Module 4 learning unit 10:

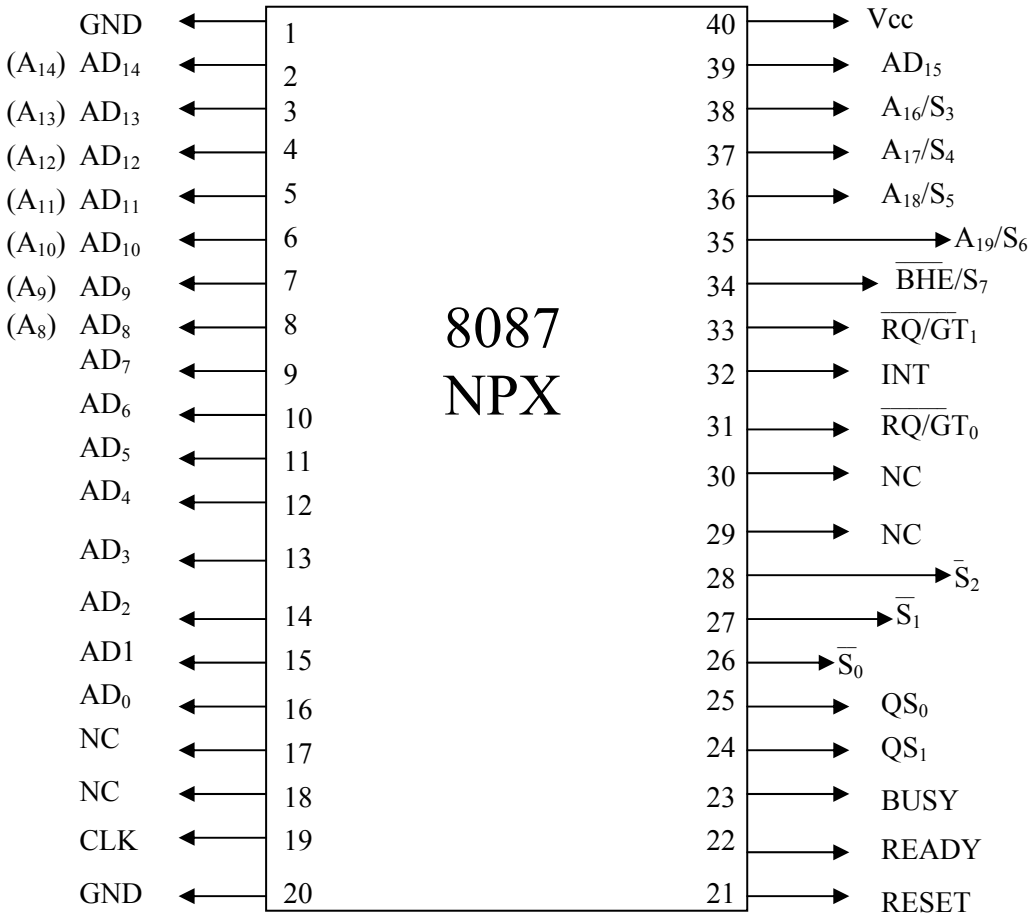
Contents ❖Architecture of 8087
❖Data types
❖Interfacing
❖Instructions and programming

Overview➢Each processor in the 80 x 86 families has a corresponding coprocessor with which it is compatible.
➢Math Coprocessor is known as NPX,NDP, FUP.
Numeric processor extension (NPX),
Numeric data processor (NDP),
Floating point unit (FUP).

