Contents

FLOATING POINT BINARY REPRESENTATION

Why Floating-Point Exists at All

Computers are great at **integers**.

They are terrible at:

- Fractions
- Very large numbers
- Very small numbers

Floating-point representation exists to solve this problem.

Instead of storing numbers directly, the CPU stores them in a **scientific-notation-like form**.

That form looks like this:

```
value = sign × significand × base^exponent
```

For x86:

- Base = 2 (binary)

The Three Core Components (Mental Model First)

Every floating-point number is broken into **three parts**:

1) Sign

- Tells whether the number is positive or negative
- 0 → positive
- 1 → negative

Example: -1.23154×10^5

Sign = negative

2) Significand (Mantissa)

This is the **main value** of the number.

Think of it as:

"The meaningful digits"

In: **1.23154×10^5**

The significand is: **1.23154**

In binary, the significand is stored in a **normalized form**, not as raw decimal digits.

3) Exponent

The exponent tells:

"How far to shift the decimal point"

It scales the significand up or down.

Example: $1.23154 \times 10^5 = 123154$

Exponent controls:

- Very large values
- Very small values

Without exponents, floating-point would be useless.

Scientific Notation → Binary Reality

Humans write: **-1.23154×10^5**

Computers write: **$-1.001110\ldots \times 2^{16}$ (example)**

The idea is the same:

- One main value
- Scaled by a power
- Signed

Only the **base changes**:

- Humans → base 10
- Computers → base 2

Why Floating-Point Is Not Exact ⚠️

Here's a critical truth:

Most decimal numbers cannot be represented exactly in binary.

Example: **0.1 (decimal)**

has **no exact binary representation**.

This leads to:

- Rounding errors
- Precision loss
- "Why is 0.1 + 0.2 ≠ 0.3?"

This is not a bug.
It's math.

IEEE 754: The Rulebook 📘

x86 processors follow: **IEEE 754 – Binary Floating-Point Arithmetic**

This standard defines:

- How floating-point numbers are stored
- How rounding works
- How special values behave
- How exceptions are handled

IEEE 754 exists so:

Floating-point behaves the same across systems

IEEE 754 Binary Formats (Overview)

IEEE 754 defines **three main binary formats**:

FORMAT	TOTAL BITS	COMMON NAME
Single precision	32 bits	float
Double precision	64 bits	double
Extended precision	80 bits	x87 FPU

For now, we focus on **single precision (32-bit)**.

Single-Precision Floating-Point Layout (32-bit)

A 32-bit floating-point number is split like this:

- Sign → 1 bit
- Exponent → 8 bits
- Fraction (significand) → 23 bits

📌 The leading 1 of the significand is **implicit** (not stored).

Normalized Representation (Why the Hidden Bit Exists)

Binary floating-point numbers are stored in **normalized form**: $1.xxxxx \times 2^{e}$

That leading 1 is always there (except special cases).

So instead of wasting a bit storing it:

- The CPU assumes it
- Gains one extra bit of precision

This is called:

The hidden (implicit) bit

Exponent Bias (Why Exponents Aren't Signed)

Exponents are stored using a **bias**.

For single precision:

- Bias = **127**

Stored exponent:

This allows:

- Positive exponents
- Negative exponents
- Without signed arithmetic

Special Values (Where Things Get Weird)

IEEE 754 defines special cases:

- Exponent = 0 → denormalized numbers
- Exponent = all 1s:
    - Fraction = 0 → ±Infinity
    - Fraction ≠ 0 → NaN (Not a Number)

These exist so:

- Division by zero doesn't crash
- Invalid operations are detectable

Why x86 Cares So Much About This

Floating-point representation affects:

- Performance
- Precision
- Security
- Stability

In x86:

- FPU uses 80-bit extended precision
- SSE often uses 32/64-bit precision
- Mixing them causes subtle bugs

This is why:

Floating-point bugs are some of the hardest bugs to find

Reverse Engineering Perspective 🔍

When reversing:

- Floating-point constants may look “wrong”
- Precision may change across instructions
- Comparisons may fail unexpectedly

Never assume:

Floating-point equality means equality

Always think:

Approximation

Summary (No Fluff)

Floating-point numbers:

- Are stored as sign + significand + exponent
- Use base-2 scientific notation
- Follow IEEE 754 rules
- Trade exactness for range

This trade-off is intentional.

Lock-In Questions

1. Why can’t most decimals be represented exactly in binary?
2. Why does IEEE 754 use a hidden bit?
3. Why does exponent bias exist?
4. Why can mixing FPU and SSE cause bugs?

FLOATING-POINT FORMATS (IEEE 754)

I. Single-Precision Floating-Point Format (32-bit)

Single precision uses **32 bits** to represent a real number.

These 32 bits are divided into **three fields**:

- **Sign** → 1 bit
- **Exponent** → 8 bits
- **Significand (Mantissa)** → 23 bits

Visually:

⚠ Important clarification
Bit layout and memory layout are different things.
The *bit fields* are defined logically as above,
but **x86 stores bytes in little-endian order in memory**.

II. The Sign Bit

The **sign bit** is the most significant bit (MSB).

- 0 → positive number
- 1 → negative number

Example:

- 0.0 → positive zero
- -0.0 → negative zero (yes, this exists in IEEE 754)

The sign bit does **nothing else** — it only sets the sign.

III. The Significand (Mantissa)

The **significand** holds the **actual value digits** of the number.

In single precision:

- Stored bits → **23**
- Real precision → **24 bits**

Why 24?

Because IEEE 754 uses a **hidden leading 1**.

All normalized numbers are stored as: **$1.xxxxx \times 2^e$**

The 1 is **not stored** — it is **assumed**.

This gives one extra bit of precision "for free".

IV. The Exponent

The **exponent** controls the **scale** of the number.

In IEEE 754:

- Exponent is stored using a **bias**
- Single-precision bias = **127**

Stored exponent: **stored = real_exponent + 127**

This avoids signed arithmetic and allows:

- Very large numbers
- Very small numbers

V. Positional Notation (Why This Works)

Floating-point is an extension of **positional notation**.

Decimal example: **123.154**

This expands to:

$(1 \times 10^{2}) +$

$(2 \times 10^{1}) +$

$(3 \times 10^{0}) +$

$(1 \times 10^{-1}) +$

$(5 \times 10^{-2}) +$

(4×10^{-3})

Each digit has:

- A value
- A position
- A weight

VI. Binary Positional Notation (Floating-Point Reality)

Binary floating-point works the **same way**, but with base-2.

Example structure: $\mathbf{1.101011 \times 2^{e}}$

Digits to the left:

- Positive powers of 2

Digits to the right:

- Negative powers of 2

The significand stores these binary digits.

VII. Why Floating-Point Is Approximate ⚠

Most decimal numbers: Cannot be represented exactly in binary

Example: **0.1 (decimal)**

Becomes: **0.0001100110011... (binary, infinite)**

So floating-point numbers are:

Approximations, not exact values

This is why:

- 0.1 + 0.2 ≠ 0.3
- Equality comparisons are dangerous

VIII. Why Single Precision Is Useful

Single precision:

- Uses less memory
- Is faster on many systems
- Is good enough for graphics, games, and many simulations

But:

- Limited precision
- More rounding error than double precision

DOUBLE & EXTENDED PRECISION FORMATS

IX. Double-Precision Floating-Point (64-bit)

Double precision uses **64 bits**:

- Sign → 1 bit
- Exponent → 11 bits
- Significand → 52 bits (53 with hidden bit)

Advantages:

- Much higher precision
- Larger range
- Fewer rounding errors

This is:

- The default double in C/C++
- The standard in scientific computing

X. Extended Double Precision (80-bit)

Extended precision uses **80 bits**.

This format is mainly used by: The **x87 FPU**

Layout:

- Sign → 1 bit
- Exponent → 15 bits
- Significand → **64 bits (explicit)**

Unlike single/double:

- The leading bit is **stored**, not hidden
- Precision is extremely high

This is why:

- x87 calculations sometimes differ from SSE results
- Floating-point bugs appear when mixing units

XI. Support and Practical Use

- **Single precision (32-bit)** → graphics, embedded, performance-critical code
- **Double precision (64-bit)** → scientific, financial, engineering work
- **Extended precision (80-bit)** → internal FPU math, high-accuracy computations

Extended precision:

- Is not consistently supported across platforms
- Is often avoided in portable code

XII. Summary

- Floating-point uses **sign + significand + exponent**
- IEEE 754 defines the rules
- Precision is traded for range
- Single ≠ Double ≠ Extended
- Binary floating-point is approximate by design

If you don't internalize this:

Floating-point bugs will feel like black magic.

If you do, they become predictable and debuggable.

Here is a table comparing double precision and double extended precision:

FEATURE	DOUBLE PRECISION	DOUBLE EXTENDED
Number of bits	64	80
Sign bit	1	1
Exponent bits	11	16
Significand bits	52	63
Approximate Range	2^{-1022} to 2^{1023}	2^{-16382} to 2^{16383}

Single-Precision Floating-Point (32-bit)

- A **single-precision floating-point number** uses **32 bits** in total.
- It is split into **three parts**.

Sign Bit (1 bit)

- Tells if the number is **positive or negative**
- 0 → positive
- 1 → negative

Exponent (8 bits)

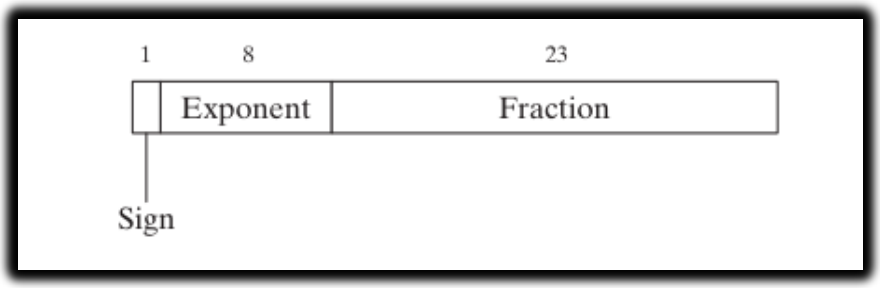
- Shows the **power of 2**
- It scales the number (makes it bigger or smaller)

Significand (23 bits)

- Also called the **mantissa**
- Stores the **fractional part** of the number
- Controls the **precision**

I. Bit Layout (Left to Right)

- **Sign bit** → first (most significant bit)
- **Exponent** → next 8 bits
- **Significand** → last 23 bits



Sign bit (1 bit)	Exponent (8 bits)	Significand (23 bits)	Padding (0 bits)

The padding bits are zeros that are not used in the floating-point representation.

Value	Approximate range	Explanation
Normalized	2^{-126} to 2^{127}	Range for typical non-zero values.
Denormalized	2^{-149} to 2^{-126}	Range for very small values close to 0.
Special values	NaN, +/-Infinity	Represents Not-a-Number and Infinity.

II. Normalized Numbers

- The **most common** floating-point numbers
- The **significand is between 1 and 2**
- **2 is not included**
- Used for most normal calculations

Conversion: Denormalized to Normalized

DENORMALIZED	NORMALIZED
1110.1	1.1101×2^{3}
.000101	1.01×2^{-4}
1010001.	1.010001×2^{6}

III. Denormalized Numbers

- The **significand is less than 1**
- Used to represent **very small numbers**
- Helpful for values **close to zero**

IV. Special Values

- **Special values** represent **NaN** and **Infinity**

V. NaN (Not a Number)

- Means the result is **undefined**
- Happens when a value **cannot be represented**
- Example: 0 ÷ 0

VI. Infinity (+∞ and −∞)

- Represents numbers that are **too large or too small**
- +∞ → positive infinity
- −∞ → negative infinity
- Often caused by **overflow or underflow**

Single-Precision Floating-Point (Overview)

- A **widely used** floating-point format
- Used in:
  - Scientific computing
  - Graphics
  - Gaming

I. Sign

- The **sign bit** shows if the number is positive or negative
- 0 → positive
- 1 → negative
- **Zero is treated as positive**

II. Significand

- The **significand** stores the **fractional part** of the number
- Made of digits **before and after** the decimal point

III. Exponents in the Significand

- Digits **left of the decimal point** → positive powers
- Digits **right of the decimal point** → negative powers

FLOATING-POINT SIGNIFICAND & PRECISION (CLEAN REBUILD)

I. Decimal Positional Notation (Baseline Idea)

Take the decimal number: **123.154**

This is just positional notation:

$$123.154 =$$

$$(1 \times 10^{2}) +$$

$$(2 \times 10^{1}) +$$

$$(3 \times 10^{0}) +$$

$$(1 \times 10^{-1}) +$$

$$(5 \times 10^{-2}) +$$

$$(4 \times 10^{-3})$$

The significand here is: **123.154**

And the **base** is 10.

This idea carries directly into binary floating-point — only the **base changes**.

II. Binary Positional Notation

Binary works exactly the same way, but with **base 2**.

Example: $\mathbf{11.1011_2}$

Expand it:

$$\mathbf{11.1011_2 =}$$

$$\mathbf{(1 \times 2^1) +}$$

$$\mathbf{(1 \times 2^0) +}$$

$$\mathbf{(1 \times 2^{-1}) +}$$

$$\mathbf{(0 \times 2^{-2}) +}$$

$$\mathbf{(1 \times 2^{-3}) +}$$

$$\mathbf{(1 \times 2^{-4})}$$

So far, this is **not floating-point yet** — this is just a binary real number.

III. Normalization (THIS IS KEY 🔑)

IEEE floating-point **never stores numbers like 11.1011**.

It **normalizes** them. Rule:

A normalized binary floating-point number has exactly one 1 to the left of the binary point.

So: 11.1011_2

Becomes: $\mathbf{1.11011_2 \times 2^1}$

Now we can identify the components:

- **Significand (mantissa)** → 1.11011
- **Exponent** → +1

This is the form IEEE 754 always uses.

IV. The Hidden Leading 1 (Why Precision Is Higher Than It Looks)

In IEEE 754 **normalized numbers**:

- The leading 1 **is not stored**
- It is **implicitly assumed**

So, when we say: "The first bit of the significand is always 1"

That's true **only for normalized numbers**.

Stored bits: **.11011**

Actual value: **1.11011**

This is why:

- Single precision has **23 stored bits**
- But **24 bits of precision**
- Double precision has **52 stored bits**
- But **53 bits of precision**

V. Exponent = Scaling, Not Changing the Value

Floating-point numbers can represent the **same value** using different exponent–significand pairs.

But ⚠ our earlier example mixed base-10 notation with base-2 meaning, so let's fix that.

Correct binary-based explanation - These represent **different values**, not the same:

$$\mathbf{1.23456789 \times 2^{10}}$$

$$\mathbf{1.23456789 \times 2^{9}}$$

$$\mathbf{1.23456789 \times 2^{8}}$$

Each exponent change multiplies or divides by **2**, not 10.

The **significand stays the same**, the **scale changes**.

That's how floating-point covers:

- Very small numbers
- Very large numbers

VI. Precision of the Significand

Precision = number of bits available to store the significand.

IEEE formats:

- **Single precision**
  - 23 stored bits
  - 24 bits of precision (hidden 1)
- **Double precision**
  - 52 stored bits
  - 53 bits of precision

This means:

A double can distinguish between about
2^{53} different values within a range

VII. Why Not All Real Numbers Can Be Represented ⚠

This is unavoidable.

- Real numbers are **infinite**
- Floating-point representations are **finite**

So, some numbers must be approximated. Example: **0.1_{10}**

In binary: **0.00011001100110011… (repeating forever)**

It **never terminates**.

So, the number is **rounded**.

VIII. Precision Limit Example

If a number requires **more bits than the format allows**, it cannot be represented exactly.

In double precision:

- Max significand precision = **53 bits**

Any binary value needing **54 or more bits**:

- Must be rounded
- Loses exactness

This is **not a bug**.
This is **mathematics colliding with hardware limits**.

IX. Summary 🧠

- Floating-point = **sign × significand × 2^exponent**
- Numbers are **normalized**
- Leading 1 is **hidden**
- Precision is **finite**
- Many decimals are **approximations**
- Equality comparisons are dangerous
- Understanding this saves **years of confusion**

Exponent (Floating-Point Numbers)

The **exponent** shows how many times the **significand is multiplied by 2**

It controls the **size** of the number

Example:

- 1.1011×2^5
- The significand 1.1011 is multiplied by **2 five times**

EXPONENT (E)	BIASED (E + 127)	BINARY
+5	132	10000100
0	127	01111111
-10	117	01110101
+127	254	11111110
-126	1	00000001
-1	126	01111110

I. Biased Exponent

- The exponent is stored using a **bias**
- This keeps the stored value **positive**

II. Single-Precision Details

- Exponent size: **8 bits**
- Bias value: **127**
- Stored as an **unsigned integer**

III. Exponent Ranges

- **Biased exponent range:** 1 to 254
- **Actual exponent range:** -126 to +127
- This range prevents **overflow** for very small values

Calculating the Biased Exponent

Biased exponent = actual exponent + 127

Example:

- 1.1011×2^5
- Actual exponent = 5
- Biased exponent = 127 + 5 = 132

Translating Binary Floating-Point to Fractions

The table below shows examples of how binary floating-point numbers can be translated into base-10 fractions.

- The **first column** lists the binary floating-point number.
- The **second column** shows the equivalent base-10 fraction.
- The **third column** gives the decimal value of that fraction.

BINARY FLOATING-POINT	BASE-10 FRACTION
`11.11`	$3\ \frac{3}{4}$
`101.0011`	$5\ \frac{3}{16}$
`1101.100101`	$13\ \frac{37}{64}$
`0.00101`	$\frac{5}{32}$
`1.011`	$1\ \frac{3}{8}$
`0.000...001`	$\frac{1}{8388608}$

I. Below is a quick explanation of each row in the table:

Row 1:

- Binary floating-point number: **11.11**
- Base-10 fraction: **3 3/4**
- Decimal value: **3.75**

Row 2:

- Binary floating-point number: **101.0011**
- Base-10 fraction: **5 3/16**
- Decimal value: **5.1875**

Row 3:

- Binary floating-point number: **1101.100101**
- Base-10 fraction: **13 37/64**
- Decimal value: **13.58125**

Row 4:

- Binary floating-point number: **0.00101**
- Base-10 fraction: **5/32**
- Decimal value: **0.15625**

Row 5:

- Binary floating-point number: **1.011**
- Base-10 fraction: **1 3/8**
- Decimal value: **1.375**

Row 6:

- Binary floating-point number: **0.000000000000000000001**
- Base-10 fraction: **1/8,388,608**
- Decimal value: **0.0000011920928955078125**

II. How to Translate a Binary Floating-Point Number

To convert a binary floating-point number into a base-10 fraction, follow these steps:

- Convert the **significand** to a decimal value.
- Multiply the significand by **2 raised to the power of the exponent**.
- Check the **sign bit**: if it's 1, the result is negative; if it's 0, the result is positive.

III. Example: Converting 11.11

Let's walk through the binary floating-point number **11.11** step by step:

IV. Understanding the Exponent

To find the value of a floating-point number, simply multiply the significand by **2 raised to the exponent**.

For example:
1.1011×2^3

Here, the exponent is **3**, which means the significand **1.1011** is multiplied by 2 three times.

The exponent tells us how much to scale the significand. Since this is a binary number, the base is always **2**.

In this case, the exponent was chosen so the final value falls between **1 and 2**.

- A smaller exponent would make the value less than 1.
- A larger exponent would make the value greater than 2.

V. Why Floating-Point Numbers Matter

Floating-point numbers are powerful because they let us represent both extremely small and extremely large values.

The exponent makes this possible by scaling the significand up or down as needed—giving us flexibility without losing precision.

Binary mapping to decimal mapping

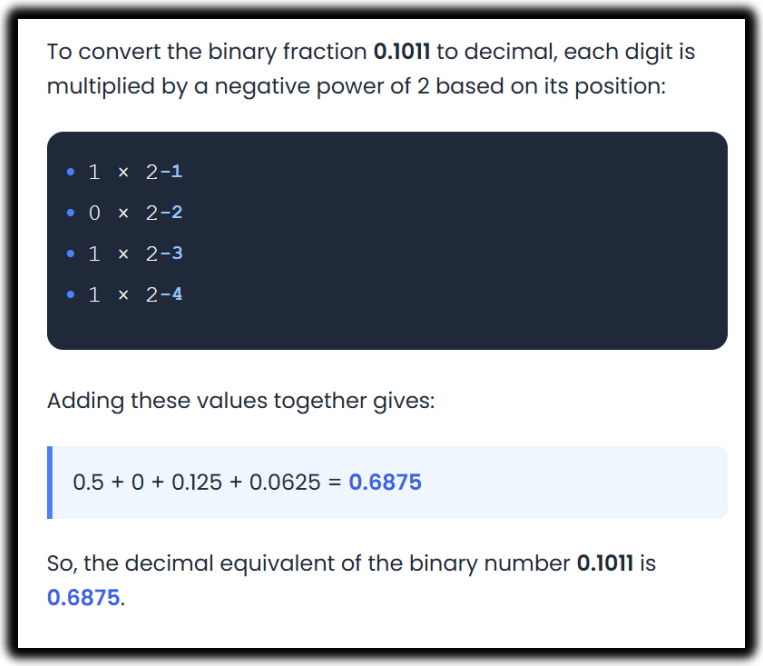
Let's discuss another table:

Binary to Decimal Mapping

BINARY	DECIMAL FRACTION	DECIMAL VALUE
.1	$\frac{1}{2}$	.5
.01	$\frac{1}{4}$	.25
.001	$\frac{1}{8}$	.125
.0001	$\frac{1}{16}$	.0625
.00001	$\frac{1}{32}$	.03125

To convert from **binary to decimal**, we multiply each digit in the binary number by its corresponding power of two, then add all the results together.

