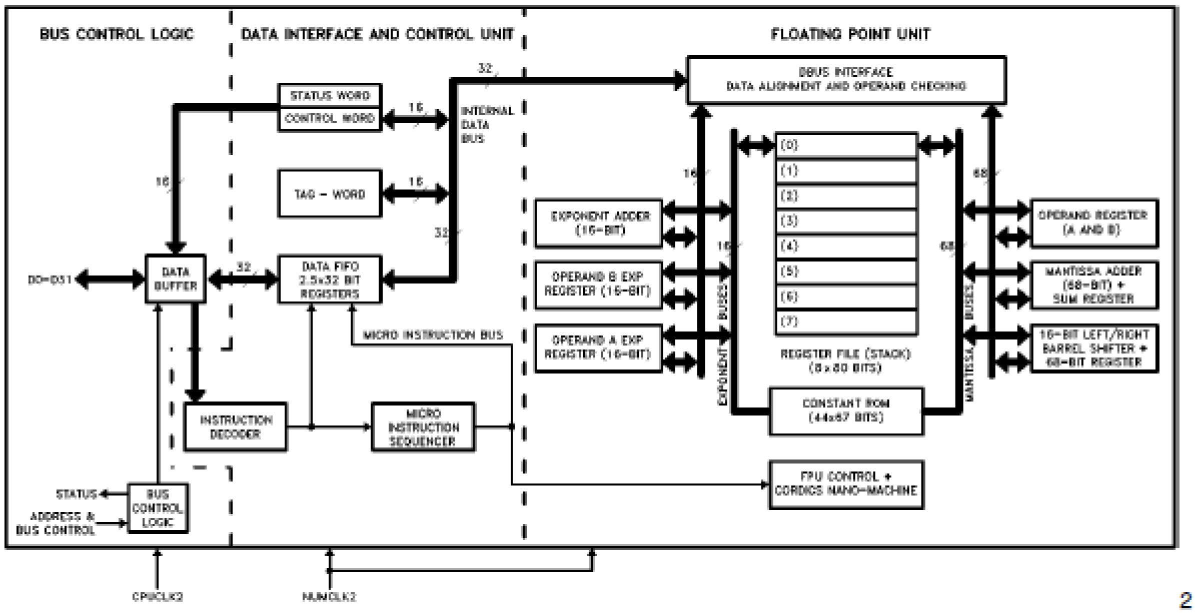
Shrey Singh
22B-CO-055
Batch C

Q1)


BUS CONTROL LOGIC
DATA INTERFACE AND CONTROL UNIT
FLOATING POINT UNIT
STATUS WORD
CONTROL WORD
TAG - WORD
INTERNAL DATA BUS
DATA FIFO 2.5x32 BIT REGISTERS
DATA BUFFER
D0-D31
MICRO INSTRUCTION BUS
INSTRUCTION DECODER
MICRO INSTRUCTION SEQUENCER
STATUS
ADDRESS & BUS CONTROL
BUS CONTROL LOGIC
CPUCLK2
NUMCLK2
DBUS INTERFACE DATA ALIGNMENT AND OPERAND CHECKING
EXPONENT ADDER (16-BIT)
OPERAND B EXP REGISTER (16-BIT)
OPERAND A EXP REGISTER (16-BIT)
EXPONENT BUSES
REGISTER FILE (STACK) (8 x 80 BITS)
CONSTANT ROM (44x67 BITS)
MANTISSA BUSES
OPERAND REGISTER (A AND B)
MANTISSA ADDER (68-BIT) + SUM REGISTER
16-BIT LEFT/RIGHT BARREL SHIFTER + 68-BIT REGISTER
FPU CONTROL + CORDICS NANO-MACHINE

2

Figure 0.1. Intel387™ DX Math CoProcessor Block Diagram

The Intel 80387 DX Math Coprocessor (MCP) is a numeric processor that adds high-speed numerics processing capabilities to the Intel386 DX Microprocessor system. It is designed to work seamlessly with the Intel386 DX CPU and can be used in both real-address mode and protected mode.