Compatible Processor and Coprocessor

Processors
1. 8086 & 8088
2. 80286
3. 80386DX
4. 80386SX
5. 80486DX
6. 80486SX

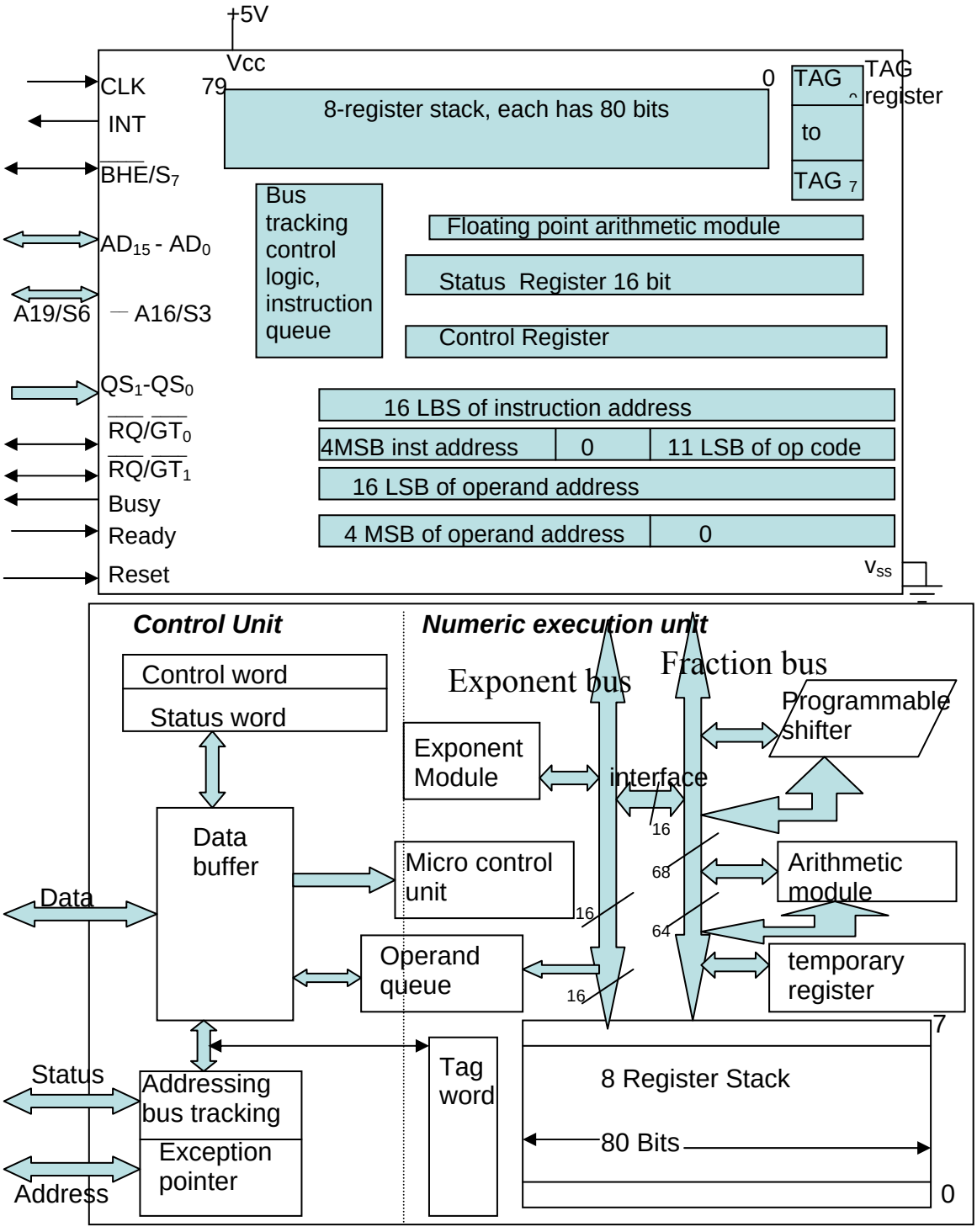
Coprocessors
1. 8087
2. 80287, 80287XL
3. 80287, 80387DX
4. 80387SX
5. It is Inbuilt
6. 80487SX