For example, to convert the binary fraction **0.1011** to decimal, each digit is multiplied by a negative power of 2 based on its position:



To convert from **decimal to binary**, we repeatedly divide the decimal number by 2 and record the remainder at each step.

The remainders, read in reverse order, form the binary representation of the number. For example, to convert the decimal number **2.0761** to binary, we would do the following:

```
2.0761 / 2 = 1.0381              (remainder of 1)
1.0381 / 2 = 0.51905             (remainder of 1)
0.51905 / 2 = 0.259525           (remainder of 1)
0.259525 / 2 = 0.1297625         (remainder of 1)
0.1297625 / 2 = 0.06488125       (remainder of 1)
0.06488125 / 2 = 0.032440625     (remainder of 0)
```

The binary representation of 2.0761 is therefore 1.0111.

Floating-point numbers can represent a much wider range of values by using more bits.

CREATING THE IEEE REPRESENTATION

CREATING THE IEEE REPRESENTATION

The passage below explains how to create the IEEE representation of a floating-point number.

The first step is to normalize the sign bit, exponent, and significand fields. This means adjusting the significand so that its leading bit is 1.

For example, the significand **1.1011** can be normalized by shifting the bits one position to the left, resulting in the normalized significand **11.011**.

Next, the sign bit, exponent, and significand fields are encoded.

The sign bit is stored as a single bit: **0** represents a positive number, and **1** represents a negative number.

The exponent is stored as an 8-bit unsigned integer using a bias of **127**. This means the actual exponent is added to 127 to produce the biased exponent.
For example, an actual exponent of **5** is encoded as the biased exponent **132**.

The significand is stored as a 23-bit unsigned integer. The leading bit of the normalized significand is not stored explicitly, since it is always assumed to be 1.
As a result, the significand **11.011** is encoded as the 23-bit value **01000100000000000000000**.

Once the sign bit, exponent, and significand fields are encoded, they are combined to form the complete IEEE short real.
The following table shows an example of how to create the IEEE representation of the floating-point number $\mathbf{1.101 \times 2^{20}}$:

Field	Value
Sign bit	0
Exponent	132
Significand	01000100000000000000000
IEEE representation	01111111010001000000000000000000

The IEEE representation of a floating-point number allows computers to store and work with real numbers efficiently.

IEEE floating-point numbers are used all over the place, including scientific computing, computer graphics, and gaming.

The table below shows the **binary value**, **exponent**, **sign bit**, and **significand** for several IEEE floating-point numbers. Below is a concise explanation of what each row represents:

Binary Value	Biased Exponent	Sign, Exponent, Fraction
-1.11	127	1 01111111 11000000000000000000000
+1101.101	130	0 10000010 10110100000000000000000
-.00101	124	1 01111100 01000000000000000000000
+100111.0	132	0 10000100 00111000000000000000000
+.0000001101011	120	0 01111000 10101100000000000000000

Binary value	Exponent	Sign bit	Significand
-1.11	127	1	1101.101
130	0	0	1.10111

IEEE 754 FLOATING-POINT ENCODINGS (REBUILT)

I. Interpreting an IEEE Floating-Point Bit Pattern

An IEEE single-precision floating-point number consists of:

```
[ Sign (1 bit) | Exponent (8 bits) | Fraction (23 bits) ]
```

The value is computed as:

```
(-1)^sign × significand × 2^(exponent – bias)
```

Where:

- Bias = **127**
- Significand = 1.fraction (for normalized numbers)

II. Correcting the Example Interpretations ⚠

❌ **Common mistakes in the original text**

- Saying "exponent = 127 is the largest possible exponent" → **false**
- Mixing **binary fractions** with **decimal meanings**
- Using **unbiased exponent = 0** incorrectly
- Calling raw binary strings "IEEE values" without normalization context

Let's fix this cleanly.

III. Normalized Finite Numbers

A **normalized** IEEE number satisfies:

- Exponent field ≠ 00000000
- Exponent field ≠ 11111111
- Leading significand bit is **implicitly 1**

Example (Corrected):

Bit pattern: **0 10000010 01011000000000000000000**

Breakdown:

- **Sign bit** = 0 → positive
- **Exponent bits** = $10000010_2 = 130_{10}$
- **Unbiased exponent** = 130 − 127 = 3
- **Fraction bits** = 01011000000000000000000

IV. Special IEEE Encodings

IEEE 754 defines **real numbers AND non-numbers**.

1. Positive and Negative Zero

- Exponent = 00000000
- Fraction = all zeros
- Sign bit distinguishes +0 and −0

These compare equal numerically but **behave differently in edge cases**.

2. Normalized Finite Numbers

- Exponent ∈ [1, 254]
- Leading significand bit = 1
- Highest precision

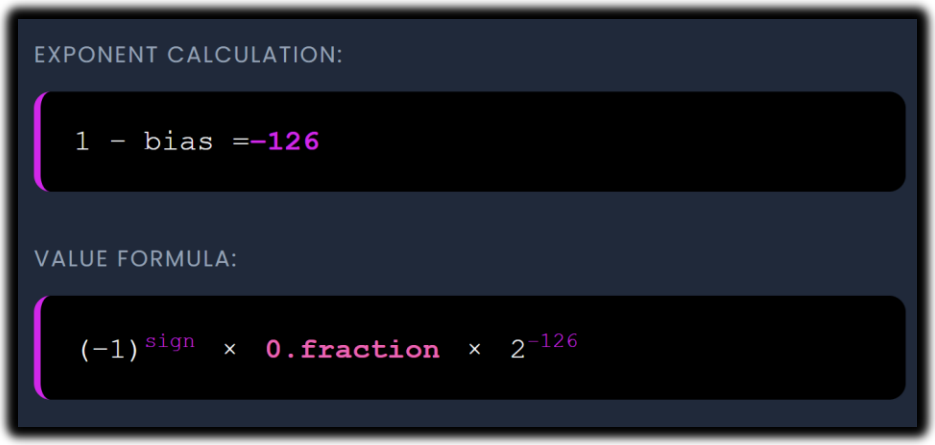
This is what **most real numbers use**.

3. Denormalized (Subnormal) Numbers ⚠

Used for values **too small** to be normalized.

Conditions:

- Exponent = 00000000
- Leading significand bit = **0** (not implicit 1)
- Exponent is fixed at:



V. Why Denormalization Exists (Very Important)

Without denormals:

- There would be a **sudden jump to zero**
- This causes numeric instability

Denormals allow:

- Gradual underflow
- Smoother behavior near zero
- Better numerical algorithms

VI. Correct Denormalization Example (Cleaned)

Original number: $\mathbf{1.01011110000000000001111 \times 2^{-129}}$

This exponent is **below −126**, so it **cannot be normalized**.

The FPU:

- Fixes exponent at −126
- Shifts the significand **right**
- Loses precision gradually

After shifting:

Key points:

- Leading bit is **0**
- Precision is reduced
- Value is still representable

This is **subnormal behavior**, not “magic”.

VII. Infinity and NaN

Positive / Negative Infinity

- Exponent = 11111111
- Fraction = all zeros
- Sign bit determines +∞ or −∞

Used for:

- Divide by zero
- Overflow

NaN (Not a Number)

- Exponent = 11111111
- Fraction ≠ zero

Used for:

- Invalid operations (0/0, ∞−∞)
- Propagates through calculations

⚠ NaN **never equals anything**, including itself.

VIII. "Indefinite" Values (x87 Context)

On x87 FPU:

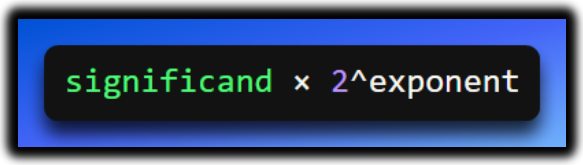
- Certain invalid operations produce **indefinite values**
- Encoded as **specific NaNs**
- Often seen when reversing legacy binaries

This is why x87 dumps look weird compared to SSE.

IX. Converting IEEE Single-Precision to Decimal (FINAL CLEAN STEPS)

Step-by-step (Correct Version)

1. Read **sign bit**
2. Extract **exponent**
    - Subtract bias (127)
3. Extract **fraction**
    - Add implicit leading 1 (unless denormal)
4. Form:



1. Shift binary point
2. Convert using positional notation

That's it. No shortcuts.

X. Why This Matters for Reverse Engineering 🧠

- Floating constants in binaries
- x87 stack dumps
- SSE registers (XMM)
- Malware anti-analysis tricks
- Precision bugs
- Compiler artifacts

If you **misunderstand IEEE**, you misread behavior.

XI. Summary

- IEEE is deterministic, not mysterious
- Normalized ≠ Denormalized
- Exponent bias matters
- Precision is finite
- Some numbers **cannot** be exact
- Zero has two signs
- NaN breaks comparisons
- This shows up **everywhere** in low-level code

POSITIVE AND NEGATIVE INFINITY

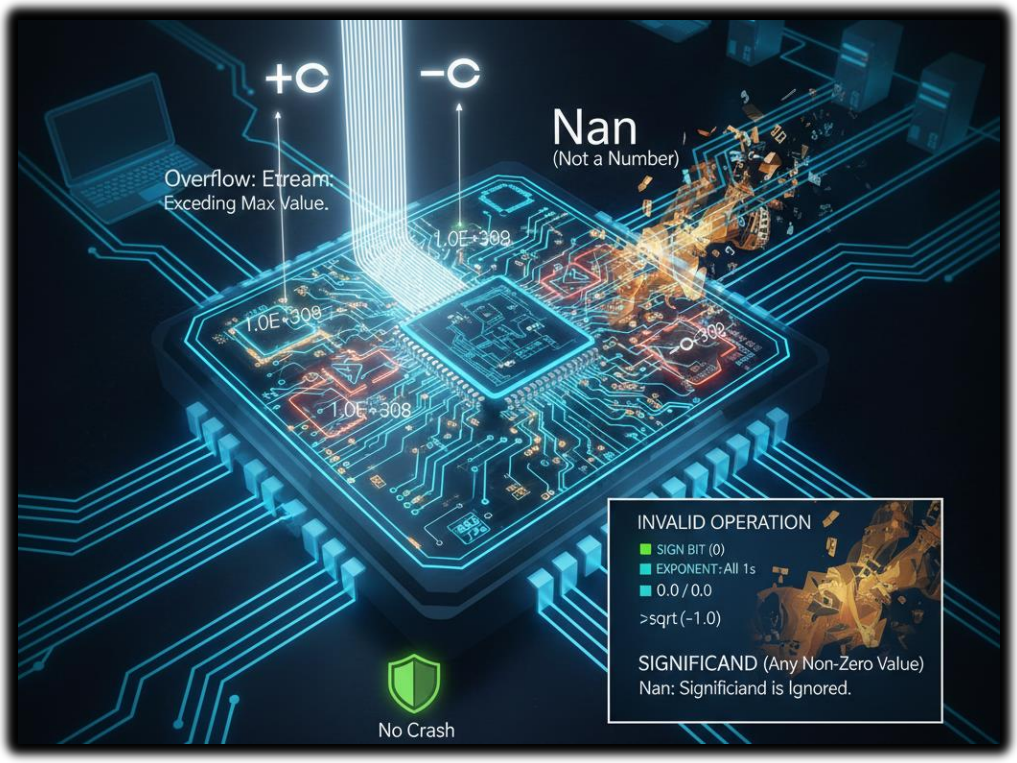
Special Floating-Point Values: Infinity & NaNs

- **Positive and Negative Infinity** In floating-point math, infinity isn't just a concept—it's an actual special value.
  - **Positive infinity** represents numbers that are too large to fit in the normal range.
  - **Negative infinity** represents numbers that are too small (on the negative side). These values pop up when operations "overflow," like dividing by zero or adding a huge number to a tiny one.
- **NaNs (Not a Number)** Sometimes math goes off the rails—like dividing zero by zero or trying to take the square root of a negative number. The result isn't a valid real number, so computers use **NaNs** to stand in.
  - A NaN is basically a special bit pattern that says: *"This isn't a real number, but I need to represent the result somehow."*
  - There are two flavors: **quiet NaNs** (which silently propagate through calculations) and **signaling NaNs** (which raise exceptions to flag errors).
- **IEEE Standard Encodings** The IEEE floating-point standard defines exactly how these special values are stored in memory. There are specific encodings for:
  - Positive infinity
  - Negative infinity
  - Quiet NaNs

Type	Sign bit	Exponent	Significand
Positive infinity	0	11111111	00000000
Negative infinity	1	11111111	00000000
Quiet NaN	x	11111111	xxxxxxx...
Signaling NaN	x	11111110	xxxxxxx...

How FPUs Handle Infinity and NaNs

- **NaN's Significand Field** For NaNs, the significand (the fraction part of the floating-point number) can actually be *anything*. That's part of what makes NaNs flexible placeholders for "not-a-number" results.
- **Special Treatment by FPUs** Floating-point units (FPUs) don't just crunch numbers blindly—they have built-in rules for handling these special values:
    - If an operation would blow up into **positive or negative infinity**, the FPU doesn't crash—it simply returns the matching infinity value.
    - If an operation leads to a **NaN**, the FPU returns a NaN instead. In some cases, it may also raise an exception to let the system know something unusual happened.
- **Consistency Across Systems** All of this behavior isn't random—it's carefully defined by the **IEEE floating-point standard**. Thanks to this standard, FPUs from different manufacturers and architectures behave consistently. That means whether you're running code on a laptop, a server, or a supercomputer, infinity and NaNs are handled the same way.



Special Single-Precision Encodings

VALUE	SIGN	EXPONENT	SIGNIFICAND
Positive zero	0	00000000	00000000000000000000000
Negative zero	1	00000000	00000000000000000000000
Positive infinity	0	11111111	00000000000000000000000
Negative infinity	1	11111111	00000000000000000000000
QNaN	x	11111111	1xxxxxxxxxxxxxxxxxxxxxx
SNaN	x	11111111	0xxxxxxxxxxxxxxxxxxxxxx

* ***SNaN*** significand field begins with 0, but at least one of the remaining bits must be 1.

CONVERTING DECIMAL FRACTIONS TO BINARY REALS

Converting Decimal Fractions to Binary Reals

There are a couple of neat ways to turn decimal fractions into their binary equivalents. Let's break them down:

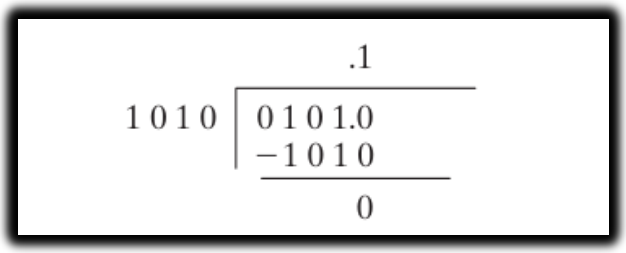
- **Method 1: Breaking into Fractions**
    1. Write the decimal fraction as a sum of fractions like 1/2+1/4+1/8+….
    2. Convert each of those fractions into binary form.
    3. Add the binary fractions together to get the final binary result.
- **Method 2: Long Division in Binary**
    1. Convert both the numerator and denominator of the decimal fraction into binary.
    2. Perform long division using those binary numbers.
    3. The quotient you get is the binary representation of the decimal fraction.

Example (Using Method 2) Let's try converting **0.5** into binary:

First, convert the numerator and denominator into binary.

- 5 in decimal → **101** in binary
- 10 in decimal → **1010** in binary

Next, perform long division on these binary numbers:



💡 **Tip:** Method 1 is often faster for simple fractions, while Method 2 is more general for any decimal fraction.

Binary Long Division Method

Example 1: Converting 0.5 (5/10)

Fraction: 5/10 = 1/2

Binary: $1 \div 10_2 = 0.1_2$

✓ Remainder = 0

Simple, exact, and done.

Example 2: 0.2 (2/10)

REPEATING PATTERN

Step 1: Decimal Fraction

$0.2 = 2/10$

Step 2: Binary Conversion

$2 = 10_2$

$10 = 1010_2$

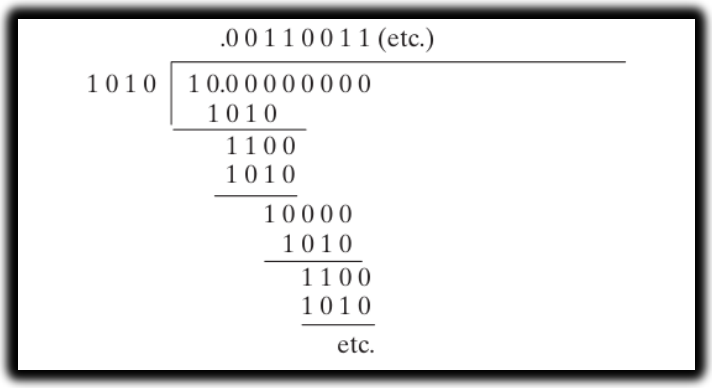
Step 3: Long Division Setup

$10_2 \div 1010_2$

10_2 is smaller than 1010_2. We add binary places (multiply by 2 each step) to keep going. This generates:
$0.0011\ldots_2$

I. Key idea:

- Some decimal fractions, like 0.2, **cannot be represented exactly in binary** because their denominator (10) has prime factors other than 2.
- That's why the division never stops cleanly.
- In single-precision floating-point (IEEE 754), we truncate or round the repeating sequence:
  - Significand for 0.2 ≈ 00110011001100110011001 (23 bits).



Examples of decimals factored as binary reals:

DECIMAL FRACTION	FACTORED AS...	BINARY REAL
$\frac{1}{2}$	$\frac{1}{2}$	.1
$\frac{1}{4}$	$\frac{1}{4}$	.01
$\frac{3}{4}$	$\frac{1}{2} + \frac{1}{4}$	.11
$\frac{1}{8}$	$\frac{1}{8}$	.001
$\frac{7}{8}$	$\frac{1}{2} + \frac{1}{4} + \frac{1}{8}$	.111
$\frac{3}{8}$	$\frac{1}{4} + \frac{1}{8}$	.011
$\frac{1}{16}$	$\frac{1}{16}$	.0001
$\frac{3}{16}$	$\frac{1}{8} + \frac{1}{16}$	.0011
$\frac{5}{16}$	$\frac{1}{4} + \frac{1}{16}$	.0101

Think of this table like a **cheat sheet** for turning everyday fractions into computer language. Here is the simple breakdown of how it works:

II. How to read the table

- **Column 1 (The Fraction):** This is the normal decimal fraction you are starting with.
- **Column 2 (The Recipe):** This shows the ingredients (like 1/2 or 1/4) used to make that fraction.
- **Column 3 (The Binary):** This is the final code the computer understands.

III. Examples of how it works

- **The Basic 1/2:** To get 1/2, the recipe is just 1/2, and the binary is **.1**.
- **The Basic 1/4:** To get 1/4, the recipe is just 1/4, and the binary is **.01**.
- **Mixing Ingredients:** To get **3/4**, you mix 1/2 and 1/4 together (1/2 + 1/4). The binary becomes **.11** because you have a piece in both the "half" spot and the "quarter" spot.

IV. How to use it for yourself

1. **Find your fraction** in the first column.
2. **Look across** to the third column to find its binary code.
3. **Example:** If you need to convert **5/8**, you find it in the list and see the binary is **.101**.

V. A quick warning: The Infinity Glitch

- Most fractions are easy to convert, but **not all of them fit perfectly**.
- Binary only has a limited number of "slots" to use, but numbers can go on forever.
- Because of this, some numbers have to be **approximated** (rounded off), which is why computers sometimes have tiny rounding errors.

VI. An Analogy to keep it simple:

Imagine you have a set of **building blocks** that are only certain sizes: a 5-inch block, a 2.5-inch block, and a 1.25-inch block.

- If you want to build a tower exactly 7.5 inches tall, you just use the 5-inch and the 2.5-inch blocks.
- But if you want a tower that is exactly 7.1 inches tall, none of your blocks fit perfectly! You have to use the ones you have to get as close as possible. That is exactly what a computer does with binary fractions.