The MCP features include:

Seven data types: single-precision floating-point, double-precision floating-point, extended-precision real, 16-, 32-, and 64-bit integers, and 18-digit packed BCD.
Eight 80-bit data registers that can be accessed as a stack or as fixed registers.
A tag word that identifies the contents of each data register.
A status word that reflects the overall state of the MCP.
A control word that configures the MCP operating mode, including precision and rounding.
Six exception conditions that can occur during instruction execution: invalid operation, denormalized operand, zero divisor, overflow, underflow, and inexact result.
The MCP communicates with the Intel386 DX CPU through a set of instructions and interrupt handlers. Programmers can use these instructions to perform a variety of numeric operations, including addition, subtraction, multiplication, division, square root, trigonometric functions, and logarithmic functions.

Here are some of the key points

The MCP uses extended-precision real format internally for all calculations.

Operands are loaded into the MCP registers from memory in their native format (e.g., integer, single-precision floating-point) and then converted to extended-precision real format.
The MCP register set can be accessed as a stack or as a fixed register set.
The tag word helps the MCP to optimize performance and stack handling by keeping track of whether each register is empty or contains a valid value.
The status word provides information about the overall state of the MCP, including the condition codes, the busy bit, the top-of-stack pointer, and the exception flags.
The control word allows programmers to configure the MCP operating mode, including precision and rounding control.
The MCP can generate six exceptions: invalid operation, denormalized operand, zero divisor, overflow, underflow, and inexact result. These exceptions are reported to the CPU through interrupt handlers.
The Intel 80387 DX MCP is compatible with the 8087 and 80287 math coprocessors, but there are some differences in functionality.

Q2); Assembly Language Program for 80387DX to compute x = a * (b + c) / d

```
FLD b ; Load b onto the stack
FLD c ; Load c onto the stack
FADD ; Add b and c (c + b is now on top of the stack)
FLD a ; Load a onto the stack
FMUL ; Multiply a by (b + c) ((b + c) * a is now on top of the stack)
FLD d ; Load d onto the stack
FDiv ; Divide ((b + c) * a) by d (x is now on top of the stack)
FSTP x ; Store the result (x) in memory location x

; End of program
```

Q3)**a. Memory Mapped I/O vs. I/O Mapped I/O**

Memory Mapped I/O (MMIO) and **I/O Mapped I/O (IO Mapped I/O)** are two techniques for interfacing the Central Processing Unit (CPU) with peripheral devices in a computer system. Here's a breakdown of their similarities and differences:

Similarities:

- Both are methods for the CPU to interact with external devices.
- Both involve assigning unique identifiers (addresses) to devices.

Differences:

Feature	Memory Mapped I/O	I/O Mapped I/O
Address Space	Same as memory	Separate from memory
Access method	Standard memory access instructions (e.g., MOV)	Special I/O instructions (e.g., IN, OUT)
Programming	Simpler, uses existing memory instructions	More complex, requires dedicated I/O instructions
Efficiency	Faster - utilizes existing memory access mechanisms	Slower - requires additional logic for I/O handling
Memory Usage	Less efficient - consumes memory space for I/O devices	More efficient - dedicated I/O address space
Error checking	Can leverage memory protection features	Requires separate error checking mechanisms

In essence:

- **MMIO** treats I/O devices as memory locations, allowing for a unified address space and simpler programming. However, it can consume memory space and might be less efficient.
- **IO Mapped I/O** has a separate address space for I/O devices, offering more efficient memory usage but requiring dedicated instructions and error handling.

b. String and Non-string based operations

String operations and **Non-string based operations** are two categories of instructions used in computer programs to manipulate data.

String operations deal with sequences of characters, typically represented as text. Here are some examples:

- Concatenation (joining strings)
- Searching for a substring within a string
- Replacing characters within a string
- Extracting specific parts of a string
- Comparing strings for equality

These operations often involve specialized instructions or libraries designed for efficient string manipulation.