Pin Diagram of 8087

Architecture of 8087❖Control Unit

❖Execution Unit



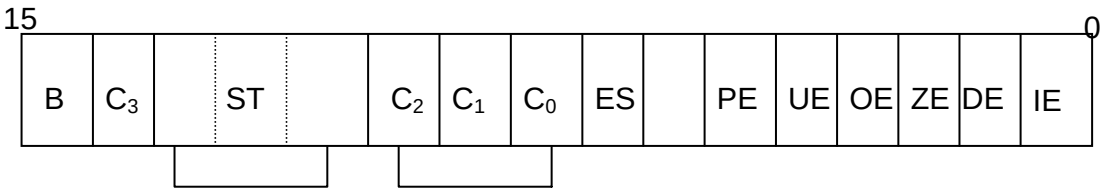
Control Unit➢Control unit: To synchronize the operation of the coprocessor and the processor.

➢This unit has a Control word and Status word and Data Buffer

➢If instruction is an *ESC*ape (coprocessor) instruction, the coprocessor executes it, if not the microprocessor executes.

➢Status register reflects the over all operation of the coprocessor.

Status Register



- C3-C0 Condition code bits
- TOP Top-of-stack (ST)
- ES Error summary
- PE Precision error
- UE Under flow error
- OE Overflow error
- ZE Zero error
- DE Denormalized error
- IE Invalid error
- B Busy bit

➢B-Busy bit indicates that coprocessor is busy executing a task. Busy can be tested by examining the status or by using the FWAIT instruction. Newer coprocessor automatically synchronize with the microprocessor, so busy flag need not be tested before performing additional coprocessor tasks.

➢C_3-C_0 Condition code bits indicates conditions about the coprocessor.

➢TOP- Top of the stack (ST) bit indicates the current register address as the top of the stack.

➢ES-Error summary bit is set if any unmasked error bit (PE, UE, OE, ZE, DE, or IE) is set. In the 8087 the error summary is also caused a coprocessor interrupt.

➢PE- Precision error indicates that the result or operand executes selected precision.

➢UE-Under flow error indicates the result is too large to be represent with the current precision selected by the control word.

➢OE-Over flow error indicates a result that is too large to be represented. If this error is masked, the coprocessor generates infinity for an overflow error.

➢ZE-A Zero error indicates the divisor was zero while the dividend is a non-infinity or non-zero number.

➢DE-Denormalized error indicates at least one of the operand is denormalized.

➢IE-Invalid error indicates a stack overflow or underflow, indeterminate from (0/0,0,-0, etc) or the use of a NAN as an operand. This flag indicates error such as those produced by taking the square root of a negative number.

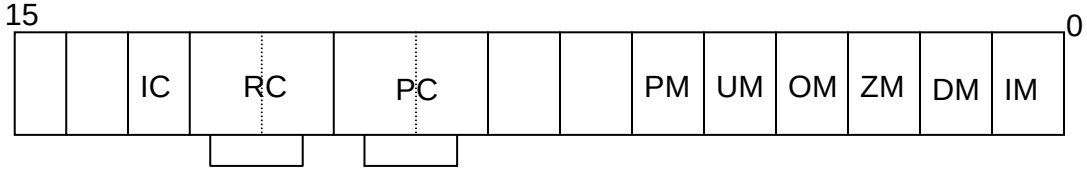
Control Register

➢Control register selects precision, rounding control, infinity control.

➢It also masks an unmasks the exception bits that correspond to the rightmost Six bits of status register.

➢Instruction FLDCW is used to load the value into the control register.

Control Register



•IC Infinity control
•RC Rounding control
•PC Precision control
•PM Precision control
•UM Underflow mask
•OM Overflow mask
•ZM Division by zero mask
•DM Denormalized operand mask
•IM Invalid operand mask

➢IC –Infinity control selects either affine or projective infinity. Affine allows positive and negative infinity, while projective assumes infinity is unsigned.