VII. How to Factor a Fraction

To find the "ingredients" for your fraction, follow these three simple steps:

1. **Find the biggest piece:** Look for the largest power of two (like 1/2, 1/4, 1/8) that fits inside your fraction.
2. **Subtract it:** Take that piece away from your total.
3. **Repeat:** Keep doing this with the leftover amount until you have nothing left.

The powers of two you subtracted are your **factors**.

VIII. A Real Example: Factoring 3/4

Let's see how we get the factors for **3/4**:

- **Step 1:** The biggest power of two that fits in 3/4 is **1/2**.
- **Step 2:** Subtracting 1/2 from 3/4 leaves you with **1/4**.
- **Step 3:** The biggest power of two that fits in 1/4 is **1/4**.
- **Step 4:** Subtracting 1/4 from 1/4 leaves you with **zero**.

Result: The factors of 3/4 are **1/2** and **1/4**.

IX. Converting Factors to Binary

Once you have your factors, you just need to know their binary "code":

- **Factor 1/2:** Divide 1 by 2. You get 0.5, which in binary is **.1**.
- **Factor 1/4:** Divide 1 by 4. You get 0.25, which in binary is **.01**.

The Final Step: Add the binary codes together.

Since **.1** + **.01** = **.11**, the binary version of **3/4** is **0.11**.

X. A quick analogy:

Imagine you are at a store and need to pay **75 cents** (3/4 of a dollar), but you can only use a **50-cent** coin (1/2) and a **25-cent** coin (1/4).

- You use one 50-cent coin (put a **1** in the first slot).
- You use one 25-cent coin (put a **1** in the second slot).
- Your total is **.11**.

Questions

Why doesn't the single-precision real format permit an exponent of -127?

The single-precision real format does not permit an exponent of -127 because it would generate a zero. This is because the unbiased exponent is calculated by subtracting 127 from the exponent. If the exponent is -127, then the unbiased exponent would be zero. However, the significand of a non-zero floating-point number cannot be zero. Therefore, an exponent of -127 is not permitted.

Why doesn't the single-precision real format permit an exponent of 128?

The single-precision real format does not permit an exponent of 128 because it would generate an overflow. This is because the unbiased exponent is calculated by subtracting 127 from the exponent. If the exponent is 128, then the unbiased exponent would be 1. However, the unbiased exponent cannot be greater than 255. Therefore, an exponent of 128 is not permitted.

In the IEEE double-precision format, how many bits are reserved for the fractional part of the significand?

The IEEE double-precision format reserves 52 bits for the fractional part of the significand. This gives double-precision floating-point numbers a much higher degree of precision than single-precision floating-point numbers.

In the IEEE single-precision format, how many bits are reserved for the exponent?

The IEEE single-precision format reserves 8 bits for the exponent. This gives single-precision floating-point numbers a range of values from 2^{-126} to 2^{127}.

FLOATING POINT UNIT

Floating Point Unit (FPU)

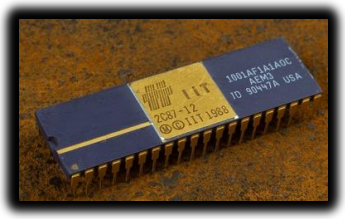
1. **Definition:**
   - The **FPU** is a specialized processor or hardware unit designed to handle **floating-point arithmetic** efficiently.
   - Floating-point numbers are numbers with a fractional part (e.g., 3.14, -0.5, 2.0×10^3).
2. **Registers:**
   - The FPU has its **own registers**, often organized as a **register stack**.
   - Registers are used to **temporarily store operands and results** during calculations.
3. **Operation:**
   - The FPU can **load values from memory** into the stack, **perform arithmetic operations**, and **store results back to memory**.
   - It performs calculations in **postfix (Reverse Polish) notation**:
     - **Postfix:** Operands come **before** the operator.
     - **Example:** Infix: $(5 \times 6) - 4 \rightarrow$ Postfix: 5 6 * 4 –
4. **Why postfix?**
   - Postfix notation eliminates the need for parentheses.
   - The FPU can evaluate expressions **directly using a stack**, which simplifies hardware design.

💡 **Extra tip for notes:** Modern FPUs support single-precision (32-bit) and double-precision (64-bit) floating-point formats and handle special values like infinity, NaN, and denormal numbers. This gives a more complete picture.

Left to Right	Stack		Action
5	5	ST (0)	push 5
5 6	5 6	ST (1) ST (0)	push 6
5 6 * 4	30	ST (0)	Multiply ST(1) by ST(0) and pop ST(0) off the stack.
5 6 * 4	30 4	ST (1) ST (0)	push 4
5 6 * 4 -	26	ST (0)	Subtract ST(0) from ST(1) and pop ST(0) off the stack.

I. FPU Expression Stack and Postfix Evaluation

1. **Expression Stack:**
   - The FPU uses a **stack** to hold intermediate results during calculations.
   - The stack follows **LIFO (Last-In-First-Out)**:
     - **Last value pushed → first value popped**
2. **Postfix Expression Evaluation:**
   To evaluate a postfix expression:
   - **Push operands** onto the stack as they appear.
   - When an **operator** appears, **pop the top two operands** from the stack.
   - **Perform the operation** and **push the result** back onto the stack.
   - Repeat until **all operators have been processed**.
   - The **final result** is the **only value left** on the stack.



II. Example Table: Infix → Postfix

Infix expression	Postfix expression
(5 * 6) - 4	5 6 * 4 -
(A + B) * C	A B + C *
(A - B) / C	A B - C /
A + B	A B +
(A - B) / D	A B - D /
(A + B) * (C + D)	A B + C D + *
((A + B) / C) * (E - F)	A B + C / E F - *

III. Postfix Expressions and FPU Evaluation

In postfix expressions, parentheses aren't needed because the **order of operations** is already determined by the order of the operands.

For example, take the infix expression **(A + B) * (C + D)**. In postfix form, the expression could be written without parentheses, yet the multiplication still happens after the additions. That's because **multiplication naturally has higher precedence than addition**, so the order is implied.

Here's how an FPU would evaluate the simple postfix expression **A B +** using its **expression stack**:

1. **Expression stack:** empty
2. **Next operand:** A → push onto stack
    - **Expression stack:** A
3. **Next operand:** B → push onto stack
    - **Expression stack:** A, B
4. **Next operator:** + → pop the top two operands (A and B), perform the addition, and push the result back onto the stack
    - **Expression stack:** A + B

FPU Expression Stack Evaluation: A B +

STEP	ACTION	STACK CONTENTS
1	Initialize	*Stack is empty*
2	push A	A
3	push B	A, B
4	pop B, A → add → push	A + B
5	Result	A + B

At this point, the FPU has no more operands or operators to process, so it stops. The final result is **A + B**.

This is the same process the FPU uses for **all postfix expressions**:

- Push operands onto the expression stack.
- Pop the top two operands when an operator appears.
- Perform the operation and push the result back onto the stack.
- Repeat until all operators are processed.

The **final value remaining on the stack** is the result of the expression.

FPU Data Registers

The FPU has **eight 80-bit data registers**, named **R0** through **R7**, which together form a **register stack**.

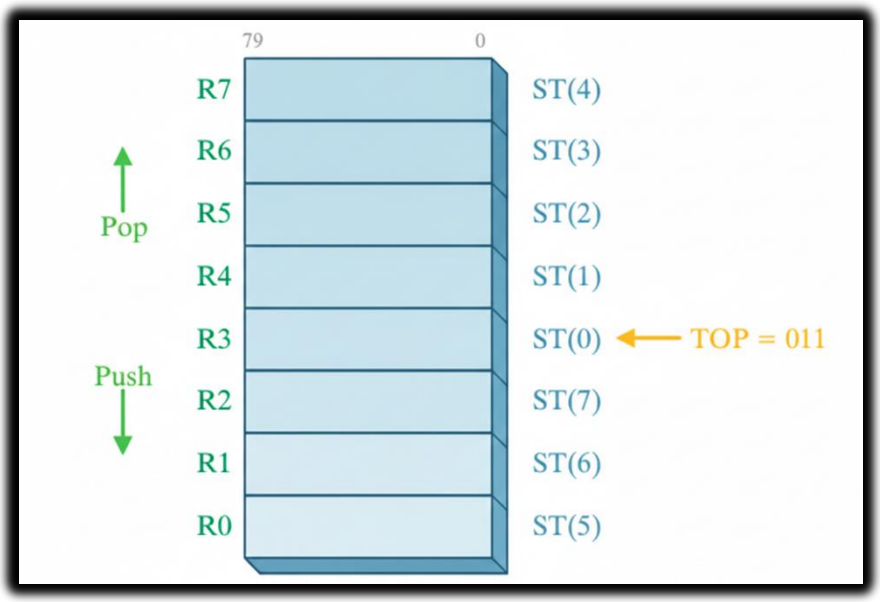
This stack operates in a **last-in, first-out (LIFO)** fashion—meaning the most recently pushed value is the first one to be popped off.

The **top of the stack** is tracked by a **three-bit field called TOP** in the FPU status word. The register at the top is referred to as **ST(0)**, while the remaining registers are labeled **ST(1), ST(2), ..., ST(7)**, in order.

I. Pushing a Value

When you push a value onto the FPU stack:

- The FPU **decrements TOP by 1**.
- The value is copied into the register currently at **ST(0)**.
- If **TOP is 0 before the push**, it wraps around to **R7**.



II. Popping a Value

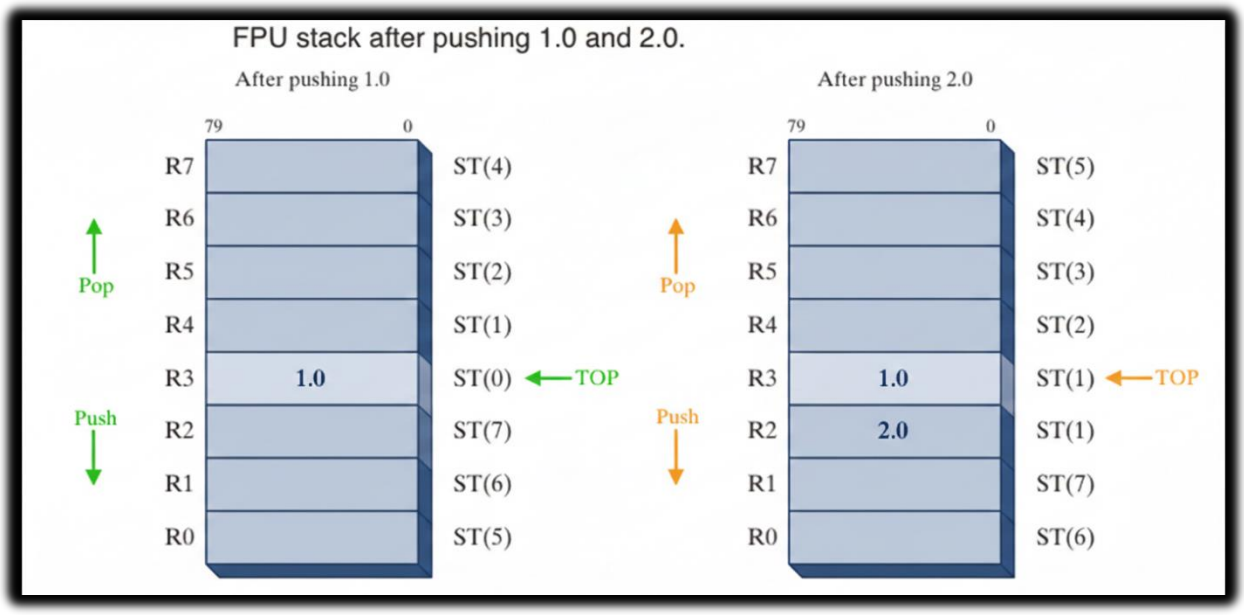
When you pop a value from the stack:

- The FPU copies the data from **ST(0)** into your operand.
- **TOP is incremented by 1**.
- If **TOP is 7 before the pop**, it wraps around to **R0**.

III. Handling Stack Overflows

If you try to load a value into the stack **when it would overwrite existing data**, the FPU raises a **floating-point exception** to signal the problem.

The following diagram shows the FPU stack after 1.0 and 2.0 have been pushed onto the stack:



IV. How the FPU Performs Floating-Point Operations

To carry out a floating-point calculation, the **FPU** follows a simple cycle:

1. **Pop the operands** off the stack.
2. **Perform the operation** (addition, subtraction, multiplication, etc.).
3. **Push the result** back onto the stack.

Example: To compute **1.0 + 2.0**, the FPU:

- Pops **1.0** and **2.0** from the stack
- Adds them together to get **3.0**
- Pushes **3.0** back onto the stack

This stack-based system allows the FPU to perform calculations efficiently **without storing intermediate results in memory**, making it especially powerful for complex floating-point operations.

Key Features of the FPU

I. ST(n) Notation:

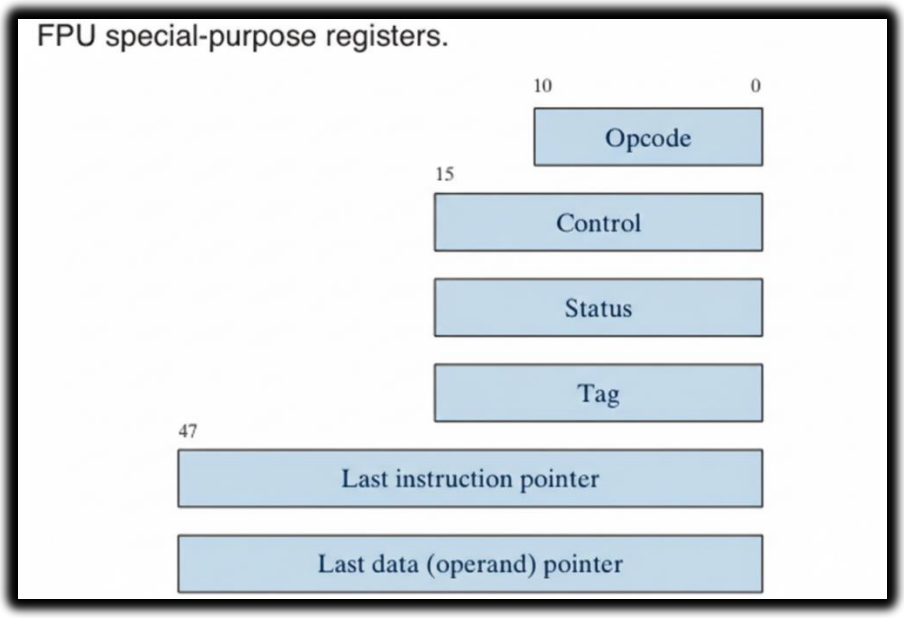
- The FPU refers to its stack registers using **ST(0), ST(1), ..., ST(7)**.
- **ST(0)** always represents the **top of the stack**.
- This notation makes it easy to address data within the FPU's **stack-based architecture**.

II. Floating-Point Value Format:

- Values in the FPU registers are stored in the **IEEE 10-byte extended real format**, sometimes called **temporary real**.
- When the FPU writes a result to memory, it can convert it into several formats:
  - **Integer**
  - **Long integer**
  - **Single precision**
  - **Double precision**
  - **Packed binary-coded decimal (BCD)**

The FPU's combination of a **stack-based design** and flexible **value formats** allows it to handle a wide variety of calculations efficiently and accurately.

III. Special-Purpose Registers in the FPU



The **Floating-Point Unit (FPU)** has several special-purpose registers that help it manage operations and maintain precision:

- **Opcode Register:** Stores the opcode of the last non-control instruction executed.
- **Control Register:** Determines the **precision** and **rounding method** used during calculations. It can also **mask individual floating-point exceptions**.
- **Status Register:** Contains the **top-of-stack pointer**, **condition codes**, and information about **exceptions or warnings**.
- **Tag Register:** Tracks the content of each FPU data-register stack entry. It uses **two bits per register** to indicate whether the register holds a valid number, zero, a special value (NaN, infinity, denormal, unsupported format), or is empty.
- **Last Instruction Pointer Register:** Stores a pointer to the last non-control instruction executed.
- **Last Data (Operand) Pointer Register:** Stores a pointer to a data operand, if any, used by the last instruction.

Why They Matter:
Operating systems use these registers to **preserve the FPU state** when switching between tasks. This is essential in **multitasking environments**, where the CPU frequently switches between different programs or threads.

IV. Rounding in the FPU

Floating-point calculations often produce results that **can't be represented exactly** in the destination format. Rounding is used to adjust these results so they fit the available precision.

Example:

- Precise result: **1.0111**
- Destination format: **3 fractional bits**

Rounding options:

1. **Round up:** Add 0.0001 → **1.100**
2. **Round down:** Subtract 0.0001 → **1.011**

For negative values, the same logic applies:

- **-1.0111 → -1.100** (round up in absolute value)
- **-1.0111 → -1.011** (round down in absolute value)

V. FPU Rounding Methods

The FPU supports **four rounding methods**, controlled by the **RC (Rounding Control) field** in the **control word**:

1. **Round to Nearest Even (default):** Round to the nearest value. If two values are equally close, choose the even one (least significant bit = 0).
2. **Round Down Toward −∞:** Round to a value **less than or equal to** the precise result.
3. **Round Up Toward +∞:** Round to a value **greater than or equal to** the precise result.
4. **Round Toward Zero (Truncation):** Round toward zero; the absolute value of the result is **less than or equal to** the precise result.

VI. RC Field Binary Encoding:

The tables below would show how these rounding methods would be applied to the binary values 1.0111 and -1.0111, respectively.

FPU Rounding Examples

Comparing binary rounding for positive vs. negative values

EXAMPLE: ROUNDING +1.0111

METHOD	PRECISE RESULT	ROUNDED
Round to Nearest Even	1.0111	1.100
Round Down (-∞)	1.0111	1.011
Round Up (+∞)	1.0111	1.100
Round Toward Zero	1.0111	1.011

EXAMPLE: ROUNDING -1.0111

METHOD	PRECISE RESULT	ROUNDED
Round to Nearest (Even)	-1.0111	-1.100
Round Down (-∞)	-1.0111	-1.100
Round Up (+∞)	-1.0111	-1.011
Round Toward Zero	-1.0111	-1.011

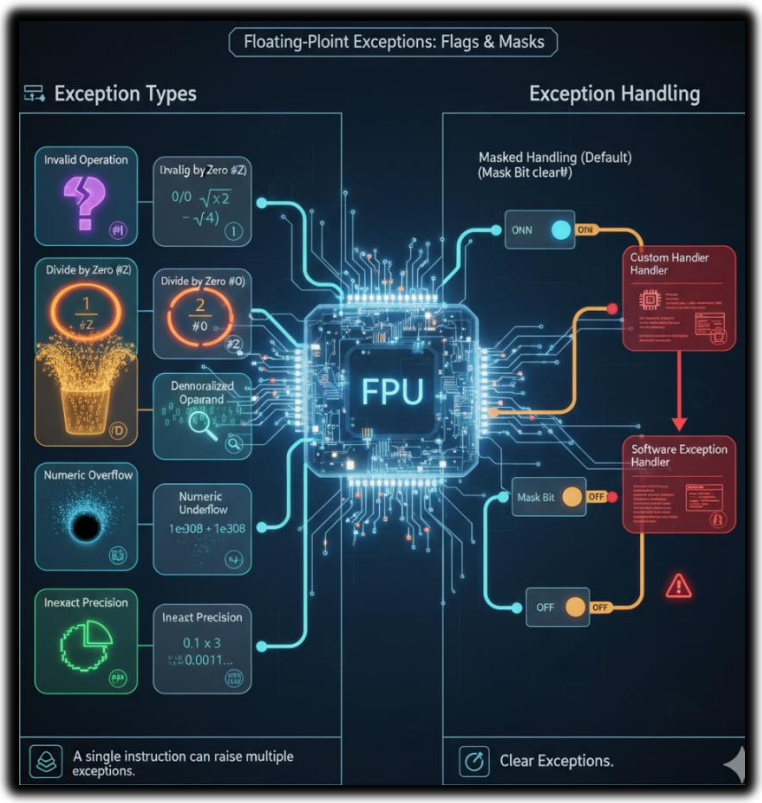
Rounding in floating-point arithmetic is essential to ensure that calculated results fit within the chosen format while minimizing error.

The method chosen depends on the specific requirements of the application.

FLOATING-POINT EXCEPTIONS

Floating-point exceptions are **errors that can occur during floating-point calculations**. The FPU recognizes six main types of exceptions:

1. **Invalid Operation (#I):**
   Raised when an illegal operation is attempted, such as dividing **0 by 0** or taking the **square root of a negative number**.
2. **Divide by Zero (#Z):**
   Triggered when a number is divided by **zero**.
3. **Denormalized Operand (#D):**
   Occurs when an operation is attempted on a **denormalized number**, which is a number very close to zero.
4. **Numeric Overflow (#O):**
   Happens when the result of a calculation is **too large** to be represented by the FPU.
5. **Numeric Underflow (#U):**
   Happens when the result is **too small** to be represented by the FPU.
6. **Inexact Precision (#P):**
   Raised when a calculation cannot be represented exactly due to the **limited number of bits** used for floating-point numbers.