Non-string based operations work on data other than strings, such as numbers, logical values, or custom data structures. Examples include:

- Arithmetic operations (addition, subtraction, multiplication, division)
- Logical operations (AND, OR, NOT)
- Bitwise operations (shifting, masking)
- Comparison operations (equal to, greater than, less than)
- Data structure manipulation (adding/removing elements from lists, trees, etc.)

These operations typically rely on basic instructions provided by the CPU architecture.

Here's a table summarizing the key points:

Feature	String Operations	Non-string Operations
Data type	Sequences of characters	Numbers, logical values, custom data structures

Instructions	Specialized instructions or libraries	Basic CPU instructions
Focus	Manipulating text	Performing calculations, comparisons, data management
Examples	Concatenation, searching, replacing	Arithmetic, logical, bitwise, comparisons

Q4)

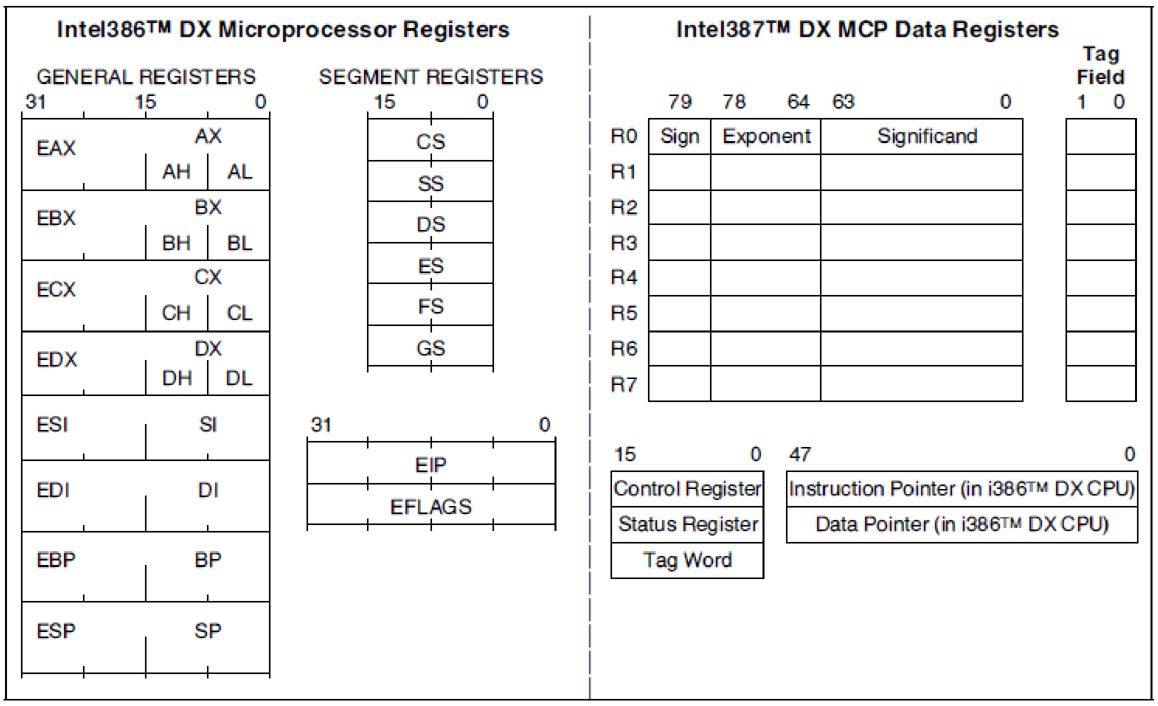


Figure 1.1. Intel386™ DX Microprocessor and Intel387™ DX Math Coprocessor Register Set

When an MCP is present in a system, programmers may use these registers in addition to the registers normally available on the Intel386 DX CPU.

2.3.1 DATA REGISTERS
Intel387 DX MCP computations use the MCP's data registers. These eight 80-bit registers provide the equivalent capacity of twenty 32-bit registers. Each of the eight data registers in the MCP is 80 bits wide and is divided into "fields" corresponding to the MCPs extended-precision real data type.
The Intel387 DX MCP register set can be accessed either as a stack, with instructions operating on the

top one or two stack elements, or as a fixed register set, with instructions operating on explicitly designated registers. The TOP field in the status word identifies the current top-of-stack register. A ‘‘push’’ operation decrements TOP by one and loads a value into the new top register. A ‘‘pop’’ operation stores the value from the current top register and then increments TOP by one. Like the Intel386 DX Microprocessor stacks in memory, the MCP register stack grows ‘‘down’’ toward lower-addressed registers. Instructions may address the data registers either implicitly or explicitly.