INFINITY CONTROL

0 = Projective
1 = Affine

➢RC –Rounding control determines the type of rounding.

ROUNDING CONTROL

00=Round to nearest or even
01=Round down towards minus infinity
10=Round up towards plus infinity
11=Chop or truncate towards zero

➢PC- Precision control sets the precision of he result as define in table

PRECISION CONTROL

00=Single precision (short)
01=Reserved
10=Double precision (long)
11=Extended precision (temporary)

➢Exception Masks – It Determines whether the error indicated by the exception affects the error bit in the status register. If a logic1 is placed in one of the exception control bits, corresponding status register bit is masked off.

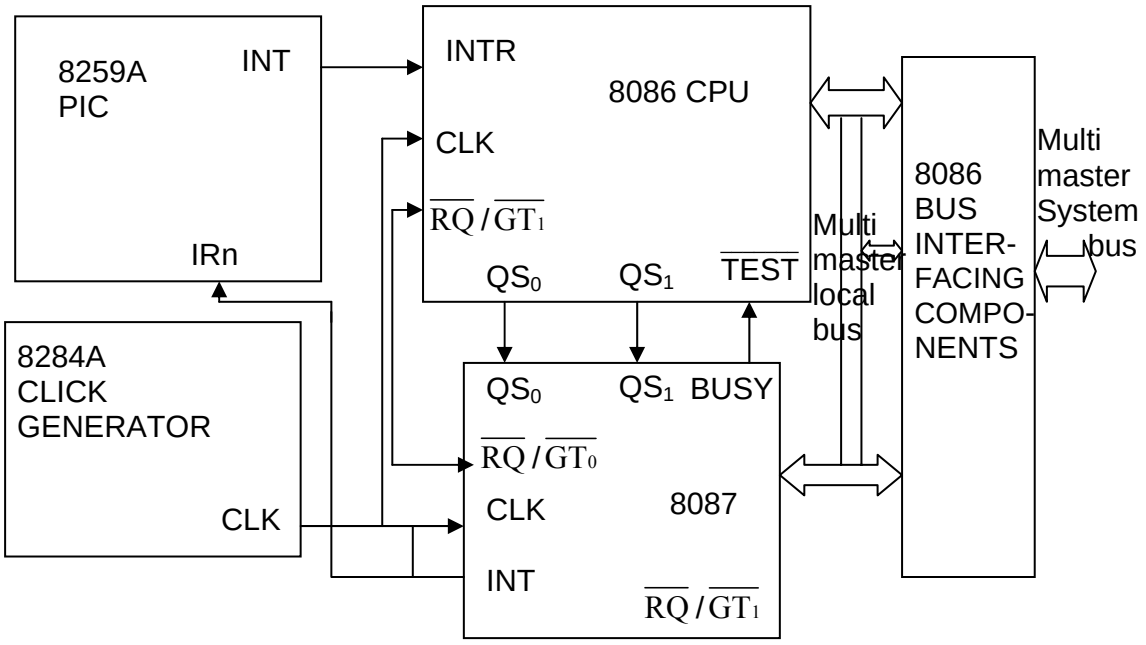
Numeric Execution Unit➢This performs all operations that access and manipulate the numeric data in the coprocessor's registers.

➢Numeric registers in NUE are 80 bits wide.

➢NUE is able to perform arithmetic, logical and transcendental operations as well as supply a small number of mathematical constants from its on-chip ROM.

➢Numeric data is routed into two parts ways a 64 bit mantissa bus and a 16 bit sign/exponent bus.

Circuit Connection for 8086 - 8087



- Multiplexed address-data bus lines are connected directly from the 8086 to 8087. The status lines and the queue status lines connected directly from 8086 to 8087.
- The $\overline{\text{Request}}$ / $\overline{\text{Grant}}$ signal RQ/GT0 of 8087 is connected to $\overline{RQ}/\overline{GT_1}$ of 8086.
- BUSY signal 8087 is connected to $\overline{\text{TEST}}$ pin of 8086.
- Interrupt output INT of the 8087 to NMI input of 8086. This intimates an error condition.