I. How Exceptions Are Handled

Each exception has:

- A **flag bit**, which is set when the exception occurs.
- A **mask bit**, which controls how the processor responds:
  - If the **mask bit is set**, the processor handles the exception automatically and the program continues.
  - If the **mask bit is clear**, a **software exception handler** is invoked.

For most programs, the processor's **automatic (masked) handling** is sufficient. However, you can use **custom exception handlers** when specific behavior is required.

II. Important Notes

- A **single instruction** can trigger **multiple exceptions**.
- The processor keeps track of **all exceptions** that occur until they are cleared.
- After a series of calculations, you can check which exceptions happened.

III. Examples of Floating-Point Exceptions

- **Divide by zero:** $10 \div 0 \rightarrow$ #Z
- **Invalid operation:** $\sqrt{(-4)} \rightarrow$ #I
- **Numeric overflow:** 1e308 + 1e308 → #O
- **Numeric underflow:** 1e-308 – 1e-308 → #U
- **Inexact precision:** 0.1 × 3 in binary → #P

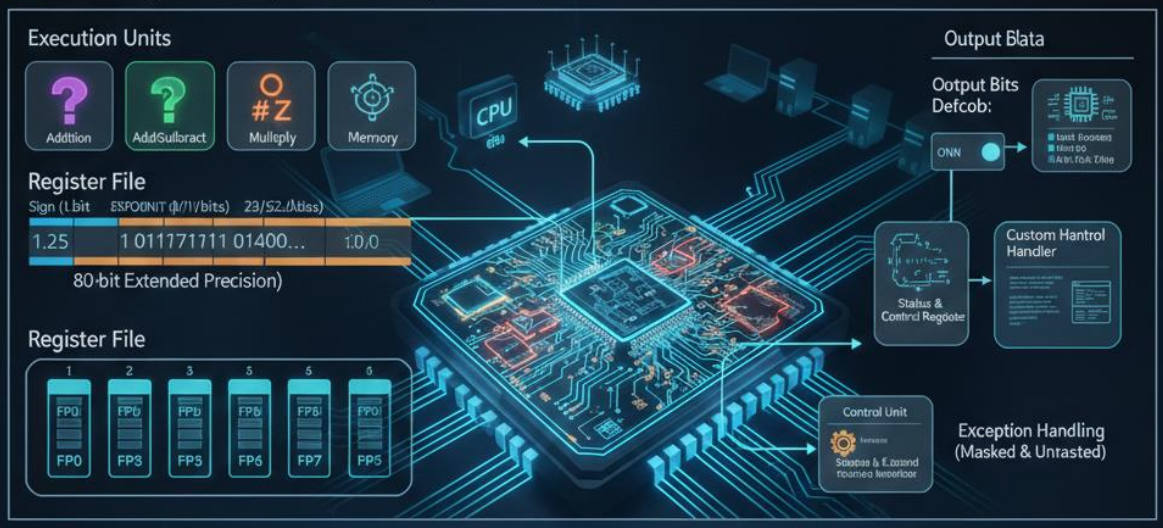
IV. Why It Matters

Floating-point exceptions can **cause unexpected program behavior**. It's important to:

- Understand the different types of exceptions
- Know how the FPU handles them automatically
- Use **mask bits** or **custom handlers** to control exception behavior when needed

Floating-Ploint Unit (FPU: Architecture & Operations

Floating-Ploint Represetration (IEEE 754)



Common FPU Operations

Common FPU Operations

Key Charactersitics

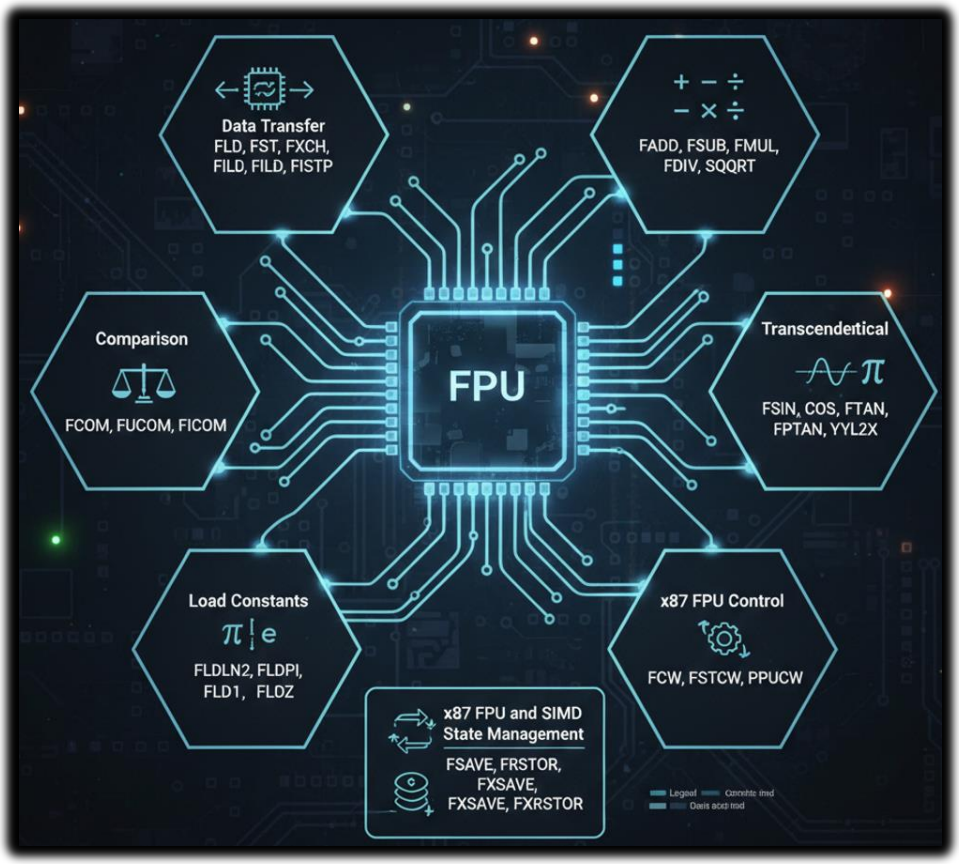
- Parallel Exegisters

The FPU is specialized coprocssor for fast, precise flooting-plonit arthemtic.

FLOATING-POINT INSTRUCTION SET

The **floating-point instruction set** is a collection of instructions that the **FPU** uses to perform floating-point calculations. These instructions are organized into several categories:

1. **Data Transfer:** Move data between **FPU registers** and **memory**.
2. **Basic Arithmetic:** Perform operations like **addition, subtraction, multiplication, and division**.
3. **Comparison:** Compare two floating-point numbers and indicate whether one is **greater than, equal to, or less than** the other.
4. **Transcendental:** Perform functions such as **sine, cosine, tangent**, and other advanced mathematical operations.
5. **Load Constants:** Load predefined constants directly into **FPU registers**.
6. **x87 FPU Control:** Manage the behavior of the FPU, including **rounding mode** and **calculation precision**.
7. **x87 FPU and SIMD State Management:** Save or restore the **FPU register stack** and other state information.



I. Instruction Naming Conventions

- Floating-point instructions **start with the letter F**.
- The **second letter** indicates how a **memory operand** is interpreted:
  - **B** → BCD operand
  - **I** → Binary integer operand
  - If neither is specified, the operand is assumed to be a **real number**.
- Instructions can have **zero, one, or two operands**.
  - If there are **two operands**, one must always be a **floating-point register**.
  - There are **no immediate operands**, but certain **predefined constants** can be loaded onto the stack.
  - Integer operands must be **loaded from memory** before they can be used in floating-point calculations.
  - When storing floating-point values into integer memory, the FPU **truncates or rounds** the result automatically.

II. Initializing the FPU

- Use the **FINIT instruction** to initialize the FPU.
- FINIT sets the FPU control word to **037Fh**, which:
  - Masks all floating-point exceptions
  - Sets **rounding to nearest even**
  - Sets **calculation precision to 64 bits**

Tip: It's recommended to call **FINIT at the start of your programs** to ensure a known starting state for the FPU.

Here are some examples of floating-point instructions:

```
FLD [memory_address] ;Load a floating-point value from memory into the FPU stack.
FADD ST(0), ST(1) ;Add the top two values on the FPU stack and store the result at the top of the stack.
FSTP [memory_address] ;Store the top value on the FPU stack to memory.
```

FLOATING POINT DATATYPES

MASM supports the following floating-point data types:

- **QWORD:** 64-bit integer
- **TBYTE:** 80-bit (10-byte) integer
- **REAL4:** 32-bit (4-byte) IEEE short real
- **REAL8:** 64-bit (8-byte) IEEE long real
- **REAL10:** 80-bit (10-byte) IEEE extended real

I. Why These Datatypes Matter

When you're working with FPU instructions, the processor doesn't just accept any random number format. It requires operands to be defined as one of these datatypes.

- For **floating-point variables**, you'll typically use **REAL4**, **REAL8**, or **REAL10**, depending on the precision you need.
- For example, if you want to load a floating-point variable into the FPU stack, it must be defined in one of these formats. Otherwise, the assembler won't know how to interpret the data correctly.

II. Quick Example

Suppose you want to load a floating-point value into the FPU stack:

```
.data
myFloat REAL4 3.14    ; Define a 32-bit floating-point variable

.code
fld myFloat           ; Load the variable into the FPU stack
```

Here, REAL4 ensures the assembler knows the variable is a single-precision floating-point number, and the fld instruction correctly loads it into the FPU.

Another example:

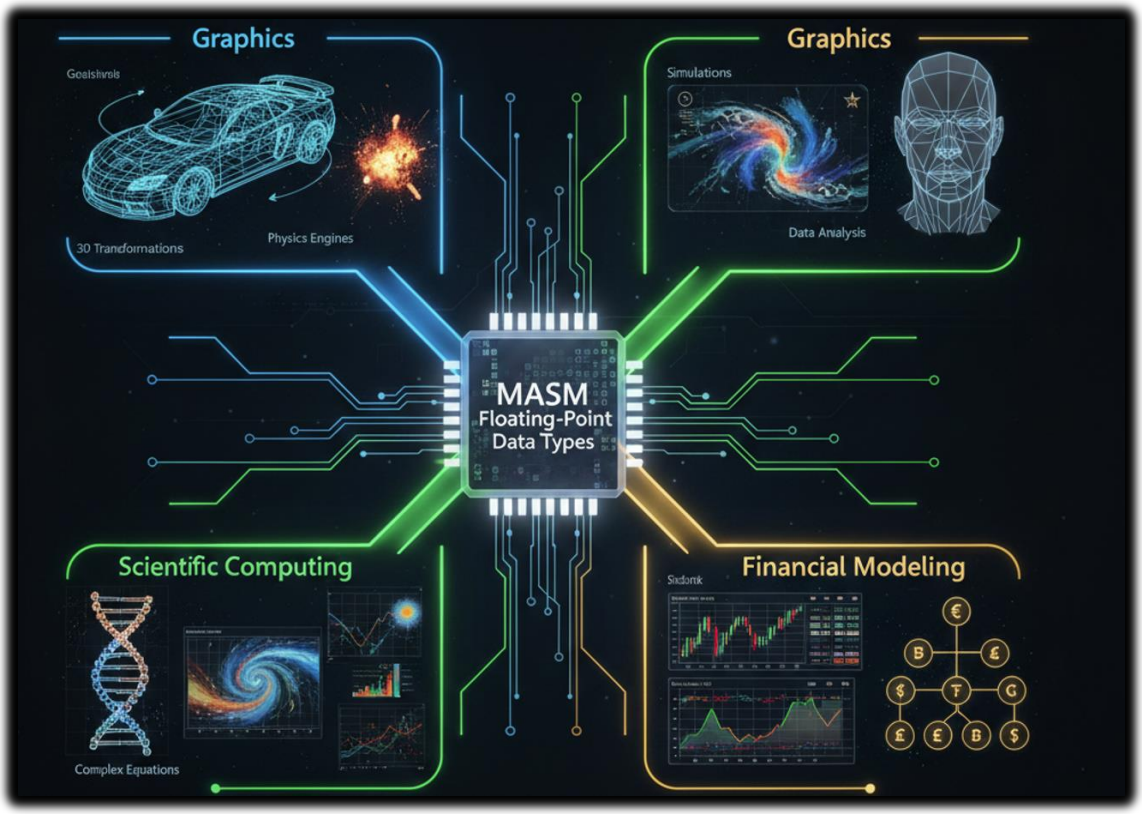
```
; Define a double-precision floating-point variable named bigVal
.data
bigVal REAL8 1.212342342234234243E+864

; Start of the code section
.code
; Load the value of bigVal into the FPU stack
fld QWORD PTR [bigVal]

; Here, QWORD PTR is used to indicate that we are loading a double-precision (64-bit) value from memory.
; The brackets [] indicate that we are accessing the memory location pointed to by bigVal.

; The value of bigVal is now loaded onto the FPU stack.
```

fld instruction loads the value of bigVal onto the FPU stack, preparing it for floating-point operations.

bigVal is defined as **REAL8**, so fld knows how to interpret its memory representation.

MASM supports various **floating-point data types**, which are widely used in:

- Graphics
- Scientific computing
- Financial modeling



Load Floating Point Values

This passage describes the FLD and FILD instructions.

The **FLD (load floating-point value)** instruction copies a floating-point operand to the top of the FPU stack (known as ST(0)).

The operand can be a 32-bit, 64-bit, or 80-bit memory operand (REAL4, REAL8, REAL10) or another FPU register.

The **FILD (load integer)** instruction converts a 16-, 32-, or 64-bit signed integer source operand to double-precision floating point and loads it into ST(0). The source operand's sign is **preserved.**

The following instructions load specialized constants on the stack. They have no operands:

- **FLD1:** pushes 1.0 onto the register stack.
- **FLDL2T:** pushes $\log_2 10$ onto the register stack.
- **FLDL2E:** pushes $\log_2 e$ onto the register stack.
- **FLDPI:** pushes π onto the register stack.
- **FLDLG2:** pushes $\log_{10} 2$ onto the register stack.
- **FLDLN2:** pushes $\log_e 2$ onto the register stack.
- **FLDZ:** pushes 0.0 on the FPU stack.

Here are some examples of how to use the FLD and FILD instructions:

```
090 .data
091     dblOne    REAL8 234.56
092     dblTwo    REAL8 10.1
093     intVal    DWORD 100
094
095 .code
096     ; Load the value of dblOne onto the FPU stack.
097     fld     QWORD PTR [dblOne]
098
099     ; Load the value of dblTwo onto the FPU stack.
100     fld     QWORD PTR [dblTwo]
101
102     ; Load the integer value intVal onto the FPU stack, preserving its sign.
103     fild    DWORD PTR [intVal]
104
105     ; Load the mathematical constant p onto the FPU stack.
106     fldpi
107
108     ; Load the constant 1.0 onto the FPU stack.
109     fld1
```

In this example, you're working with a few variables and then loading them onto the **Floating-Point Unit (FPU) stack**:

- **dblOne** and **dblTwo**
  - Both are **double-precision floating-point variables** (64 bits each).
  - dblOne holds the value **234.56**.
  - dblTwo holds the value **10.1**.
- **intVal**
  - A **32-bit integer variable** with the value **100**.

I. Code Instructions Explained

- fld QWORD PTR [dblOne] Loads the value of dblOne (234.56) onto the FPU stack.
  - QWORD PTR tells the assembler you're accessing a 64-bit value in memory.
  - fld is the instruction used for loading floating-point values.
- fld QWORD PTR [dblTwo] Works the same way as above, but loads dblTwo (10.1) onto the stack.
- fild DWORD PTR [intVal] This one's different: it loads the **integer** value intVal (100) onto the FPU stack.
  - fild is specifically for loading integers.
  - It preserves the sign and automatically converts the integer into floating-point format inside the FPU.
- fldpi Loads the mathematical constant **π (pi)** onto the stack.
  - This is a predefined constant available in the FPU.
- fld1 Loads the constant **1.0** onto the stack.
  - Another predefined constant, representing the floating-point value 1.0.

II. Storing Values: FST and FSTP

Once values are on the stack, you'll often want to store them back into memory or another register. That's where **FST** and **FSTP** come in:

- **FST (Store Floating-Point Value)**
  - ❖ Copies the value at the top of the FPU stack (**ST(0)**) into memory or another FPU register.
  - ❖ Supports memory operand types like **REAL4**, **REAL8**, and **REAL10**.
  - ❖ Important: it **does not pop the stack**—the value stays on the stack for further use.
- **FSTP (Store and Pop)**
  - ❖ Similar to FST, but after storing the value, it **pops the stack**, removing the top element.
  - ❖ Useful when you're done with a value and want to free up stack space.

👉 **In short:**

- fld and fild are about **loading values onto the stack**.
- fst and fstp are about **storing values back** (with or without popping).

III. FSTP (Store Floating-Point Value and Pop)

The **FSTP** instruction is like the sibling of **FST**, but with an extra step built in:

- It **copies the value in ST(0)** (the top of the FPU stack) into memory.
- Then, it **pops ST(0) off the stack**, physically removing it.
- It supports the same memory operand types as FST (**REAL4, REAL8, REAL10**) and can also store values into other FPU registers.
- By popping the stack, it updates the **TOP pointer**, which keeps track of the current stack position.

In other words, FSTP doesn't just store—it also cleans up the stack at the same time. This makes it especially handy when you're done with a value and want to free up space for the next calculation.

IV. Working with the FPU: Loading and Storing Values

Here's a code example illustrating how these instructions are used:

```
.data
    dblOne   REAL8 234.56
    dblTwo   REAL8 10.1
    dblThree REAL8 0.0
    dblFour  REAL8 0.0

.code
    ; Load values onto the FPU stack
    fld QWORD PTR [dblOne]  ; ST(0) = 234.56
    fld QWORD PTR [dblTwo]  ; ST(0) = 10.1, ST(1) = 234.56

    ; Store values from the stack to memory without popping
    fst QWORD PTR [dblThree]  ; Stores 10.1 in dblThree
    fst QWORD PTR [dblFour]   ; Stores 10.1 in dblFour

    ; Reset the values
    fld QWORD PTR [dblOne]  ; ST(0) = 234.56
    fld QWORD PTR [dblTwo]  ; ST(0) = 10.1, ST(1) = 234.56

    ; Store values from the stack to memory and pop
    fstp QWORD PTR [dblThree]  ; Stores 10.1 in dblThree and pops ST(0)
    fstp QWORD PTR [dblFour]   ; Stores 234.56 in dblFour and pops ST(0)
```

In this assembly example, you're using the **Floating-Point Unit (FPU)** to load values onto its stack and then store them back into memory with the **FST** and **FSTP** instructions. Let's walk through it step by step:

- **Defining the Data Variables**
    - ❖ dblOne and dblTwo are **double-precision floating-point variables** holding the values **234.56** and **10.1**, respectively.
    - ❖ dblThree and dblFour start off as **0.0** and are used later to store results from the FPU stack.
- **Loading Values with** fld
    - ❖ The fld instruction loads values onto the FPU stack.
    - ❖ First, dblOne (234.56) is loaded into **ST(0)**.
    - ❖ Then, dblTwo (10.1) is loaded into **ST(0)**, which pushes the previous value down—so now **ST(1) = 234.56**.

- **Storing with** fst
    - ❖ The fst instruction stores the value at the top of the stack (**ST(0)**) into memory **without popping it off**.
    - ❖ In this case, the value **10.1** is stored into both dblThree and dblFour.
    - ❖ The stack remains unchanged, so the values are still available for further operations.
- **Resetting for Demonstration**
    - ❖ To show the difference between fst and fstp, you reload dblOne and dblTwo onto the stack again.
- **Storing with** fstp
    - ❖ The fstp instruction stores the value from **ST(0)** into memory **and pops it off the stack** at the same time.
    - ❖ The first fstp stores **10.1** into dblThree and removes it from the stack.
    - ❖ The second fstp stores **234.56** into dblFour and pops that value too.
    - ❖ Each pop updates the stack's **TOP pointer**, shifting the remaining values upward.