Many instructions operate on
the register at the TOP of the stack. These instructions implicitly address the register at which TOP

points. Other instructions allow the programmer to explicitly specify which register to user. This explicit register addressing is also relative to TOP.

2.3.2 TAG WORD

The tag word marks the content of each numeric data register, as Figure 2.1 shows. Each two-bit tag represents one of the eight numerics registers. The principal function of the tag word is to optimize the MCPs performance and stack handling by making it

possible to distinguish between empty and nonempty register locations. It also enables exception handlers to check the contents of a stack location without the need to perform complex decoding of the

actual data.

2.3.3 STATUS WORD

The 16-bit status word (in the status register) shown in Figure 2.2 reflects the overall state of the MCP. It may be read and inspected by CPU code.

Bit 15, the B-bit (busy bit) is included for 8087 compatibility only. It reflects the contents of the ES bit

(bit 7 of the status word), not the status of the BUSYÝ output of the Intel387 DX MCP.

Bits 13 –11 (TOP) point to the Intel387 DX MCP register that is the current top-of-stack.

The four numeric condition code bits (C3–C0) are
similar to the flags in a CPU; instructions that per-
form arithmetic operations update these bits to re-
flect the outcome. The effects of these instructions

on the condition code are summarized in Tables 2.2
through 2.5.
Bit 7 is the error summary (ES) status bit. This bit is
set if any unmasked exception bit is set; it is clear
otherwise. If this bit is set, the ERRORÝ signal is
asserted.

Bit 6 is the stack flag (SF). This bit is used to distin-
guish invalid operations due to stack overflow or un-
derflow from other kinds of invalid operations. When

SF is set, bit 9 (C1) distinguishes between stack
overflow (C1 e 1) and underflow (C1 e 0).
Figure 2.2 shows the six exception flags in bits 5 –0
of the status word. Bits 5 –0 are set to indicate that
the MCP has detected an exception while executing
an instruction. A later section entitled ‘‘Exception
Handling’’ explains how they are set and used.

Note that when a new value is loaded into the status
word by the FLDENV or FRSTOR instruction, the
value of ES (bit 7) and its reflection in the B-bit (bit
15) are not derived from the values loaded from
memory but rather are dependent upon the values of
the exception flags (bits 5 –0) in the status word and
their corresponding masks in the control word. If ES
is set in such a case, the ERRORÝ output of the
MCP is activated immediately.

Q5)The 80387 math coprocessor utilizes a register stack for numerical operations. Here's how to achieve the desired actions using 80387 instructions:

a. Add contents of register 3 to the top of the stack:

- FLD R3 : Load the value from register 3 onto the top of the stack (ST(0)).

b. Subtract contents of register 2 from the top of the stack and store the result in register 2:

1. **Pop, subtract, store:**

   - FSTP R2 : Pop the top value from the stack (ST(0)) into register 2 and decrement the stack pointer.
   - SUB R2, #value : Subtract the immediate value (replace #value with actual value) from R2 (performed in the main CPU, not the 80387).
2. **Load top of stack, subtract from register 2, store result:**

   - FLD R2 : Push the value from register 2 onto the stack (ST(0)).
   - FSUB ST(1), ST(0) : Subtract the value at ST(0) (original value in R2) from ST(1) (which now holds the value from R2 again).
   - FSTP R2 : Pop the top value from the stack (the result) into register 2.

Note:

- In both instructions, replace `#value` with the actual value you want to add/subtract if it's not 0.
- The 80387 operates on the top elements of the stack (ST(0) and ST(1) in these cases).
- Option b.2 performs both pushing and subtraction within the 80387 for a more streamlined approach.

c.The FSAVE instruction in the 80387 math coprocessor is used to save the state of the x87 Floating-Point Unit (FPU) on the memory stack. This includes:

Register values: It saves the contents of all eight x87 registers (ST(0) to ST(7)) along with their corresponding status flags (tags) into memory.