➢The main purpose of the circuitry between the INT output of 8087 and the NMI input is to make sure that an NMI signal is not present upon reset, to make it possible to mask NMI input and to make it possible for other devices to cause an NMI interrupt.
➢BHE pin is connected to the system BHE line to enable the upper bank of memory.
➢The RQ/GT_1 input is available so that another coprocessor such as 8089 I/O processor can be connected and function in parallel with the 8087.
➢One type of Cooperation between the two processors that you need to know about it is how the 8087 transfers data between memory and its internal registers.
➢When 8086 reads an 8087 instruction that needs data from memory or wants to send data to memory, the 8086 sends out the memory address code in the instruction and sends out the appropriate memory read or memory write signal to transfer a word of data.
➢In the case of memory read, the addressed word will be kept on the data bus by the memory. The 8087 then simply reads the word of data bus. The 8086 ignores this word .If the 8087 only needs this one word of data, it can then go on and executes its instruction.
➢Some 8087 instructions need to read in or write out up to 80-bit word. For these cases 8086 outputs the address of the first data word on the address bus and outputs the appropriate control signal.
➢The 8087 reads the data word on the data bus by memory or writes a data word to memory on the data bus. The 8087 grabs the 20-bit physical address that was output by the 8086.To transfer additional words it needs to/from memory, the 8087 then takes over the buses from 8086.
➢To take over the bus, the 8087 sends out a low-going pulse on

$\overline{RQ}/\overline{GT_0}$ pin. The 8086 responds to this by sending another low going pulse back to the $\overline{RQ}/\overline{GT_0}$ pin of 8087 and by floating its buses.

- The 8087 then increments the address it grabbed during the first transfer and outputs the incremented address on the address bus. When the 8087 output a memory read or memory write signal, another data word will be transferred to or from the 8087.
- The 8087 continues the process until it has transferred all the data words required by the instruction to/from memory.
- When the 8087 is using the buses for its data transfer, it sends another low-going pulse out on its $\overline{RQ}/\overline{GT_0}$ pin to 8086 to know it can have the buses back again.

The next type of the synchronization between the host processor and the coprocessor is that required to make sure the 8086 hast does not attempt to execute the next instruction before the 8087 has completed an instruction.

- Taking one situation, in the case where the 8086 needs the data produced by the execution of an 8087 instruction to carry out its next instruction.
- In the instruction sequence for example the 8087 must complete the ***FSTSW STATUS*** instruction before the 8086 will have the data it needs to execute the ***MOV AX , STATUS*** instruction.
- Without some mechanism to make the 8086 wait until the 8087 completes the FSTSW instruction, the 8086 will go on and execute the ***MOV AX , STATUS*** with erroneous data .
- We solve this problem by connecting the 8087 BUSY output to the TEST pin of the 8086 and putting on the WAIT instruction in the program.
- While 8087 is executing an instruction it asserts its BUSY pin high. When it is finished with an instruction, the 8087 will drop its BUSY pin low. Since the BUSY pin from 8087 is connected to the TEST pin 8086 the processor can check its pin of 8087 whether it finished it instruction or not.
- You place the 8086 WAIT instruction in your program after the 8087 FSTSW instruction .When 8086 executes the WAIT instruction it enters an internal loop where it repeatedly checks the logic level on the TEST input. The 8086 will stay in this loop until it finds the TEST input asserted low, indicating the 8087 has completed its instruction. The 8086 will then exit the internal loop, fetch and execute the next instruction.