V. Key Takeaway

This example highlights the difference between the two instructions:

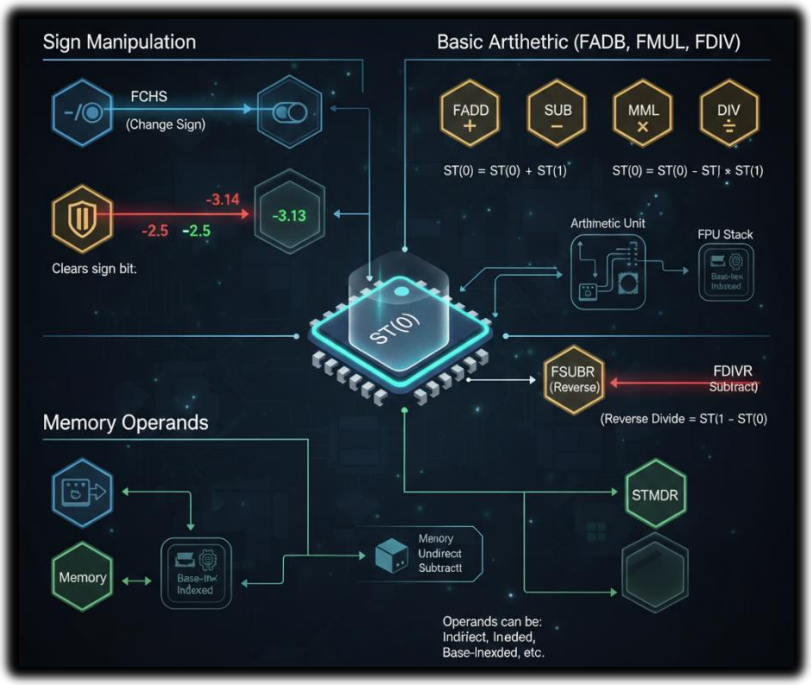
- **FST** → Stores the value in memory but leaves it on the stack.
- **FSTP** → Stores the value in memory and removes it from the stack.

Together, they give you flexible control over how floating-point values move between the FPU stack and memory—making them essential tools for efficient floating-point programming in assembly.

FLOATING POINT ARITHEMETIC

The arithmetic instructions in x86 assembly allow you to perform basic arithmetic operations on floating-point numbers. These instructions support the same memory operand types as **FLD (load)** and **FST (store)**, meaning operands can be **indirect**, **indexed**, **base-indexed**, and more. The key arithmetic instructions are as follows:

- **FCHS (Change Sign):** This instruction reverses the sign of the floating-point value in ST(0), effectively changing it from positive to negative or vice versa.
- **FABS (Absolute Value):** FABS clears the sign of the number in ST(0) to obtain its absolute value, effectively making it positive.
- **FADD (Add):** FADD performs addition. It can add two values from the FPU stack or add a value from memory to a value on the stack.
- **FSUB (Subtract):** FSUB subtracts the source value from the destination value.
- **FSUBR (Reverse Subtract):** FSUBR subtracts the destination value from the source value.
- **FMUL (Multiply):** FMUL multiplies the source value by the destination value.
- **FDIV (Divide):** FDIV divides the destination value by the source value.
- **FDIVR (Reverse Divide):** FDIVR divides the source value by the destination value.
- **FCHS and FABS:**
  - FCHS is used to change the sign of the value in ST(0), essentially negating it if it was positive or making it positive if it was negative.
  - FABS, on the other hand, computes the absolute value by clearing the sign of the value in ST(0).

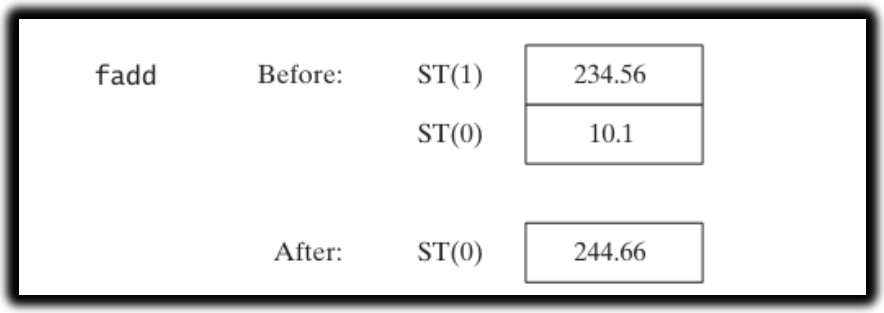


FADD (Add)

The **FADD** instruction performs floating-point addition, and it can be used in different formats.

I. With no operands:

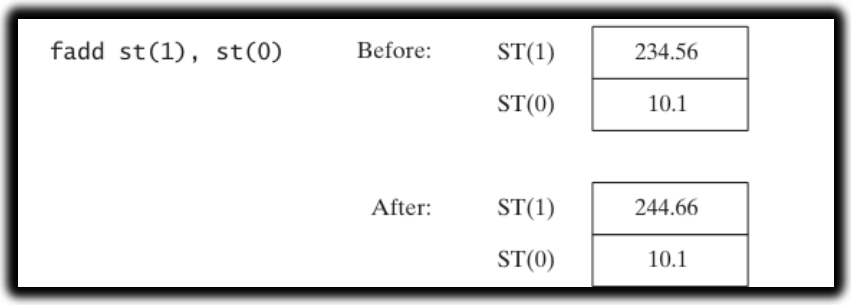
- It adds the value in **ST(0)** to **ST(1)**.
- The result is stored in **ST(1)**.
- Finally, **ST(0)** is popped off the stack, leaving the result at the top.



II. FADD (Add) – Other Formats

- **With a memory operand (REAL4 or REAL8)**
  - The instruction adds the value stored in memory (either a 32-bit REAL4 or a 64-bit REAL8) to the value currently in **ST(0)**.
  - The result replaces the value in **ST(0)**.
- **With a register number (ST(i))**
  - The instruction adds the value in **ST(i)** to the value in **ST(0)**.
  - Again, the result is stored back in **ST(0)**.

👉 In short: FADD can work directly with values in memory or with other registers in the FPU stack, giving you flexibility depending on where your data lives.



Register Format: FADD ST(i), ST(0)

- This version adds the value in **ST(0)** to the value in **ST(i)**.
- The result is stored in **ST(i)**, while **ST(0)** remains unchanged.
- In practice, this makes it look like the two positions have swapped roles, since ST(i) now holds the updated sum.

Memory Operand Format

- When FADD is used with a memory operand, it adds the value stored in memory directly to **ST(0)** (the top of the FPU stack).
- Example:
  - fadd mySingle → takes the value in the memory location mySingle and adds it to **ST(0)**.
  - The result replaces the old value in **ST(0)**, effectively increasing it by whatever was stored in mySingle.

```
.data
mySingle REAL4 5.5          ; Define a single-precision floating-point value in memory
myDouble REAL8 10.1         ; Define a double-precision floating-point value in memory
result   REAL8 0.0          ; Define a memory location to store the result

.code
main:
    fld     myDouble        ; Load myDouble onto the FPU stack
    fadd    mySingle        ; Add mySingle to ST(0)

    fstp    result          ; Store the result in the "result" memory location

    ; Exit the program (endless loop to prevent immediate termination)
    mov     eax, 1          ; Specify the exit system call
    int     0x80            ; Call the kernel

    ; Rest of the program here (not shown in this example)
```

- Two floating-point values, mySingle and myDouble, are defined in memory.
- fld loads myDouble onto the FPU stack (ST(0)).
- fadd adds mySingle to the value in ST(0).
- fstp stores the result into the memory location result.
- The program then exits.

Notes:

- The code assumes a **Linux environment** (uses int 0x80 system call).
- Adjust system calls and assembly syntax for **other OSes or assemblers** as needed

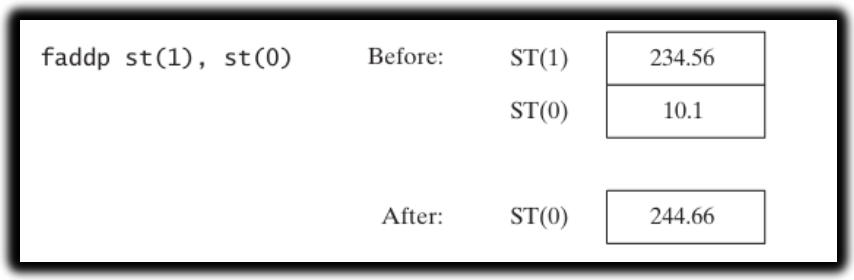
III. FADDP (Add with Pop)

FADDP ("Add with Pop"): Performs addition and then **pops ST(0)** from the FPU stack.

Purpose: Helps efficiently **manage the FPU stack** during calculations.

Syntax: FADDP ST(1), ST(0)

- Adds ST(0) to ST(1).
- Stores the result in ST(1).
- Removes ST(0) from the stack.



IV. FIADD (Add Integer)

FIADD: Adds an **integer value** to ST(0) after converting it to **double-extended precision floating-point**.

Supported operand types:

- m16int → 16-bit integer
- m32int → 32-bit integer

Example: fiadd myInteger

- Converts the integer at myInteger to floating-point
- Adds it to the value in ST(0)

```
.data
myInteger DWORD 1        ; Define a 32-bit integer value in memory
result REAL8 0.0         ; Define a memory location to store the result

.code
main:
    finit                ; Initialize the FPU
    fild    myInteger    ; Load myInteger onto the FPU stack and convert it to a floating-point value
    fadd                 ; Add ST(0) to ST(1), effectively adding myInteger to ST(0)

    fstp    result       ; Store the result in the "result" memory location

    ; Exit the program (endless loop to prevent immediate termination)
    mov     eax, 1       ; Specify the exit system call
    int     0x80         ; Call the kernel

    ; Rest of the program here (not shown in this example)
```

In this code:

- Define an **integer value** myInteger in memory and a memory location result to store the result.
- Use finit to **initialize the FPU**.
- Load myInteger onto the FPU stack and **convert it to floating-point** using fild.
- Use fadd to **add the value in ST(0) to ST(1)** (effectively adding myInteger to the existing floating-point value).
- Store the **result** in memory using fstp into result.

Note:

- Uses int 0x80 system call for Linux.
- Adjust system calls and assembly syntax for **Windows or other platforms**.

V. FSUB (Floating-Point Subtract)

- **Purpose:** Subtracts a **source operand** from a **destination operand** in the FPU.
- **Destination:** Always ST(0) (top of stack).
- **Source:** Can be an **FPU register** or **memory operand** (same types as FADD).
- **Operation:** Similar to FADD, but performs **subtraction**.

Usage without operands:

- Subtracts ST(0) from ST(1).
- Result is stored in ST(1).
- ST(0) is **popped**, leaving the result on top.

Examples:

- fsub mySingle → ST(0) -= mySingle
- fsub array[edi*8] → subtracts value from memory from ST(0) (stack not popped)

VI. FSUBP (Floating-Point Subtract with Pop)

- **Purpose:** Performs subtraction **and pops ST(0)** from the stack.
- After execution:
  - Result is left in **ST(0)**
  - Stack pointer is **adjusted** accordingly

```
.data
mySingle REAL8 5.0           ; Example value for mySingle
array REAL8 10.0              ; Example value for array[edi*8]

.code
; Subtract mySingle from ST(0) using FSUB
fld QWORD PTR [mySingle]     ; Load mySingle onto the FPU stack
fsub                         ; Subtract mySingle from ST(0)
; The result is now in ST(0)

; Subtract array[edi*8] from ST(0) using FSUB
fld QWORD PTR [array+edi*8]  ; Load array[edi*8] onto the FPU stack
fsub                         ; Subtract array[edi*8] from ST(0)
; The result is now in ST(0)

; Now, let's use FSUBP to subtract and pop ST(0)

; Subtract 5.0 from ST(0) and pop the stack
fld QWORD PTR [mySingle]     ; Load mySingle onto the FPU stack
fsubp                        ; Subtract mySingle from ST(0) and pop ST(0)
; The result is left in ST(0), and ST(0) is removed from the stack
```

Example: FSUBP

- fsubp ST(1), ST(0)
  - Subtracts ST(0) from ST(1).
  - Leaves the **result in ST(0)**.
  - Pops the **old ST(0)** from the stack.

VII. FISUB (Floating-Point Subtract Integer)

- Subtracts an **integer** from the value in ST(0).
- Converts the **integer source operand** to **double-extended precision floating-point** before subtraction.
- Enables **arithmetic between integers and floating-point numbers** in the FPU.

```
.data
myInteger DWORD 3      ; Example integer value for myInteger

.code
; Subtract an integer from ST(0) using FISUB
fld QWORD PTR [myInteger]   ; Load myInteger as a 64-bit integer onto the FPU stack
fisub                       ; Subtract myInteger from ST(0)
; The result is now in ST(0)

; Use FSUBP to subtract ST(0) from ST(1) and pop ST(0)
fsubp ST(1), ST(0)
; After executing this, the result is in ST(0), and the old ST(0) has been removed from the stack
```

FISUB and FSUBP Examples

FISUB (Floating-Point Subtract Integer):

- Subtracts an **integer** from ST(0) after converting it to **double-extended precision floating-point**.
- Examples:
  - ❖ fisub m16int → subtracts a **16-bit integer** from ST(0).
  - ❖ fisub m32int → subtracts a **32-bit integer** from ST(0).

FSUBP (Floating-Point Subtract with Pop):

- fsubp ST(1), ST(0)
  - ❖ Subtracts ST(0) from ST(1).
  - ❖ Stores the **result in ST(0)**.
  - ❖ Pops the **old ST(0)** from the stack.

```
.data
    my16BitInt WORD 5    ; Example 16-bit integer value
    my32BitInt DWORD 10  ; Example 32-bit integer value

.code
    ; Subtract a 16-bit integer from ST(0) using FISUB
    fld QWORD PTR [my16BitInt]  ; Load my16BitInt as a 16-bit integer onto the FPU stack
    fisub                       ; Subtract my16BitInt from ST(0)
    ; The result is now in ST(0)

    ; Reset ST(0) to another value
    fld QWORD PTR [my32BitInt]  ; Load my32BitInt as a 32-bit integer onto the FPU stack

    ; Subtract a 32-bit integer from ST(0) using FISUB
    fisub                      ; Subtract my32BitInt from ST(0)
    ; The result is now in ST(0)
```

These code examples demonstrate how to use FISUB to subtract 16-bit and 32-bit integers from an FPU register after converting them to floating-point format.

This provides flexibility for performing arithmetic operations with different data types and operands in floating-point arithmetic.

VIII. FMUL (Floating-Point Multiply)

- **Purpose:** Multiplies a **source operand** by a **destination operand** (always an FPU register).
- **Source operand:** Can be an **FPU register** or **memory operand** (like FADD/FSUB).

FMUL usage:

- **No operands:**
  - Multiplies ST(0) by ST(1).
  - Result temporarily stored in ST(1).
  - ST(0) is **popped**, leaving the product in ST(0).
- **With memory operand:**
  - Multiplies ST(0) by the value in memory.

FMULP (Floating-Point Multiply with Pop):

- Similar to FMUL, but **pops ST(0)** after multiplication.
- Result is left in ST(0); stack pointer is adjusted.

FIMUL (Floating-Point Integer Multiply):

- Works like FIADD, but performs **multiplication** instead of addition.
- Converts **integer source operand** to **double-extended precision floating-point** before multiplying.

```
.data
    mySingle REAL4 3.0          ; Example single-precision floating-point value
    my16BitInt WORD 2           ; Example 16-bit integer
    my32BitInt DWORD 4          ; Example 32-bit integer
    result REAL8 0.0            ; Storage for results

.code
    ; Multiply ST(0) by a single-precision floating-point value
    fld QWORD PTR [mySingle]   ; Load mySingle as a single-precision value onto the FPU stack
    fmul                      ; Multiply ST(0) by mySingle
    fstp QWORD PTR [result]  ; Store the result in result

    ; Reset ST(0) to another value
    fld QWORD PTR [my32BitInt]  ; Load my32BitInt as a 32-bit integer onto the FPU stack

    ; Multiply ST(0) by another value
    fld QWORD PTR [mySingle]   ; Load another single-precision value onto the FPU stack
    fmul                      ; Multiply ST(0) by the new single-precision value
    fstp QWORD PTR [result]  ; Store the result in result

    ; Reset ST(0) to another value
    fld QWORD PTR [my16BitInt]  ; Load my16BitInt as a 16-bit integer onto the FPU stack

    ; Multiply ST(0) by another value
    fld QWORD PTR [mySingle]   ; Load another single-precision value onto the FPU stack
    fimul WORD PTR [esp]     ; Multiply ST(0) by the 16-bit integer
    fstp QWORD PTR [result]  ; Store the result in result
```

Variable Initialization:

.data section defines variables:

- mySingle → single-precision float (3.0)
- my16BitInt → 16-bit integer (2)
- my32BitInt → 32-bit integer (4)
- result → stores results of multiplications

Single-Precision Floating-Point Multiplication:

- fld mySingle → loads mySingle onto FPU stack (ST(0)).
- fmul → multiplies ST(0) by itself.
- fstp result → stores the result in memory (result).

Resetting the FPU Stack:

- fld my32BitInt → loads a new value onto ST(0) for the next multiplication.

Second Single-Precision Multiplication:

- fld → loads another single-precision value into ST(0).
- fmul → multiplies the value in ST(0) by the new single-precision value.
- fstp result → stores the result in memory.

Resetting FPU Stack Again: fld my16BitInt → prepares stack for integer multiplication.

Integer Multiplication with FIMUL:

- fld → loads a 16-bit integer onto ST(0).
- fld → loads a single-precision float onto ST(0).
- fimul WORD PTR [esp] → multiplies the integer in ST(0) with the float.
- fstp result → stores the result in memory.

Completion:

- Result of integer multiplication is stored in result.
- Demonstrates **different multiplication scenarios** in x86 FPU:
  - Floating-point × floating-point
  - Integer × floating-point
  - Using fld, fmul, fimul, and fstp for stack management and result storage

IX. FDIV (Floating-Point Division)

- **Purpose:** Divides the **destination operand** by the **source operand**, result stored in the destination.
- **Operands:**
  - Destination → must be an **FPU register** (ST(0)).
  - Source → can be an **FPU register** or **memory operand**.
- **Syntax examples:**
  - fdiv ST(0), ST(i) → divides ST(i) by ST(0)
  - fdiv m64fp → divides ST(0) by a 64-bit float in memory
- **Special cases:**
  - Division by **zero** → divide-by-zero exception
  - Division by **infinity**, **zero**, or **NaN** → handled according to Intel rules

X. FIDIV (Floating-Point Divide Integer)

- **Purpose:** Performs division of an **integer by a floating-point value** in ST(0).
- **Conversion:** Source integer is converted to **double-extended precision floating-point** before division.
- **Supported integer types:**
  - m16int → 16-bit integer
  - m32int → 32-bit integer
- **Use case:** Enables **mixed integer and floating-point arithmetic** in the FPU.
- **Code examples:** Can demonstrate division with:
  - FPU registers
  - Memory operands
  - Mixed integer/floating-point operands

```
.data
    dblOne REAL8 1234.56
    dblTwo REAL8 10.0
    intDivisor WORD 2
    quotient REAL8 ?

.code
    ; Single-Precision Floating-Point Division
    fld dblOne        ; Load dblOne onto the FPU stack
    fdiv dblTwo       ; Divide ST(0) by dblTwo
    fstp quotient     ; Store the result in quotient

    ; Reset the FPU stack
    fld dblOne        ; Load dblOne again

    ; Double-Precision Floating-Point Division
    fdiv dblTwo       ; Divide ST(0) by dblTwo
    fstp quotient     ; Store the result in quotient

    ; Integer Division Using FIDIV
    fld intDivisor    ; Load the integer divisor onto the FPU stack
    fidiv intDivisor  ; Divide ST(0) by intDivisor
    fstp quotient     ; Store the result in quotient
```

Variable Initialization (.data section):

- dblOne → double-precision float (first dividend)
- dblTwo → double-precision float (divisor)
- intDivisor → integer divisor
- quotient → stores the results of division operations

Single-Precision Floating-Point Division:

- fld dblOne → loads dblOne onto the FPU stack (ST(0))
- fdiv dblTwo → divides ST(0) by dblTwo
- fstp quotient → stores the result in memory

Double-Precision Floating-Point Division:

- fld dblOne → reloads dblOne onto the stack
- fdiv dblTwo → divides ST(0) by dblTwo with double precision
- fstp quotient → stores the result

Integer Division with FIDIV:

- fld intDivisor → loads the integer onto the FPU stack
- fidiv → divides ST(0) by the integer (converted to floating-point)
- fstp quotient → stores the result

Summary:

- Demonstrates **various division scenarios** in the x86 FPU:
  - Single-precision floating-point division
  - Double-precision floating-point division
  - Integer division
- Results are consistently stored in the **quotient variable**
- Provides a **comprehensive example** of FPU division operations in assembly language

COMPARING FLOATING-POINT VALUES

- **FCOM (Floating-Point Compare):**
  - Compares ST(0) (top of FPU stack) to a **source operand** (memory or FPU register).
- **FNSTSW (Store FPU Status Word):**
  - Moves the **FPU status word** into AX.
  - Status word contains **condition codes** indicating the comparison result (greater, less, equal, unordered).
- **SAHF (Store AH into Flags):**
  - Copies AH (upper byte of AX) into **EFLAGS** register.
  - EFLAGS now has the CPU **Zero, Parity, Carry, and Sign flags** reflecting the comparison.
- **Conditional Jumps:**
  - Use CPU flags from EFLAGS to **branch** based on comparison.
  - Example: branch to greater_than if ST(0) > ST(1).

```
; Load the value in ST(0) to ST(0) (no change)
fld st(0)

; Load the value in ST(1) to ST(0)
fld st(1)

; Compare ST(0) to ST(1) and set FPU condition codes
fcom

; Move the FPU status word into the AX register
fnstsw ax

; Copy the AH register (containing FPU condition codes) to EFLAGS
sahf

; Check if the result of the comparison was greater (JG stands for "jump if greater")
jg greater_than

; Your code for handling the case where the comparison result was not greater would go here

greater_than:
; Your code for handling the case where the comparison result was greater goes here
```

- **greater_than label:**
  - Marks the location to continue execution if ST(0) > ST(1).
- **Handling the "not greater" case:**
  - Place code in a designated section for when the comparison result is **≤**.
- **Code sequence:**
  - **Load values** onto the FPU stack (ST(0) and ST(1)).
  - **FCOM** → compares ST(0) with the source operand (ST(1) or memory).
  - **FNSTSW AX** → moves FPU status word into AX (contains condition codes).
  - **SAHF** → copies relevant bits from AH into **EFLAGS**.
  - **JG greater_than** → jumps to greater_than if comparison result indicates ST(0) > ST(1).
- **Summary:**
  - This sequence enables **floating-point comparisons** to control program flow.
  - It bridges **FPU status codes** to **CPU conditional jumps**.