Control word: It saves the current value of the x87 Control Word (CW) which controls various aspects of the FPU operation like rounding mode, precision, and exception handling.

Status word: It saves the current value of the x87 Status Word (SW) which reflects the state of the FPU like busy flag, exception flags, and stack pointer.

Tag word: The tag word is an internal register that keeps track of the format (empty, valid, or special) of each x87 register. FSAVE doesn't directly save the tag word itself, but the saved status of each register implicitly reflects the tag information.

Q6)Here's an explanation of the following 80387 instructions:

a. FINIT

The `FINIT` instruction is used to initialize the 80387 math coprocessor. It performs several key actions:

- **Clears registers:** It sets all eight x87 registers (ST(0) to ST(7)) to empty (NaN - Not a Number).
- **Sets control word:** It initializes the x87 Control Word (CW) to a default state. This typically sets the rounding mode to round-to-nearest and precision to single-precision (32-bit) floating-point format.
- **Sets status word:** It initializes the x87 Status Word (SW) to a defined state, clearing any exception flags and setting the stack pointer (TOP) to point to ST(0) (making it the top of the stack).

In essence, FINIT prepares the 80387 for use by establishing a clean and default operating state. It's generally recommended to execute `FINIT` at the beginning of any program that uses floating-point math operations to ensure a predictable starting point for calculations.

b. F2XM1

The `F2XM1` instruction calculates `2^x - 1`, where x is the value on the top of the x87 stack (ST(0)). It performs the following:

1. **Loads exponent:** It isolates the exponent part of the value in ST(0) and subtracts 1 from it.
2. **Moves mantissa:** It copies the mantissa (significand) part of the value in ST(0) to another register (ST(1)).
3. **Clears ST(0):** It sets the value in ST(0) to 1.0.
4. **Exponentiation:** It performs 2 raised to the power of the modified exponent (calculated in step 1) and stores the result in ST(0).
5. **Multiplication:** It multiplies the result in ST(0) by the mantissa saved in ST(1) and stores the final answer in ST(0).

In simpler terms, F2XM1 calculates one less than two raised to the power of the top-of-stack value.

c. FSQRT

The `FSQRT` instruction calculates the square root of the value on the top of the x87 stack (ST(0)). It follows these steps:

1. **SQRT operation:** It computes the square root of the value in ST(0) and stores the result in the same register (ST(0)).
2. **Exception handling:** It checks for potential exceptions during the square root operation, such as a negative operand (no real square root exists). If an exception occurs, it sets the appropriate exception flag in the x87 Status Word (SW).

FSQRT efficiently calculates the square root of a floating-point number using the 80387 hardware. Remember to check for exceptions (e.g., using the `FPU_ERROR` instruction) after using FSQRT to handle potential errors gracefully.

Q7)

Feature	FTST	FXAM
Focus	Sign and zero comparison	Detailed value examination
Status Flags Set	C0, C2, C3 for comparison result	Sign, exponent, tag information
Modifies Stack (ST(0))	No	No
Use Cases	Conditional branching based on sign/zero	Detailed value analysis (NaNs, denormals)

Q8)DX MCP supports and presents the format for each type. Operands are stored in memory with the least significant digit at the lowest memory address. Programs retrieve these values by generating the lowest Address.
For maximum system performance, all operands should start at physical-memory addresses evenly divisible by four (doubleword boundaries); operands may begin at any other addresses, but will require extra memory cycles to access the entire operand.

Internally, the Intel387 DX MCP holds all numbers in the extended-precision real format. Instructions that load operands from memory automatically convert operands represented in memory as 16-, 32-, or 64-bit integers, 32- or 64-bit floating-point numbers, or 18-digit packed BCD numbers into extended-precision real format.

Instructions that store operands in memory perform the inverse type conversion.

Q8) Discussed in Q4