Example

```
FSTSW       STATUS      ;copy  8087 status word to memory
MOV         AX, STATUS  ;copy status word to AX to check
                        ; bits
```

(a)

- In this set of instructions we are not using WAIT instruction. Due to this the flow of execution of command will takes place continuously even though the previous instruction had not finished it's completion of its work .so we may lost data .

```
FSTSW       STATUS      ;copy 8087 status word to memory
FWAIT                   ;wait for 8087 to finish before-
                        ; doing next 8086 instruction
```

```
MOV          AX,STATUS  ;copy status word to AX to check
                        ; bits
                        ( b )
```

➢In this code we are adding up of FWAIT instruction so that it will stop the execution of the command until the above instruction is finishes it's work .so that you are not loosing data and after that you will allow to continue the execution of instructions.
➢Another case where you need synchronization of the processor and the coprocessor is the case where a program has several 8087 instructions in sequence.
➢ The 8087 are executed only one instruction at a time so you have to make sure that 8087 has completed one instruction before you allow the 8086 to fetch the next 8087 instruction from memory.
➢Here again you use the BUSY-TEST connection and the $\overline{\text{FWAIT}}$ instruction to solve the problem. If you are hand coding, you can just put the 8086 WAIT(FWAIT) instruction after each instruction to make sure that instruction is completed before going on to next.
➢If you are using the assembler which accepts 8087 mnemonics, the assembler will automatically insert the 8-bit code for the WAIT instruction ,10011011 binary (9BH), as the first byte of the code for 8087 instruction.

INTERFACING

➢Multiplexed address-data bus lines are connected directly from the 8086 to 8087.
➢The status lines and the queue status lines connected directly from 8086 to 8087.

➢The Request/Grant signal $\overline{\text{RQ}}/\overline{\text{GT0}}$ of 8087 is connected to

$\overline{\text{RQ}}/\overline{\text{GT1}}$ of 8086.

➢BUSY signal 8087 is connected to $\overline{\text{TEST}}$ pin of 8086.
➢Interrupt output INT of the 8087 to NMI input of 8086. This intimates an error condition.
➢A WAIT instruction is passed to keep looking at its $\overline{\text{TEST}}$ pin, until it finds pin Low to indicates that the 8087 has completed the computation.
➢SYNCHRONIZATION must be established between the processor and coprocessor in two situations.
a) The execution of an ESC instruction that require the participation of the NUE must not be initiated if the NUE has not completed the execution of the previous instruction.
b) When a processor instruction accesses a memory location that is an operand of a previous coprocessor instruction .In this case CPU must synchronize with NPX to ensure that it has completed its instruction.
Processor WAIT instruction is provided.

Exception Handling

➢The 8087 detects six different types of exception conditions that occur during instruction execution. These will cause an interrupt if unmasked and interrupts are enabled.
1)INVALID OPERATION
2)OVERFLOW
3)ZERO DIVISOR

4)UNDERFLOW
5)DENORMALIZED OPERAND
6)INEXACT RESULT

Module 4 learning unit 11:

Data Types

➢Internally, all data operands are converted to the 80-bit temporary real format.
We have 3 types.
•Integer data type
•Packed BCD data type
•Real data type

Coprocessor data typesInteger Data Type

Packed BCD
Real data type

Example

➢Converting a decimal number into a Floating-point number.
1) Converting the decimal number into binary form.
2) Normalize the binary number
3) Calculate the biased exponent.
4) Store the number in the floating-point format.

Example

Step	Result
1)	100.25
2)	$1100100.01 = 1.10010001 * 2^6$
3)	110+01111111=10000101
4)	Sign = 0
	Exponent =10000101
	Significand = 10010001000000000000000

•In step 3 the biased exponent is the exponent a 2^6 or 110,plus a bias of 01111111(7FH) ,single precision no use 7F and double precision no use 3FFFH.
•IN step 4 the information found in prior step is combined to form the floating point no.

INSTRUCTION SET

➢The 8087 instruction mnemonics begins with the letter F which stands for Floating point and distinguishes from 8086.
➢These are grouped into Four functional groups.
➢The 8087 detects an error condition usually called an exception when it executing an instruction it will set the bit in its Status register.