The following table shows the condition codes and the corresponding conditional jump instructions:

Condition code	Description	Conditional jump instruction
CF = 1 and ZF = 0	ST(0) is greater than the source operand.	JG
CF = 0 and ZF = 0	ST(0) is less than the source operand.	JL
CF = 0 and ZF = 1	ST(0) is equal to the source operand.	JE
CF = 1 and ZF = 1	The comparison is unordered.	JAE

Floating-Point Comparisons (Handling NaN & Unordered Cases)

Unordered comparison:

- Occurs if **either operand is NaN**.
- Also occurs if **operands have different signs** and **one is zero**.

Importance:

- Comparing floating-point values and branching is critical for:
  - Graphics
  - Scientific computing
  - Financial modeling

FCOMI Instruction

Purpose:

- Compares two floating-point values **directly**.
- Sets **CPU flags** (Zero, Parity, Carry) based on comparison.

Advantages over FCOM + FNSTSW + SAHF:

- Eliminates the need to:
  - Store FPU status word in AX
  - Copy AH into EFLAGS
- Flags are set **directly**, simplifying branching logic.

Usage:

- Pass the two floating-point values as operands.
- CPU flags reflect comparison:
  - **ZF** → Zero flag (ST(0) = ST(i))
  - **PF** → Parity flag (unordered, e.g., NaN)
  - **CF** → Carry flag (ST(0) < ST(i))

Summary:

- FCOMI is **modern and efficient** for floating-point comparisons on Pentium-class (P6) and later processors.
- Useful for **conditional branching** without extra instructions.

Unordered Condition in FPU Comparisons

The following table shows the condition codes set by the FCOMI instruction:

Condition	C3 (Zero Flag)	C2 (Parity Flag)	C0 (Carry Flag)	Conditional Jump to Use
ST(0) > SRC	0	0	0	JA, JNBE
ST(0) < SRC	0	0	1	JB, JNAE
ST(0) = SRC	1	0	0	JE, JZ
Unordered[a]	1	1	1	(*None*)

An unordered condition occurs in floating-point comparisons when either operand is a NaN, or when the operands have different signs and one of them is zero.

In this case, the condition codes are set to **111**, meaning all three flags (CF, ZF, and PF) are equal to 1.

There is no standard conditional jump instruction that directly corresponds to this code.

After using the **FCOMI** instruction, conditional jumps can only be applied if the comparison result is ordered.

If the result is unordered, special handling is required—for example, checking the **PF flag** to detect the presence of a NaN.

For example, the following code branches to the L1 label if the value in ST(0) is less than the value in ST(1):

```
330 ; Load the value of Y onto the FPU stack
331 fld Y           ; ST(0) = Y
332
333 ; Load the value of X onto the FPU stack, creating a stack with X on top and Y below
334 fld X           ; ST(0) = X, ST(1) = Y
335
336 ; Compare the values on the FPU stack (X and Y)
337 fcomi ST(0), ST(1) ; Compare ST(0) to ST(1)
338
339 ; Jump to label L1 if not below (if X is not less than Y), skipping the next instruction
340 jnb L1          ; Jump if not below (ST(0) not < ST(1)?), skip to L1
341
342 ; If the jump condition is not met, set the integer N to 1
343 mov N, 1        ; N = 1
344
345 ; Label L1 for reference
346 L1:
```

fld Y – This instruction grabs the value of Y (our second floating-point number) and puts it on the FPU stack. After this, ST(0) holds Y.

fld X – Next, we load X (the first floating-point number) onto the stack. This pushes Y down to ST(1) and puts X on top at ST(0). Now our stack is ready for a comparison.

fcomi ST(0), ST(1) – Here's where the magic happens! This instruction compares X (ST(0)) with Y (ST(1)). If X is **not less than** Y, it sets the carry flag (CF) to 1.

jnb L1 – This checks that carry flag. "Jump if not below" basically means: if X is **not less than** Y, skip the next instruction and go straight to label L1.

mov N, 1 – If the comparison shows that X **is less than** Y, this instruction sets the integer N to 1. So N becomes 1 only when the condition is true.

L1: – This is just a marker in the code. It marks the end of this little conditional block so the program knows where to continue.

In short: This snippet compares two floating-point numbers, X and Y, and sets N to 1 if X is smaller than Y. It's a neat example of using the FPU's condition codes—especially the carry flag—to make decisions in your program.

FCOM Instruction

FCOM is an instruction that compares two floating-point numbers on the FPU stack and updates the condition flags based on the outcome.

Here's a simple example: we have two double-precision numbers, X and Y, and we want to compare them. If X is **less than** Y, we'll set an integer N to 1. It's a neat way to use the FPU to make decisions in your code!

```
350 .data
351     X REAL8 1.2
352     Y REAL8 3.0
353     N DWORD 0
354
355 .code
356     ; if( X < Y )
357     ;
358     N = 1
359     fld X   ; Load X into ST(0)
360     fcomp Y   ; Compare ST(0) to Y
361     fnstsw ax   ; Move status word into AX
362     sahf   ; Copy AH into EFLAGS
363     jnb L1   ; X not < Y? Skip
364     mov N, 1   ; N = 1
365     L1:
```

This code snippet shows how **FPU condition codes** are set and how **conditional jumps** control program flow based on comparison results.

A useful improvement is using the **FCOMI instruction**, which is available on newer processors like the Intel P6 family (Pentium Pro, Pentium II).

FCOMI is more efficient because it **directly compares two values** on the FPU stack and sets the condition flags, making comparisons faster than FCOM.

With FCOMI, you can perform floating-point comparisons and **set the Zero, Parity, and Carry flags** directly.

Here's an example of performing the same comparison using **FCOMI**.

```
.data
    X REAL8 1.2
    Y REAL8 3.0
    N DWORD 0

.code
    ; if( X < Y )
    ;
    N = 1
    fld Y   ; Load Y into ST(0)
    fld X   ; Load X into ST(0), Y is now in ST(1)
    fcomi ST(0), ST(1)   ; Compare ST(0) to ST(1)
    jnb L1   ; ST(0) not < ST(1)? Skip
    mov N, 1   ; N = 1
    L1:
```

Comparing For Equality

Comparing for equality in assembly means checking if two floating-point values are "close enough" rather than exactly equal.

The standard approach is to compute the **absolute difference**, |x - y|, and see if it's smaller than a tiny number called **epsilon**.

Why? Because **floating-point numbers are approximate**, and tiny rounding errors can make direct comparisons unreliable.

The following assembly code shows how to compare two values, val2 and val3, using a tolerance of **epsilon**.

```
.data
    epsilon REAL8 1.0E-12   ; Define epsilon as the tolerance
    val2 REAL8 0.0           ; Define val2 as the first value to compare
    val3 REAL8 1.001E-13    ; Define val3 as the second value, considered equal to val2

.code
    fld epsilon              ; Load epsilon into the FPU stack
    fld val2                 ; Load val2 into the FPU stack
    fsub val3                ; Subtract val3 from val2
    fabs                    ; Compute the absolute value of the difference
    fcomi ST(0), ST(1)       ; Compare ST(0) to ST(1) and set condition flags
    ja skip                  ; Jump if the absolute difference is greater
    ; If we reach here, the values are considered equal
    mWrite <"Values are equal", 0dh, 0ah>  ; Display "Values are equal"
    skip:
```

This assembly code compares two floating-point values, val2 and val3, for equality **within a small tolerance** called epsilon.

Why? Floating-point numbers aren't exact due to precision limits, so direct comparisons can fail. Using a tolerance ensures "close enough" values are treated as equal.

Data Section (.data):

- epsilon = 1.0E-12 → the maximum difference allowed to consider two values equal.
- val2 = 0.0 → first value to compare.
- val3 = 1.001E-13 → second value, slightly different from val2.

Code Section (.code):

- fld epsilon → loads the tolerance onto the FPU stack.
- fld val2 → loads the first value (val2).
- fsub val3 → subtracts val3 from val2 to get the difference.
- fabs → converts the difference to its absolute value.

- fcomi ST(0), ST(1) → compares the absolute difference with epsilon and sets condition flags.
- ja skip → jumps to skip if the difference is greater than epsilon (values are **not equal**).
- mWrite <"Values are equal", 0dh, 0ah> → prints a message if the difference is within the tolerance.
- skip: → label to continue execution when the values are not equal.

Summary:

- This snippet shows a safe way to compare floating-point numbers, accounting for rounding errors.
- Using epsilon makes the comparison robust for real-world calculations.

FLOATING-POINT INPUT/OUTPUT PROCEDURES

Assembly doesn't have "nice" built-in input/output like high-level languages, so special procedures are used to handle floating-point numbers. Two key ones are:

- **ReadFloat**
  - Reads a floating-point value typed in from the keyboard.
  - Pushes that value directly onto the **FPU stack**.
  - It's flexible — it understands lots of formats, like:
    - 35
    - +35.
    - -3.5
    - .35
    - 3.5E5 or 3.5E005
    - -3.5E+5
    - 3.5E-4 or +3.5E-4
- **WriteFloat**
  - Takes the floating-point value at the top of the FPU stack (**ST(0)**).
  - Prints it to the console in **exponential format** (scientific notation).

What the Example Program Does

The demo program ties these procedures together:

1. Pushes two floating-point values onto the FPU stack.
2. Displays the current contents of the stack.
3. Asks the user to type in two floating-point numbers.
4. Multiplies those two numbers together.
5. Displays the product back to the console using WriteFloat.

👉 So in short: **ReadFloat** is your input tool, **WriteFloat** is your output tool, and the program shows how to use them with the FPU stack to do actual math.

```
; 32-bit Floating-Point I/O Test (floatTest32.asm)
INCLUDE Irvine32.inc
INCLUDE macros.inc
.data
    first REAL8 123.456
    second REAL8 10.0
    third REAL8 ?

.code
    main PROC
        finit                   ; Initialize FPU
        ; Push two floats and display the FPU stack.
        fld first               ; Push the first value onto the FPU stack.
        fld second              ; Push the second value onto the FPU stack.
        call ShowFPUStack       ; Display the FPU stack.
        ; Input two floats and display their product.

        mWrite "Please enter a real number: "
        call ReadFloat          ; Read the first floating-point number.
        mWrite "Please enter a real number: "
        call ReadFloat          ; Read the second floating-point number.

        fmul                    ; Multiply ST(0) by ST(1).

        mWrite "Their product is: "
        call WriteFloat         ; Display the product.
        call Crlf               ; Add a line break.

        exit
    main ENDP
END main
```

What the Program Really Does

1. **Initialize the FPU** → Get the floating-point unit ready.
2. **Load values** → Push two numbers onto the FPU stack.
3. **Show the stack** → Display what's currently inside the stack.
4. **Take user input** → Ask for two floating-point numbers and push them onto the stack.
5. **Multiply** → Use fmul to multiply the two numbers.
6. **Display result** → Print "Their product is:" followed by the result in exponential format.
7. **Exit cleanly** → Add a line break and finish.

👉 So in one line: **The program reads two floating-point numbers, multiplies them using the FPU, and prints the result.**

EXCEPTION SYNCHRONIZATION

In concurrent systems, multiple tasks can run at the same time. This creates a special challenge for **floating-point exceptions**, because the **CPU** and the **FPU (Floating-Point Unit)** operate as separate units.

- **The Problem** - If an unmasked floating-point exception occurs while the CPU is busy executing an integer or system instruction, the exception won't be handled right away. Instead, it will only be processed when the next floating-point instruction runs—or when a **WAIT/FWAIT** instruction is explicitly used.
- **Why This Matters** - This delay can cause trouble if the floating-point instruction that triggered the exception is immediately followed by an integer or system instruction that modifies the same memory operand. In that case, the CPU might change memory before the floating-point exception is even recognized, leading to inconsistent or incorrect results.
- **Example Scenario -** Imagine code where a floating-point operation causes an exception, but the very next instruction is an integer operation that writes to the same memory location. Without synchronization, the exception handling gets delayed, and the memory update happens first—creating a conflict.

👉 The **WAIT** and **FWAIT** instructions are the solution here. They force the CPU to pause until the FPU has finished its work and any pending exceptions are handled, ensuring proper synchronization between the two units.

```
.data
    intVal DWORD 25    ; Define an integer value

.code
    fild intVal        ; Load the integer value into ST(0)
    inc intVal         ; Increment the integer value
```

This code **loads an integer value (intVal) onto the FPU stack** (ST(0)) using the FILD instruction and then **increments the original integer in memory** with INC.

Important note: INC changes the value in memory, not the value on the FPU stack.

If an **unmasked floating-point exception** occurs during FILD, it won't be handled until INC runs. This can cause problems if INC modifies the same memory location—then the exception handler might see the wrong value.

To fix this, you can use **WAIT or FWAIT**:

- These instructions **pause execution until any pending floating-point exceptions are handled**.
- This ensures the exception handler gets a chance to run **before any other instructions modify the memory operand**.

In practice, you'd insert a WAIT or FWAIT **right after FILD**, so the FPU exception is handled before the INC instruction.

```
.data
    intVal DWORD 25

.code
    ; Load the integer into ST(0)
    fild intVal

    ; Wait for pending exceptions
    fwait

    ; Increment the integer
    inc intVal
```

In this version, FILD loads the integer value into the **FPU stack**.

FWAIT comes next, making sure any **pending floating-point exceptions** are handled before moving on.

INC then increments the integer in memory.

This sequence ensures the **exception handler gets a chance to run** before the memory value is modified, keeping everything safe and predictable.

Example Code For FPU Operations:

Here's the code for the expression **valD = -valA + (valB * valC)**

```
.data
    valA REAL8 1.5
    valB REAL8 2.5
    valC REAL8 3.0
    valD REAL8 ? ; Initialize valD as a placeholder for the result

.code
    ; Load valA on the FPU stack and negate it
    fld valA        ; ST(0) = valA
    fchs            ; Negate the value in ST(0)

    ; Load valB into ST(0) and multiply by valC
    fld valB        ; Load valB into ST(0)
    fmul valC       ; Multiply ST(0) by valC, leaving the product in ST(0)

    ; Add the two values on the stack (ST(0) and ST(1))
    fadd            ; Add ST(0) and ST(1), result in ST(0)

    ; Store the result in valD
    fstp valD       ; Store the result in valD
```

This code calculates valD using the expression **-valA + (valB * valC)**.

First, valA is loaded onto the FPU stack and **negated** with fchs.

Next, valB is loaded onto the stack and **multiplied by valC**, leaving the product in ST(0).

The two values on the stack are **added together**.

Finally, fstp **stores the result in valD** and **removes it from the stack**.

Sum Of Array Of Double Precision Numbers

In the code below, you are calculating the sum of an array of double-precision real numbers.

```
510 ; Define the size of the array
511 ARRAY_SIZE = 20
512
513 .data
514     sngArray REAL8 ARRAY_SIZE DUP(?)
515
516 .code
517     mov esi, 0          ; Initialize array index
518     fldz               ; Push 0.0 onto the FPU stack
519     mov ecx, ARRAY_SIZE ; Set the loop counter to ARRAY_SIZE
520
521     L1:
522     fld sngArray[esi]  ; Load the current element into ST(0)
523     fadd               ; Add ST(0) to the accumulator (ST(0)), and pop
524     add esi, TYPE REAL8 ; Move to the next element
525     loop L1            ; Continue the loop until all elements are processed
526
527     call WriteFloat    ; Display the sum in ST(0)
```

This code calculates and displays the sum of a double-precision array.

The array size is defined by the constant ARRAY_SIZE = 20, making it easy to adjust the array length in one place.

We start by initializing a loop counter (esi = 0) and pushing 0.0 onto the FPU stack (fldz) to serve as our sum accumulator.

Another counter, ecx, is set to ARRAY_SIZE to control the loop.

The loop (L1) does the following for each array element:

Load the current element with fld sngArray[esi].

Add it to the accumulator using fadd, which also pops the value from the stack.

Move to the next element with add esi, TYPE REAL8 (8 bytes per double).

The loop repeats until all elements are processed.

After the loop, WriteFloat is called to display the sum stored in ST(0).

Calculating The Sum Of Squares Of Two Numbers

In this code, we are calculating the sum of the square roots of two numbers, valA and valB. Here are the steps involved:

We define two real numbers, valA and valB, which have the values 25.0 and 36.0, respectively.

```
.data
    valA REAL8 25.0
    valB REAL8 36.0

.code
    fld valA    ; Load valA onto the FPU stack
    fsqrt       ; Replace ST(0) with the square root of valA
    fld valB    ; Load valB onto the FPU stack
    fsqrt       ; Replace ST(0) with the square root of valB
    fadd        ; Add the two square roots and leave the result in ST(0)
```

This code calculates the **sum of the square roots** of valA and valB using FPU instructions.

Step by step:

- fld valA → loads valA onto the FPU stack (ST(0)).
- fsqrt → replaces ST(0) with the **square root of valA**.
- fld valB → loads valB onto the stack (ST(0)).
- fsqrt → computes the **square root of valB**.
- fadd → adds the two square roots, leaving the **sum in ST(0)**.

In short, the FPU stack is used efficiently to **load values, compute square roots, and add them**, giving the final result in ST(0).

Calculating The Dot Product Of Two Pairs Of Numbers

Here, you have a code snippet that calculates the dot product of two pairs of numbers from an array. The input data contains two pairs of numbers stored in the "array" variable.

```
.data
    array REAL4 6.0, 2.0, 4.5, 3.2

.code
    fld dword ptr [array]     ; Load the first number of the first pair
    fmul dword ptr [array+4]  ; Multiply with the second number of the first pair
    fld dword ptr [array+8]   ; Load the first number of the second pair
    fmul dword ptr [array+12]  ; Multiply with the second number of the second pair
    fadd                      ; Add the two products in ST(0)
```

This code calculates the dot product of two pairs of numbers from the array.

It multiplies each pair of elements together and then adds the results.

The final value, stored in ST(0) , is the sum of the products, giving the dot product.

MIXED-MODE ARITHMETIC

Mixed-mode arithmetic is just a fancy way of saying arithmetic that mixes **integers** and **real numbers** (floating-point numbers).

In assembly language, the Intel instruction set has some handy instructions that help with this. Basically, it can **promote integers to reals** and put them on the floating-point stack so you can do your calculations without any fuss.

Step 1: Promote an integer to a real

Before you can do arithmetic with a real number, you need to convert your integer. For example:

```
fild N
```

This instruction takes the integer in the N register, turns it into a real, and loads it onto the floating-point stack. Easy, right?

Step 2: Do your arithmetic

Once your number is on the floating-point stack, you can do all your floating-point operations just like normal.

Step 3: Store the result back as an integer

After you're done with the arithmetic, you might want to store the result as an integer. That's where FIST comes in:

```
fist Z
```

This stores the real number in ST(0) back into the integer variable Z. One important thing to remember: FIST **rounds to the nearest integer**.

If you want to **truncate** instead of round, there's a way to do that too (we'll get into the code for that next).

```
fstcw ctrlWord      ; Store the FPU control word in ctrlWord
or ctrlWord, 110000000000b ; Set RC (Rounding Control) to truncate
fldcw ctrlWord      ; Load the modified control word to change the rounding mode

; Perform arithmetic operations here using the modified rounding mode
fild N              ; Load the integer N into ST(0)
fadd X              ; Add the real X to ST(0)
fist Z              ; Store the truncated result in integer Z

fstcw ctrlWord      ; Store the FPU control word in ctrlWord again
and ctrlWord, 001111111111b ; Reset the rounding mode to default (round to nearest)
fldcw ctrlWord      ; Load the control word to restore the default rounding mode
```

- Store the FPU control word in the variable ctrlWord.
- Modify the RC field in ctrlWord to set the rounding mode to **truncate**.
- Perform the desired arithmetic operations.
- Store the FPU control word again.
- Reset the rounding mode to the default (**round to nearest**) by using a **bitwise AND** to clear the rounding mode bits.
- Load the modified control word to ensure the default rounding mode is restored.

