 and C0 indictors are set. |
| **FCOMP** | The values from ST (0) and ST (1) are compared and C3, C2 and C0 indictors are set. ST (0) is deleted from the stack |
| **FICOMP op** | The values ST from (0) and memory or stack (floating point variable) are compared and C3, C2 and C0 indictors are set ST (0) is deleted from the stack. |
| **FCOMPP** | The values from ST (0) and ST (1) are compared and C3, C2 and C0 indictors are set. ST (0) and ST (1) are deleted from the stack. |
| **FTST** | C3, C2 and C0 indicators are set according to the result of the comparison between the values ST (0) with 0. |
| **FXAM** | The condition bits are set according to the value of ST (0).No comparison is being done. |

**Remarks:**

- FCOMP and FCOMPP allow us the easiest ways of eliminating one or two elements from the stack

- FXAM is used for analyzing the special conditions that result where computing errors occur.

## 2.8. Floating point functions

**FSQRT** Square root – ST (0) 's square root is put in ST (0). The number has to be positive, or the result will not make sense.

**FSCALE** 2's power. Puts in ST (0) the ST (0)' s value multiplied with $2^{ST(1)}$:
ST (0) ←ST (0) *2**ST (1) ST (1) has to be an integer, and ST (0)'s absolute value ha to be smaller then 2**15.

**FPREM** Partial remainder. ST (0) is divided by ST (1) and stored in ST () ←ST (0) -ST (1) * (the biggest lower integer for ST (0) /ST (1)).

**FRMDINT** Round. ST (0) is replaced with ST (0) rounded. The rounding method is set in the command line.

**FXTRACT** The value stored in ST (0) is split into Characteristic (in ST (0)) and mantissa (in ST (1)).

**FABS** ST (0) is replaced with its absolute value.

**FCHS** ST (0) sign is changed.

**FPTAN** Partial tangent. The tangent of the angle contained in ST (0) is determined as a ST (1) /ST (0) fraction. The initial value of the angle contained in ST (0) must be between 0 and "pi"/4.

**FPATAN** Partial Arctangent. The arctangent of the value contained in ST (0) is determined as a ST (1) /ST (0) fraction. The initial value contained in ST (0) must be positive, while ST (1) must be larger ST (0).

**F2XM1** 2's power. ST (0) will be replaced by 2**ST (0) -1. Initially ST (0) must be between 0 and 0.5.

**FYL2X** Logarithm. ST (0) ←ST (1) *LOG2 (ST (0)). ST (0) has to be a positive number, while ST (1) has to be a finite number.

**FYL2XP1** Logarithm. ST (0) ←ST (1) *LOG2 (ST (0) +1). ST (0) has to be a positive number lower then 0.3, while ST (1) has to be a finite number.

**Remarks:**

- Sin and Cosine can be determined by using the tangent
- Any exponent can be computed using F2XM1
- For determining the exponent ST (0) $^{ST(1)}$ it is recommended to use the functions FYL2X and then F2XM1!

## 2.9. Control Instructions

Control instructions have the task of coordinating the microprocessors actions. Usually they have no arithmetic meaning, but some of them do influence drastically the actions of the coprocessor because they save or load the coprocessor's state, more exactly all of its work registers. Among these registers is the stack thus, these can be regarded as gigantic load and save instructions.

**FINIT** Initialization- the coprocessor is brought in an initial status known as software reset". After the FINIT instruction all of the coprocessors registers will be in their initial status and the stack will be empty.

**FENI** Interrupt accept- if the coprocessor needs to generate an interrupt when an error is detected, besides the correct positioning of the command register it needs to explicitly accept the interrupt.

**FDISI** Interrupt ignores- this instruction ignores all interrupts regardless of the command register's bits; to accept a new interrupt the instruction FENI must be called.

**FLDCW adr** The command register is loaded from the memory location indicated by **adr**

**FSTCW adr** The command register is saved in a word located at the memory location indicated by **adr**

**FSTSW adr** The status register is saved in a word located at the memory location indicated by **adr**.

**FCLEX** The bits that define the exceptions are erased- the instruction erases the corresponding bits regardless of the status of the error bits

**FSTENV adr** Environment save- the coprocessor's internal registers are saved in a memory location starting at **adr** that has a size of 14 bytes.

**FLDENV adr** Environment load- the coprocessor's internal registers are loaded from a memory location starting at **adr** that has a size of 14 bytes.

**FSAVE adr** Status save- the coprocessor's internal registers and its stack are saved in a memory location starting at **adr** that has a size of 94 bytes.

**FRSTOR adr** Status load- the coprocessor's internal registers and its stack are loaded from a memory location starting at **adr** that has a size of 94 bytes.

**FINCSTP** Stack indicators increment- after the instruction's action it is incremented with a stack indicator; thus the element that became ST (0) remains unchanged (fact pointed out by the stack description register's bits).

**FDECSTP** Stack indicators decrement- after the stack indicator is decremented by one; thus the stack's elements remain unchanged (fact pointed out by the stack description register's bits).

**FFREE ST (i)** the "i" ranked element from the stack is eliminated. The operation does not influence the stack pointer.

**FNOP** No operation executed.

**FWAIT** Waits for the current action to finish (similar to the 8086 WAIT instruction)

A simple program that uses the mathematical coprocessor

```
; Program that determines the area of a circle with the radius R
; And volume of a sphere with radius R

DATE      SEGMENT   PARA `DATA`    ;SEGMENT

RAZA      DQ        8. 567
ARIE      DQ        ?              ; RESERVE SPACE
VOLUM     DQ        ?              ; RESULTS
PATRU     DD        4. 0
TREI      DD        3. 0
DATE      ENDS

COD       SEGMENT   PARA `CODE`
.8087
CALCUL    PROC FAR  ;
          ASSUME CS:COD, DS: DATE

          PUSH DS              ; PREPARE
          XOR  AX, AX          ; STACK FOR
          PUSH      AX         ; DOS RETURN
          MOV  AX, DATE        ; LOADING DS
          MOV  DS, AX          ; WITH DATA SEGMENT
```

```
            FINIT           ;COPROCESOR INITIALIZATION
            FLD  RAZA       ;LOAD RAZA ON COPROC STACK
            FMUL RAZA       ;COMPUTE R x R
            FLDPI           ;LOAD PI TO COPROC STACK
            FMUL            ;COMPUTE R x R x PI
            FSTP ARIE       ;SAVING RESULT
            FWAIT           ;SYNCHRONIZATION


            LEA  SI, VOLUM ; VOLUM ADDRSS IN SI
            FINIT           ;COPROCESOR INITIALIZATION
            FLD  RAZA            ; COMPUTATION
            FMUL RAZA            ; R x R
            FMUL RAZA            ; R x R x R
            FLDPI                    ; LOAD PI
            FMUL                 ; MULTYPLY WITH PI
            FMUL PATRU           ; MULTIPLY WITH FOUR
            FDIV TREI                ; DIVISION BY 3
            FSTP QWORD PTR [SI]      ; SAVING RESULT
            FWAIT                ; SYNCHRONIZATION

            RET
CALCUL      ENDP                     ; END PROCEDURE
COD         ENDS                 ; END CODE SEGMENT

            END  CALCUL              ; PROGRAM END
```

## 3. Lab tasks

1. Run the given example

2. Write a program that determines $\sqrt[3]{2}$. Hint: Use the instructions F2XM1 and FYL2X.