Types

I. DATA TRANSFER INSTRUCTIONS.
II. ARITHMETIC INSTRUCTIONS.
III. COMPARE INSTRUCTIONS.
IV. TRANSCENDENTAL INSTRUCTIONS.
(Trigonometric and Exponential)

Data Transfers Instructions**REAL TRANSFER**

FLD	Load real
FST	Store real

FSTP Store real and pop
FXCH Exchange registers

➢ **INTEGER TRANSFER**

FILD Load integer
FIST Store integer
FISTP Store integer and pop

➢**PACKED DECIMAL TRANSFER(BCD)**

FBLD Load BCD
FBSTP Store BCD and pop

Example

➢**FLD Source**- Decrements the stack pointer by one and copies a real number from a stack element or memory location to the new ST.

```
•FLD   ST(3)                ;Copies ST(3) to ST.
•FLD   LONG_REAL[BX]        ;Number from memory
                            ;copied to ST.
```

➢**FLD Destination-** Copies ST to a specified stack position or to a specified memory location .

```
•FST        ST(2)           ;Copies ST to ST(2),and
                            ;increment stack pointer.
•FST        SHORT_REAL[BX]  ;Copy ST to a memory at a
                            ;SHORT_REAL[BX]
```

➢**FXCH Destination** – Exchange the contents of ST with the contents of a specified stack element.

```
•FXCH       ST(5)  ;Swap ST and ST(5)
```

➢**FILD Source** – Integer load. Convert integer number from memory to temporary-real format and push on 8087 stack.

```
•FILD  DWORD PTR[BX] ;Short integer from memory at [BX].
```

➢**FIST Destination-** Integer store. Convert number from ST to integer and copy to memory.

```
•FIST  LONG_INT    ;ST to memory locations named LONG_INT.
```

➢**FISTP Destination**-Integer store and pop. Identical to FIST except that stack pointer is incremented after copy.

➢**FBLD Source-** Convert BCD number from memory to temporary- real format and push on top of 8087 stack.

Arithmetic Instructions.❖Four basic arithmetic functions:
Addition, Subtraction, Multiplication, and
Division.

➢**Addition**

FADD Add real
FADDP Add real and pop
FIADD Add integer

➢**Subtraction**

FSUB Subtract real
FSUBP Subtract real and pop
FISUB Subtract integer
FSUBR Subtract real reversed

FSUBRP Subtract real and pop
FISUBR Subtract integer reversed

➢**Multiplication**

FMUL Multiply real
FMULP Multiply real and pop
FIMUL Multiply integer

➢**Advanced**

FABS Absolute value
FCHS Change sign
FPREM Partial remainder
FPRNDINT Round to integer
FSCALE Scale
FSQRT Square root
FXTRACT Extract exponent and mantissa.

Example

➢**FADD** – Add real from specified source to specified destination Source can be a stack or memory location. Destination must be a stack element. If no source or destination is specified, then ST is added to ST(1) and stack pointer is incremented so that the result of addition is at ST.

```
•FADD    ST(3), ST    ;Add ST to ST(3), result in ST(3)
•FADD    ST,ST(4)     ;Add ST(4) to ST, result in ST.
•FADD                 ;ST + ST(1), pop stack result at ST
•FADDP   ST(1)        ;Add ST(1) to ST. Increment stack
                      ;pointer so ST(1) become ST.
•FIADD   Car_Sold     ;Integer number from memory + ST
```

➢**FSUB -** Subtract the real number at the specified source from the real number at the specified destination and put the result in the specified destination.

```
•FSUB ST(2), ST   ;ST(2)=ST(2) – ST.
•FSUB Rate        ;ST=ST – real no from memory.
•FSUB             ;ST=( ST(1) – ST)
```