Example 1: Adding an Integer to a Double

In this example, you're adding an integer (N) to a double (X) and storing the result in a double (Z).

The C++ code automatically promotes the integer to a real before performing the addition. The equivalent assembly code for this operation is as follows:

```
.data
    N SDWORD 20
    X REAL8 3.5
    Z REAL8 ?

.code
    fild N    ; Load the integer into ST(0)
    fadd X    ; Add the memory value (X) to ST(0)
    fstp Z    ; Store ST(0) to memory (Z)
```

This code loads the integer N into ST(0), adds the real value X, and stores the result in the real Z.

Example 2: Promoting and Truncating

In this example, you're promoting an integer (N) to a double and evaluating a real expression involving N and X.

The result is stored in an integer (Z). C++ typically performs the conversion automatically, but in assembly, you use FIST to convert the result to an integer. Here's the code:

```
607 fild N    ; Load the integer into ST(0)
608 fadd X    ; Add the real X to ST(0)
609 fist Z    ; Store ST(0) to the integer Z
```

This code loads the integer N into ST(0), adds the real X, and then stores the truncated result in the integer Z.

Changing the Rounding Mode

The FPU (Floating-Point Unit) has a control word, and the **RC field** in this control word lets you choose how numbers get rounded.

Here's how it works:

- Use **FSTCW** to store the current control word in a variable.
- Change the **RC field** in that variable to pick your rounding mode—like **truncate** or **round to nearest**.
- Use **FLDCW** to load your modified control word back into the FPU.

Once you do this, all floating-point operations will follow the rounding mode you set. It's a neat way to control exactly how your numbers are rounded!

Example 3:

```
INCLUDE Irvine32.inc
.data
    N           SDWORD 20       ; Integer value
    X           REAL8 3.5        ; Double-precision real value
    Z           REAL8 ?         ; Result stored as a double-precision real
    ctrlWord    WORD 0          ; FPU control word

.code
    main PROC
        ; Initialize FPU
        finit

        ; Store the FPU control word
        fstcw ctrlWord
        ; Modify the control word to set the rounding mode to truncate
        or ctrlWord, 110000000000b
        fldcw ctrlWord
        ; Perform mixed-mode arithmetic
        fild N          ; Load integer N into ST(0)
        fadd X          ; Add real X to ST(0)
        fstp Z          ; Store the result in Z
        ; Display the result
        mov edx, OFFSET Z
        call WriteFloat
        ; Reset the rounding mode to default (round to nearest)
        and ctrlWord, 001111111111b
        fldcw ctrlWord
        exit
    main ENDP
END main
```

Initialization:

- Include Irvine32.inc for handy I/O and utility functions.
- Define variables:
  - N SDWORD 20 — signed integer 20
  - X REAL8 3.5 — double-precision real 3.5
  - Z REAL8 ? — double-precision real to store the result
  - ctrlWord WORD 0 — stores the FPU control word

FPU Setup:

- finit — initializes the FPU for a clean start
- fstcw ctrlWord — save the current FPU control word for modification

Rounding Mode:

- or ctrlWord, 110000000000b — sets FPU rounding to **truncate**
- Perform mixed-mode arithmetic:
  - fild N — load integer as real
  - fadd X — add double-precision real
  - fstp Z — store result in Z

Displaying the Result:

- Prepare Z for output and call WriteFloat

Resetting Rounding Mode and exit:

- and ctrlWord, 001111111111b — restore default rounding (round to nearest)
- Program Exit

Summary of What This Program Demonstrates:

- Modifying the FPU control word to change rounding modes
- Performing mixed-mode arithmetic between integers and double-precision reals
- Controlling rounding behavior to ensure predictable results
- Displaying the calculated result and then restoring the FPU to default settings

UNMASKING FLOATING-POINT EXCEPTIONS

Masking and Unmasking Floating-Point Exceptions

- By default, **floating-point exceptions are masked** — the processor handles them automatically and keeps running.
- To **unmask** an exception, you clear the corresponding bit in the FPU control word.
- Example: clearing the right bit will unmask the **divide-by-zero** exception, allowing the processor to signal it instead of using a default result.

```
.data
    ctrlWord WORD ?

.code
    ; Store the current FPU control word in a 16-bit variable (ctrlWord).
    fstcw ctrlWord

    ; Clear bit 2 (Divide by zero exception mask) in the ctrlWord.
    and ctrlWord, 1111111111111011b

    ; Load the modified control word back into the FPU.
    fldcw ctrlWord
```

Unmasking the Divide-by-Zero Exception

- Store the current FPU control word in ctrlWord.
- Clear **bit 2** to unmask the divide-by-zero exception.
- Load the modified control word back into the FPU.
- After this, any divide-by-zero triggers the processor to run an exception handler.
- If no handler is installed, the processor shows an error and terminates the program.

Here is an example of code that divides by zero and generates an unmasked exception:

```
; Code that intentionally divides by zero, generating an unmasked exception
fild val1       ; Load a value into ST(0)
fdiv val2      ; Attempt to divide by zero (unmasked exception)
fst val2       ; Store the result (won't be reached due to exception)
```

Triggering an Unmasked Divide-by-Zero Exception

- Load val1 into the FPU stack with fild.
- Attempt to divide by val2 using fdiv. Since val2 is zero, this triggers the **unmasked divide-by-zero exception**.
- The following fst val2 won't execute because the exception stops normal flow.
- The system displays a **Floating-Point Division by Zero** error, giving options to retry or terminate.

The program will trigger a system dialog indicating "Floating-Point Division by Zero," giving the user the option to retry or terminate the program.

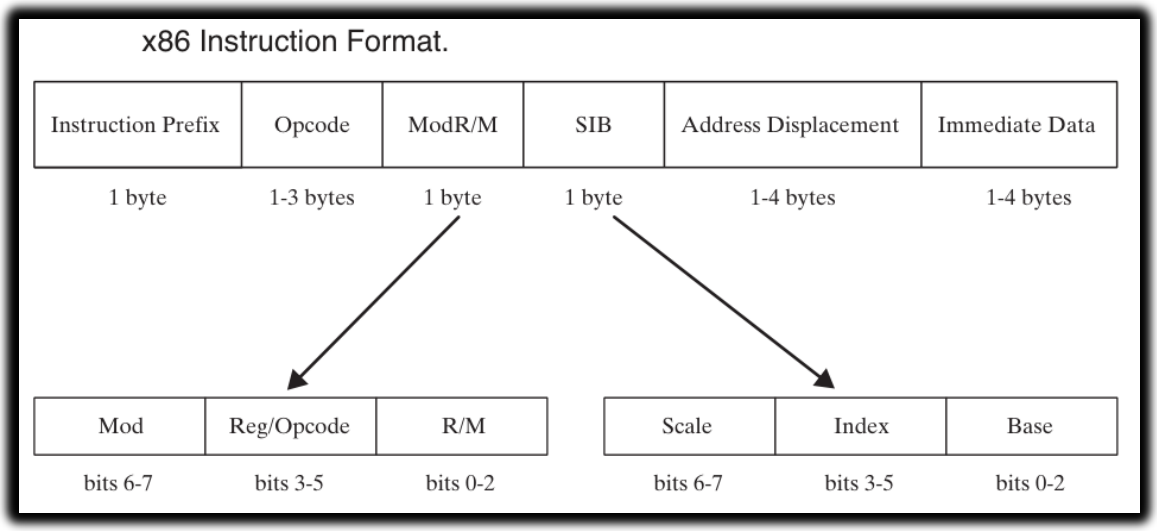
- Write an instruction that loads a duplicate of ST(0) onto the FPU stack.
- If ST(0) is positioned at absolute register R6 in the register stack, what is the position of ST(2)?
- Name at least three FPU special-purpose registers.
- When the second letter of a floating-point instruction is B, what type of operand is indicated?
- Which floating-point instructions accept immediate operands?
- How do you promote an integer in memory to a floating-point value on the FPU stack?
- Which instruction stores a floating-point value from ST(0) to memory as an integer?
- How can you change the FPU rounding mode to truncate?
- How do you restore the FPU rounding mode to round-to-nearest after changing it?
- How do you mask or unmask a floating-point exception?
- Which FPU control word bit is cleared to unmask the divide-by-zero exception?
- What happens if an unmasked divide-by-zero exception occurs and no handler is installed?
- Which instruction initializes the FPU for a clean start?
- How would you perform a mixed-mode addition between an integer and a double-precision real?
- Which instruction stores the FPU control word into a memory variable?
- Which instruction loads a modified control word back into the FPU?
- What is the effect of the fadd instruction when used after fild?
- What does the fstp instruction do to the FPU stack?
- What is the top-of-stack register in the FPU called?
- Which instructions are used to load common constants like +1.0, 0.0, or π onto the FPU stack?

X86 INSTRUCTION FORMAT

The passage you sent describes the general x86 machine instruction format and its components.

The general x86 machine instruction format contains the following fields:

- **Instruction prefix byte:** Overrides default operand sizes.
- **Opcode (operation code):** Identifies a specific variant of an instruction.
- **Mod R/M field:** Identifies the addressing mode and operands.
- **Scale index byte (SIB):** Used to calculate offsets of array indexes.
- **Address displacement field:** Holds an operand's offset, or it can be added to base and index registers in addressing modes such as base-displacement or base-index-displacement.
- **Immediate data field:** Holds constant operands.



Instruction prefix byte

- Single byte used to override default operand sizes
- Example: REX prefix changes operand size from 16-bit to 32-bit

Opcode

- Single or multiple bytes that identify a specific instruction variant
- Example: ADD instruction opcode is 01

Mod R/M field

- Single byte that specifies addressing mode and operands
- First 2 bits indicate addressing mode
- Remaining bits specify operands

Scale Index Byte (SIB)

- Single byte used to calculate array index offsets
- Works with base and index registers to determine effective address

Address displacement field

- Single or multiple bytes holding an operand's offset
- Can be added to base and index registers in base-displacement or base-index-displacement modes

Immediate data field

- Single or multiple bytes holding constant operands
- Used in instructions like MOV and ADD

Notes

- Not all instructions include all fields (e.g., ADD only has opcode and Mod R/M)
- Understanding these fields is important for writing assembly code

MOD field

- Two bits in the Mod R/M field specifying the addressing mode for the operand
- Values define which addressing mode is being used

MOD value	Addressing mode
00	Register direct
01	Register indirect with 8-bit displacement
10	Register indirect with 16-bit displacement
11	R/M field contains the operand address

Mod Field Values in x86 Instruction Encoding

- **Mod 00:**
  - Displacement = 0
  - No disp-low or disp-high bytes, **except** if R/M = 110 → 32-bit displacement is used
  - Used for memory addressing with no offset
- **Mod 01:**
  - Displacement = disp-low byte, sign-extended to 16 bits
  - No disp-high byte
  - Used when a small (8-bit) displacement is needed
- **Mod 10:**
  - Displacement = disp-high and disp-low bytes used
  - Full 32-bit displacement included
  - Used when a large displacement is needed
- **Mod 11:**
  - R/M field contains a **register number**, not memory address
  - No displacement is used
  - Operand is directly in a register

```
operand_address = [register] + displacement
```

where [register] is the value of the register specified by the R/M field, and displacement is the displacement value specified in the instruction.

- **Operand Address Calculation (if Mod ≠ 11):**
  - operand_address = [register] + displacement
  - [register] = value of register specified by R/M
  - displacement = value specified in instruction

Purpose: These modes control **how memory operands are addressed** and whether a displacement is included in the instruction encoding

Here are some examples of how the MOD field is used to specify the addressing mode for the operand in the Mod R/M field:

Instruction	MOD field	R/M field	Operand address
MOV EAX, EAX	00	00	EAX register
MOV EAX, [EBX]	00	03	The memory location at the address in the EBX register
MOV EAX, [EBX + 10]	01	03	The memory location at the address in the EBX register plus 10 bytes
MOV EAX, [EBX + ESI * 4]	10	03	The memory location at the address in the EBX register plus the value of the ESI register multiplied by 4 bytes

Then…

R/M value	Effective Address
000	[BX] + D16
001	[BX + SI] + D16
010	[BP + SI] + D16
011	[BP + DI] + D16
100	[SI] + D16
101	[DI] + D16
110	[BP] + D16
111	[BX] + D16

I. The ModR/M Byte Structure

The **ModR/M byte** is made up of several parts, and the R/M field is not a full byte by itself; it is a **3-bit sub-field** within the 8-bit ModR/M byte.

The ModR/M byte is structured as follows: **Mod** (bits 7-6) indicates the addressing mode, such as 10 for a 16-bit displacement.

The **Reg/Opcode** field (bits 5-3) usually specifies a register or provides additional opcode information.

The **R/M** field (bits 2-0) specifies the register or combination of registers used to calculate the effective address.

III. The MOD = 10 Mode and Displacement

When the **Mod field equals 10**, it means a **displacement is added to the registers**. Specifically, MOD = 10 indicates a **16-bit displacement (disp16) in 16-bit addressing mode** or a **32-bit displacement (disp32) in 32-bit addressing mode**

III. R/M Field Values (16-bit Addressing)

Our notes suggest that 000 is [BX + 10], but in standard 16-bit addressing, the R/M field maps to specific combinations:

R/M Field (Effective Address)

R/M (BINARY)	REGISTER COMBINATION
000	[BX + SI] + disp
001	[BX + DI] + disp
010	[BP + SI] + disp
011	[BP + DI] + disp
100	[SI] + disp
101	[DI] + disp
110	[BP] + disp
111	[BX] + disp

There is a common misconception that the R/M value 111 is the same as 000 because the field is only 3 bits.

This is not correct. In a 3-bit system, the values 000 through 111 represent **eight distinct possibilities**, numbered 0 through 7.

According to the table, 000 refers to [BX + SI], while 111 specifically refers to [BX]. These two values are **not interchangeable**.

For example, in the instruction MOV EAX, [BX + 10], the R/M field would actually be 111 to indicate [BX].

The MOD field would be 01 if the displacement 10 is treated as an 8-bit signed value, or 10 if it is treated as a 16-bit value. The R/M value 000 is reserved for [BX + SI].

In summary, the technical reality is as follows: the R/M field is **3 bits inside a ModR/M byte**, not a full byte.

The value 000 represents [BX + SI], while 111 represents [BX]. These values are **distinct**, and 111 is not the same as 000.

Single Byte Instructions

The simplest type of instruction is one that has **no operand or an implied operand**.

These instructions only need the **opcode field**, and its value is predetermined by the processor's instruction set.

Some common single-byte instructions are shown in the table. You might wonder why an instruction like INC DX appears here. It’s not a mistake.

The designers of the instruction set chose to assign **unique opcodes** to certain frequently used instructions.

This allows **register increments** to be performed efficiently, optimizing both **code size** and **execution speed**.

Single byte instructions:

INSTRUCTION	OPCODE (HEX)
AAA	37
AAS	3F
CBW	98
LODSB	AC
XLAT	D7
INC DX	42

- **AAA**: ASCII adjust after addition; fixes the carry flag for ASCII arithmetic.
- **AAS**: ASCII adjust after subtraction; fixes the carry flag for ASCII arithmetic.
- **CBW**: Convert byte to word by filling the upper 8 bits of the word with the sign bit of the byte.
- **LODSB**: Load a byte from the current position pointed to by the string index (SI).
- **XLAT**: Translate a byte using the ASCII translation table in memory.
- **INC DX**: Increment the DX register by 1.

Additional Notes:

- INC DX is included in the table of single-byte instructions because it is **frequently used**.
- The instruction set designers assigned **unique opcodes** to commonly used instructions like INC DX to **optimize code size and execution speed**.

Here is an example of how the INC DX instruction can be used:

```
MOV DX, 10
INC DX
```

This code will increment the DX register by one, so the value of DX will be 11 after the code is executed.

Single-byte instructions are the simplest type of instruction, but they can be used to perform a variety of operations.

Mov Immediate To Register

When you want to move a number directly into a CPU register (like AX, BX, etc.) in **x86 assembly**, the processor expects the instruction to be encoded in a very specific **binary format**. This is what the "b8 + rw dw" stuff is about.

What the parts mean:

1. **B8 (the opcode)**
   - This is the base code that tells the CPU: "Hey, I want to **move an immediate value into a register**."
   - Think of it like a **label that identifies the type of instruction**.
2. **rw (register code)**
   - Every register has a **number code** (from 0 to 7 for 8-bit or 16-bit registers).
   - This tells the CPU **which register** to put the number into.
   - For example:
     - AX = 0
     - CX = 1
     - DX = 2
     - BX = 3
     - And so on…
3. **dw (data word)**
   - This is the **immediate value** you want to put in the register.
   - If it's a 16-bit number, it's stored **low byte first**, then high byte (little-endian order).
   - Example: To put 0x1234 into AX, the CPU sees it as 34 12.

Putting it together

- Suppose you want to move 0x1234 into AX.
- Opcode for MOV immediate to AX = B8 + 0 (because AX code = 0) → B8.
- Immediate value = 0x1234 → encoded as 34 12.
- Full instruction bytes = **B8 34 12**.

So basically:

“b8 + rw dw” = opcode B8 for MOV + register code + immediate value

Think of it like ordering a pizza:

- **B8** = “I want pizza” (the instruction type)
- **rw** = “Which topping?” (which register)
- **dw** = “Here’s what’s on it” (the actual number/value)

```
B8 + rw dw
```

To encode a MOV immediate instruction, you must first determine the register code for the destination register. The register codes are listed in the following table:

2. REGISTER ENCODING

Register	Code
AX/AL	0
CX/CL	1
DX/DL	2
BX/BL	3
SP/AH	4
BP/CH	5
SI/DH	6
DI/BH	7

Standard 3-bit register codes for x86 instructions.

Once you have determined the register code, you can encode the MOV immediate instruction as follows:

- Add the register code to B8 to get the opcode.
- Append the immediate operand to the opcode, low byte first.

When you want to move a number directly into a register, like AX or BX, the instruction is encoded in **machine code** using a combination of an **opcode**, the **register number**, and the **immediate value**. Here's how it works:

Example 1: MOV AX, 1

1. The **register code** for AX is 0.
2. The base opcode for moving an immediate value to a 16-bit register is B8.
3. To get the final opcode, you **add the register code** to the base opcode: B8 + 0 = B8.
4. The **immediate value** 1 is stored in **little-endian order**: 01 00.
5. The complete machine code is: **B8 01 00**.

Steps Summarized:

- Start with base opcode: B8
- Add register code: B8 + 0 = B8
- Append immediate value in little-endian: 01 00
- Machine code: **B8 01 00**

Example 2: MOV BX, 1234h

1. Register code for BX = 3.
2. Base opcode = B8. Add register code: B8 + 3 = BB.
3. Immediate value 1234h in little-endian order: 34 12.
4. Complete machine code: BB 34 12.

Steps Summarized:

- Base opcode: B8
- Add register code: B8 + 3 = BB
- Immediate value in little-endian: 34 12
- Machine code: **BB 34 12**

Quick Practice: PUSH CX

1. Base opcode for PUSH reg16 = 50.
2. Register code for CX = 1. Add it: 50 + 1 = 51.
3. Machine code: **51**.

Key Takeaways

- **Opcode + Register Code** tells the CPU which instruction and which register.
- **Immediate values** are stored **little-endian**: low byte first, high byte second.
- You can **practice encoding instructions by hand**, and then check your work with MASM or another assembler.

Register Mode Instructions

When you want to move a value **from one register to another** (like MOV AX, BX), the CPU doesn't just read the names—you have to encode it into **machine code**. Here's how it works:

```
89/r reg
```

Where:

- **89** is the opcode for moving a value from one register to another.
- **/r** indicates that a Mod R/M byte follows the opcode.
- **reg** is the register code for the destination register.

Opcode

- The base opcode for **register-to-register MOV** is **89**.
- This tells the CPU: "Move a value from one register to another."

Mod R/M Byte

After the opcode comes a **Mod R/M byte**.

This byte tells the CPU **which register is the source** and **which register is the destination**.

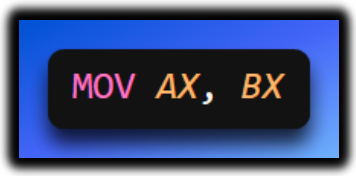
The Mod R/M byte is **split into three fields**:

- **mod (bits 7–6)** – indicates the addressing mode (register or memory).
- **reg (bits 5–3)** – the **source or destination register**, depending on the instruction.
- **r/m (bits 2–0)** – the other register or memory location.

So basically, the Mod R/M byte is a **tiny instruction inside the instruction**, telling the CPU exactly which registers to use.

Example:

Let's say you want to encode:



1. **Opcode** = 89 (move register-to-register).
2. **Mod R/M byte**:
   - mod = 11 (binary) → indicates both operands are registers.
   - reg = code for BX = 011 (binary).
   - r/m = code for AX = 000 (binary).

So, the Mod R/M byte = 11011000 (binary) = D8 (hex).