➢**FSUBP** - Subtract ST from specified stack element and put result in specified stack element .Then increment the pointer by one.

```
•FSUBP   ST(1)    ;ST(1)-ST. ST(1) becomes new ST
```

➢**FISUB** – Integer from memory subtracted from ST, result in ST.

```
•FISUB   Cars_Sold   ;ST becomes ST – integer from memory
```

Compare Instructions.➢**Comparison**

FCOM Compare real
FCOMP Compare real and pop
FCOMPP Compare real and pop twice
FICOM Compare integer
FICOMP Compare integer and pop
FTST Test ST against +0.0
FXAM Examine ST

Transcendental Instruction.➢**Transcendental**

FPTAN Partial tangent
FPATAN Partial arctangent

F2XM1	$2^x - 1$
FYL2X	$Y \log_2 X$
FYL2XP1	$Y \log_2(X+1)$

Example

➢**FPTAN –** Compute the values for a ratio of Y/X for an angle in ST. The angle must be in radians, and the angle must be in the range of 0 < angle < π/4.➢**F2XM1 –** Compute $Y=2^x-1$ for an X value in ST. The result Y replaces X in ST. X must be in the range $0 \le X \le 0.5$.

➢**FYL2X** - Calculate $Y(LOG_2X)$.X must be in the range of $0 < X < \infty$ any Y must be in the range $-\infty<Y<+\infty$.

➢**FYL2XP1 –** Compute the function $Y(LOG_2(X+1))$.This instruction is almost identical to FYL2X except that it gives more accurate results when compute log of a number very close to one.

Constant Instructions.➢**Load Constant Instruction**

FLDZ	Load +0.0	
FLDI	Load+1.0	
FLDPI	Load π	**FLDL2T** Load $\log_2 10$
FLDL2E	Load $\log_2 e$	
FLDLG2	Load $\log_{10} 2$	
FLDLN2	Load $\log_e 2$	

ALGORITHM**To calculate x to the power of y**

•Load base, power.
•Compute (y)*($\log_2$ x)
•Separate integer (i),fraction(f) of a real number
•Divide fraction (f) by 2
•Compute ($_2{}^{f/2}$) * ($_2{}^{f/2}$)
•$x^y = (_2{}^x) * (_2{}^y)$

Program:Program to calculate x to the power of y

```
        .MODEL SMALL
        .DATA
        x               Dq      4.567   ;Base
        y               Dq      2.759   ;Power
        temp    DD
        temp1   DD
        temp2   DD              ;final  real result
        tempint         DD
        tempint1        DD      ;final integer result
        two             DW
        diff            DD
        trunc_cw        DW      0fffh
                        .STACK 100h
                        .CODE
start:          mov ax, @DATA       ;init data segment
                        mov ds, ax
```

```
load:       fld y              ;load the power
            fld x              ;load the base
comput:     fyl2x              ;compute (y * log2(x))
            fst temp           ;save the temp result
trunc:      fldcw  trunc_cw    ;set truncation command
            frndint
            fld  temp          ;load real number of fyl2x
            fist  tempint      ;save integer after truncation
            fld  temp          ;load the real number
getfrac:    fisub  tempint      ;subtract the integer
            fst   diff         ;store the fraction
fracby2:    fidiv two          ;divide the fraction by 2
twopwrx:    f2xm1               ;calculate the 2 to the power fraction
            fst temp1          ;minus 1 and save the result
            fld1               ;load1
            fadd               ;add 1 to the previous result
            fst temp1          ;save the result
sqfrac:     fmul st(0),st(0)    ;square the result as fraction
            fst temp1           ;was halved and save the result
            fild tempint        ;save the integer portion
            fxch                ;interchange the integer
                                ;and power of fraction.
scale:      fscale              ;scale the result in real  and
            ;integer
            fst temp2           ;in st(1) and store
            fist tempint1       ;save the final result in real and integer
over:       mov ax,4c00h        ;exit to dos
            int 21h
            end start
```