Final machine code: **89 D8**

This means: "Move the value from BX into AX."

If it helps, I like to think of it like a **little map inside the instruction**:

- 89 → "I'm moving stuff."
- D8 → "From BX to AX, follow the coordinates."

The opcode for a register-to-register MOV instruction is 89.

The Mod R/M byte specifies which register is the source and which is the destination.

If mod = 11, both operands are registers.

The reg field identifies the source register, and the r/m field identifies the destination register.

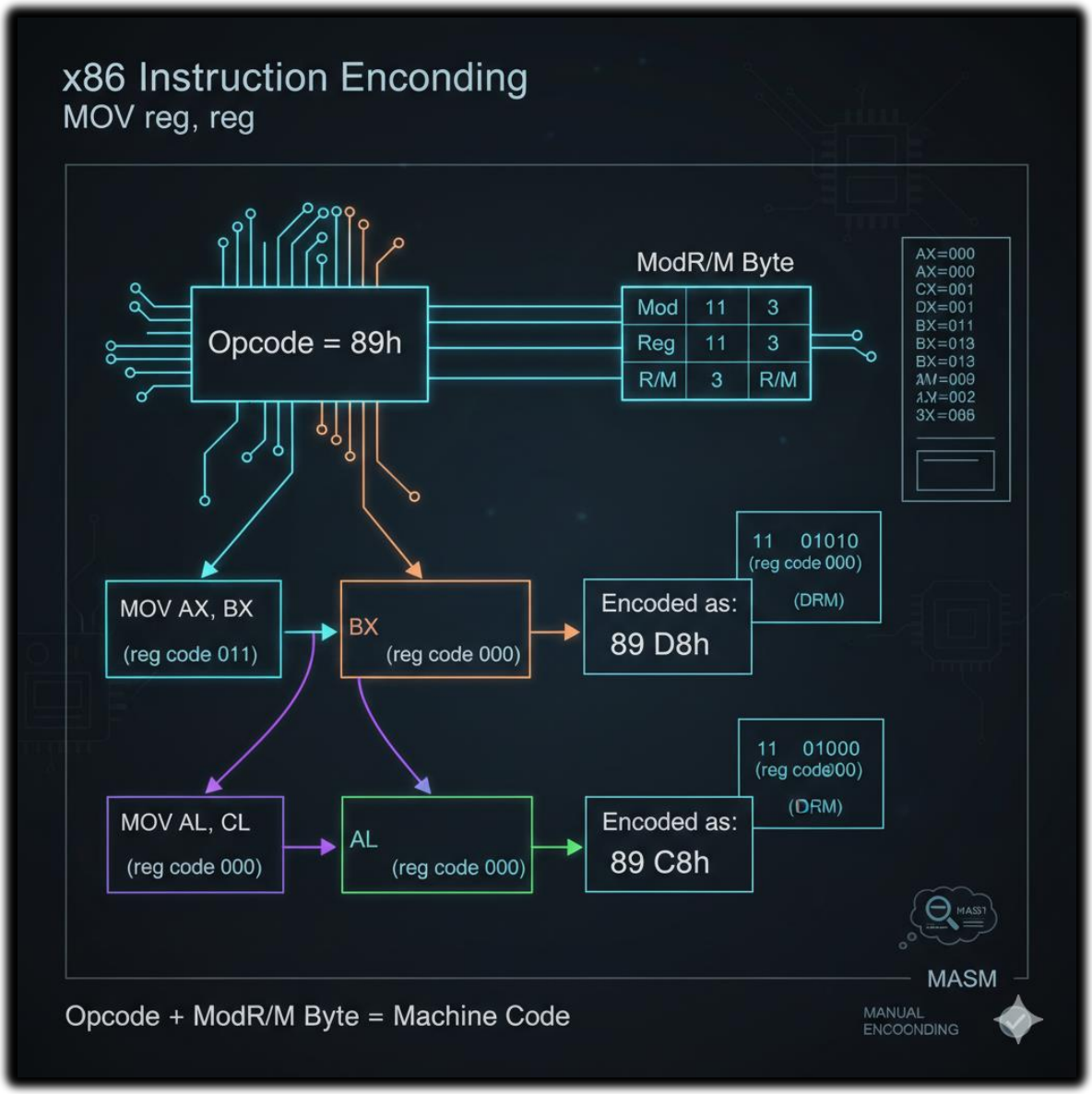
For example, MOV AX, BX is encoded as 89 D8, and MOV AL, CL is 89 C8.

Register codes are fixed, such as AX = 0, BX = 3, CL = 1, and AL = 0.

Practicing encoding by hand is helpful, but MASM will always generate the correct machine code if you check the listing.

The detailed breakdown of the Mod R/M byte (like how D8 is calculated) is only necessary if you want to manually encode instructions.

Other instructions, such as ADD, have different opcodes and Mod R/M interpretations, but the basic principle is the same: opcode plus Mod R/M byte equals the machine code.



PROCESSOR OPERAND-SIZE PREFIX

The **operand-size prefix** (66h) is a little tag you can add to an instruction to change its default size. Back in the days of the **8088/8086**, almost all 256 possible opcodes were already used for instructions with 8- and 16-bit operands. When Intel moved to **32-bit processors**, they needed a clever way to handle 32-bit operands **without breaking older programs**.

Here's how it works:

- On **16-bit processors**, if a program wants to use a **32-bit operand**, you add the prefix byte.
- On **32-bit processors**, 32-bit operands are the default, so if you want a **16-bit operand**, you add the prefix byte.
- **8-bit operands** are simple—they **never need a prefix**.

So basically, the prefix is a way for the CPU to know, "Hey, I want to use a different operand size than usual for this instruction."

Here's an example of using the operand-size prefix in **16-bit mode**:

```
715 .model small
716 .286
717 .stack 100h
718 .code
719 main PROC
720     mov ax, dx ; 8B C2
721     mov al, dl ; 8A C2
722 main ENDP
723 END main
```

The **mov ax,dx** instruction uses 16-bit operands, so it does not need a prefix byte.

The **mov al,dl** instruction uses 8-bit operands, so it also does not need a prefix byte.

Here is an example of how to use the operand-size prefix in 32-bit mode:

```
726 .model small
727 .386
728 .stack 100h
729 .code
730 main PROC
731     mov ax, dx ; 66 8B C2
732     mov al, dl ; 8A C2
733 main ENDP
734 END main
```

The **mov ax,dx** instruction now needs a prefix byte to indicate that it is using 16-bit operands.

The **mov al,dl** instruction still does not need a prefix byte because it is using 8-bit operands.

The operand-size prefix is a powerful tool that allows programmers to control the operand size of instructions. This can be useful for improving performance or for ensuring compatibility with older processors.

MEMORY MODE INSTRUCTIONS

In **x86 assembly**, Memory Mode instructions are your go-to tools for working with data stored in memory. They let you **load data from memory into registers** or **store data from registers into memory**, using all sorts of addressing tricks.

The key to these instructions is the **Mod field** in the **Mod R/M byte**, which tells the CPU exactly how to find the memory operand. Here's the breakdown:

I. How Mod Memory Mode Works

- **Mod R/M Byte:** This is part of the instruction itself. It encodes both the addressing mode and the operands.
- **Mod Field (bits 6-7):** This small part of the byte tells the CPU which way to interpret the memory address.

II. Here's what the Mod field values mean:

- 00 → **Register-indirect addressing**: Access memory through a register.
- 01 → **Displacement-only addressing**: Add a small immediate value (displacement) to the base register to find the memory location.
- 10 → **SIB (Scale-Index-Base) addressing**: Great for more complex memory calculations involving scaled indexes.
- 11 → **Register mode**: No memory involved—both operands are registers.

III. Registers and Memory

Memory Mode instructions are flexible—they can mix registers and memory in different ways, which makes them super handy for reading and writing data in all sorts of scenarios.

Next, we'll break down the **example code** from earlier so you can see these rules in action.

```
770 main PROC
771     mov ax, [si]          ; Load AX with the value at memory address pointed to by SI
772     ; Mod 00, R/M 101 (SI is used as the offset address)
773
774     mov [si], al          ; Store the value in AL at the memory address pointed to by SI
775     ; Mod 00, R/M 101 (SI is used as the offset address)
776
777     add bx, 10h           ; Add 10h to BX
778     mov bx, [bx]          ; Load BX with the value at memory address [BX + 10h]
779     ; Mod 00, R/M 010 (Offset BX is used)
780
781     mov [bx + si], cx     ; Store the value in CX at memory address [BX + SI]
782     ; Mod 00, R/M 110 (Offset BX+SI is used)
```

- mov ax, [si] → Loads memory at SI into AX; uses **Mod 00**, R/M 101
- mov [si], al → Stores AL into memory at SI; uses **Mod 00**, R/M 101
- add bx, 10h → Adds 10h to BX
- mov bx, [bx] → Loads memory at BX into BX; uses **Mod 00**, R/M 010
- mov [bx + si], cx → Stores CX into memory at [BX + SI]; uses **Mod 00**, R/M 110
- Demonstrates **Mod Memory Mode instructions** for accessing memory operands with different addressing modes

IV. Mod R/M Byte and Addressing Modes

The **Mod R/M byte** lets programmers specify different memory-addressing modes in x86.

The **Mod field** identifies the general **group of addressing modes** (e.g., no displacement, 8-bit displacement, 32-bit displacement, or register mode).

The **R/M field** identifies the **specific addressing mode** within that group (e.g., which register or combination of registers is used).

The **table for Mod 00** will show the mapping of R/M values to the exact memory addressing options when the Mod field is 00.

R/M	Effective address
000	Register
001	[Base register]
010	[Base register + Displacement]
011	[Base register + Index register]
100	[Base register + Index register + Displacement]
101	[SI]
110	[DI]
111	[Base register + Displacement + 8]

The following examples show how to use the Mod R/M byte to encode MOV instructions that use different memory-addressing modes:

```
.model small
.data
    data1   dw 05h        ; Define a data item
    data2   dw 0A0Bh      ; Define another data item

.code
main PROC
    mov ax, [si]         ; Load AX with the value at memory address pointed to by SI
    ; Mod 00, R/M 101 (SI is used as the offset address)

    mov [si], al         ; Store the value in AL at the memory address pointed to by SI
    ; Mod 00, R/M 101 (SI is used as the offset address)

    add bx, 10h          ; Add 10h to BX
    mov bx, [bx]         ; Load BX with the value at memory address [BX + 10h]
    ; Mod 00, R/M 010 (Offset BX is used)

    mov [bx + si], cx    ; Store the value in CX at memory address [BX + SI]
    ; Mod 00, R/M 110 (Offset BX+SI is used)

    ; Terminate the program
    mov ah, 4Ch
    int 21h

main ENDP
END main
```

mov ax, [si]

- Loads memory at [SI] into AX
- **Mod 00** → register-indirect addressing
- **R/M 101** → SI as offset

mov [si], al

- Stores AL into memory at [SI]
- **Mod 00**, **R/M 101** → SI as offset

add bx, 10h

- Adds immediate value 10h to BX

mov bx, [bx]

- Loads memory at [BX + 10h] into BX
- **Mod 00** → register-indirect addressing
- **R/M 010** → BX as base with displacement

mov [bx + si], cx

- Stores CX into memory at [BX + SI]
- **Mod 00** → register-indirect addressing
- **R/M 110** → BX + SI as offsets

Key point:

- These examples show how **Mod Memory Mode instructions** move data between registers and memory
- **Mod R/M byte** specifies the addressing mode and operands for each instruction

Use these tables as references when hand-assembling MOV instructions
(For more details, refer to the Intel manuals.)

Partial List of Mod R/M Bytes (16-Bit Segments).

Byte:		AL	CL	DL	BL	AH	CH	DH	BH	
Word:		AX	CX	DX	BX	SP	BP	SI	DI	
Register ID:		000	001	010	011	100	101	110	111	
Mod	**R/M**	**Mod R/M Value**								**Effective Address**
00	000	00	08	10	18	20	28	30	38	[BX + SI]
	001	01	09	11	19	21	29	31	39	[BX + DI]
	010	02	0A	12	1A	22	2A	32	3A	[BP + SI]
	011	03	0B	13	1B	23	2B	33	3B	[BP + DI]
	100	04	0C	14	1C	24	2C	34	3C	[SI]
	101	05	0D	15	1D	25	2D	35	3D	[DI]
	110	06	0E	16	1E	26	2E	36	3E	16-bit displacement
	111	07	0F	17	1F	27	2F	37	3F	[BX]

Table 2:

MOV Instruction Opcodes.

Opcode	Instruction	Description
88/r	MOV eb,rb	Move byte register into EA byte
89/r	MOV ew,rw	Move word register into EA word
8A/r	MOV rb,eb	Move EA byte into byte register
8B/r	MOV rw,ew	Move EA word into word register
8C/0	MOV ew,ES	Move ES into EA word
8C/1	MOV ew,CS	Move CS into EA word
8C/2	MOV ew,SS	Move SS into EA word
8C/3	MOV ew,DS	Move DS into EA word
8E/0	MOV ES,mw	Move memory word into ES
8E/0	MOV ES,rw	Move word register into ES
8E/2	MOV SS,mw	Move memory word into SS
8E/2	MOV SS,rw	Move register word into SS
8E/3	MOV DS,mw	Move memory word into DS
8E/3	MOV DS,rw	Move word register into DS
A0 dw	MOV AL,xb	Move byte variable (offset dw) into AL
A1 dw	MOV AX,xw	Move word variable (offset dw) into AX
A2 dw	MOV xb,AL	Move AL into byte variable (offset dw)

A3 dw	MOV xw,AX	Move AX into word register (offset dw)
B0 +rb db	MOV rb,db	Move immediate byte into byte register
B8 +rw dw	MOV rw,dw	Move immediate word into word register
C6 /0 db	MOV eb,db	Move immediate byte into EA byte
C7 /0 dw	MOV ew,dw	Move immediate word into EA word

Table 3:

Key to Instruction Opcodes.	
/n:	A Mod R/M byte follows the opcode, possibly followed by immediate and displacement fields. The digit n (0–7) is the value of the reg field of the Mod R/M byte.
/r:	A Mod R/M byte follows the opcode, possibly followed by immediate and displacement fields.
db:	An immediate byte operand follows the opcode and Mod R/M bytes.
dw:	An immediate word operand follows the opcode and Mod R/M bytes.
+rb:	A register code (0–7) for an 8-bit register, which is added to the preceding hexadecimal byte to form an 8-bit opcode.
+rw:	A register code (0–7) for a 16-bit register, which is added to the preceding hexadecimal byte to form an 8-bit opcode.

Table 4:

Key to Instruction Operands.	
db	A signed value between −128 and +127. If combined with a word operand, this value is sign-extended.
dw	An immediate word value that is an operand of the instruction.
eb	A byte-sized operand, either register or memory.
ew	A word-sized operand, either register or memory.
rb	An 8-bit register identified by the value (0–7).
rw	A 16-bit register identified by the value (0–7).
xb	A simple byte memory variable without a base or index register.
xw	A simple word memory variable without a base or index register.

Table 12-28 contains a few additional examples of MOV instructions that you can assemble by hand and compare to the machine code shown in the table. We assume that myWord begins at offset 0102h.

Sample MOV Instructions, with Machine Code.

Instruction	Machine Code	Addressing Mode
mov ax,myWord	A1 02 01	direct (optimized for AX)
mov myWord,bx	89 1E 02 01	direct
mov [di],bx	89 1D	indexed
mov [bx+2],ax	89 47 02	base-disp
mov [bx+si],ax	89 00	base-indexed
mov word ptr [bx+di+2],1234h	C7 41 02 34 12	base-indexed-disp

QUESTIONS

These questions are already answered, so just read:

Provide the opcodes for the following MOV instructions.

Instruction	Opcode
MOV AX, @DATA	A1
MOV DS, AX	8E D8
MOV AX, BX	89 D0
MOV BL, AL	88 C3
MOV AL, [SI]	8A 36
MOV MYBYTE, AL	88 44 01
MOV MYWORD, AX	89 84 01

Mod R/M bytes for MOV instructions:

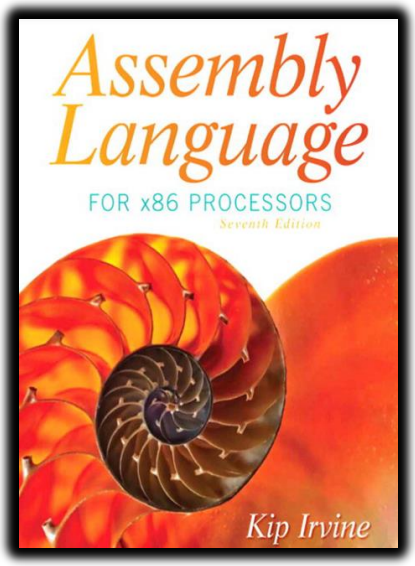
Instruction	Mod R/M byte
MOV AX, @DATA	00 00 00
MOV DS, AX	8E D8
MOV DL, BL	00 10
MOV BL, [DI]	00 3E
MOV AX, [SI + 2]	04 36 02
MOV AX, ARRAY[SI]	00 46
MOV ARRAY[DI], AX	00 3C

Concept-wise:

- Mod R/M byte = defines addressing mode and operands. ✅
- Different types of operands (register, memory, segment, global variable) affect how the byte is encoded. ✅
- Understanding this is essential for calculating instruction encodings. ✅

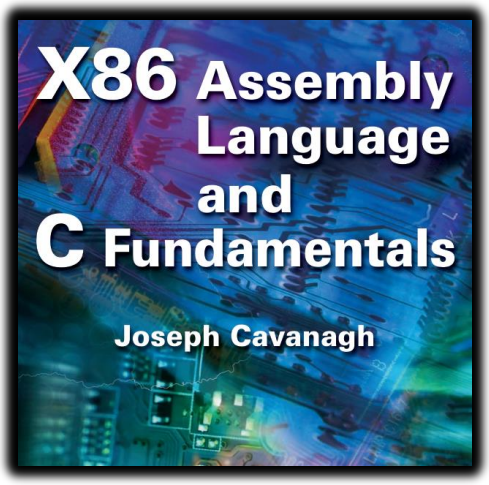
WHY I SKIPPED BIOS LEVEL PROGRAMMING AND THE REST OF THE BOOK

This is the book!



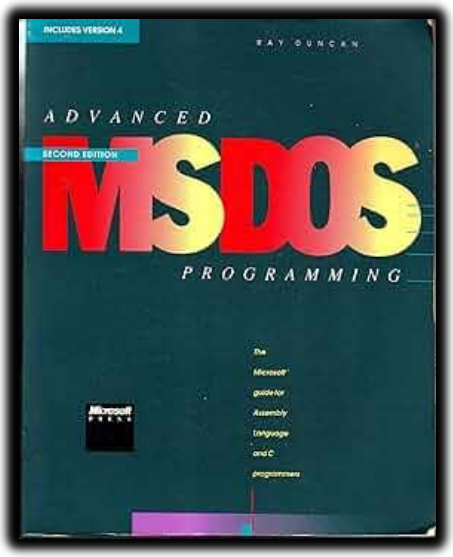
Reasons why did I decide to skip BIOS level programming?

- BIOS **+ Assembly** = 💀💀 Death cocktail. No thanks. I'll stick to ASM + modern code.
- My motto: **"No BIOS, no problem!"** 🚀
- The remaining topics are super advanced, and my Assembly skills aren't ready yet.
- **Assembly + C/C++**: I'll tackle this book later… when I'm feeling brave 📜😈
- If you want to mix C and ASM (Inline ASM), read this…

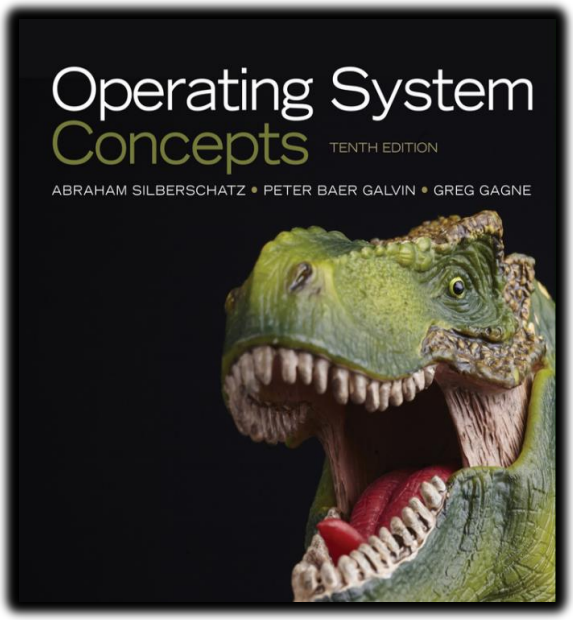


16-bit MS-DOS Programming

- Meh… don't need it now, probably never.
- Maybe a fun side project someday, but not for me right now 🤔



Disk fundamentals, I will cover that topic when I learn about operating systems storage management, using this book...



CRUCIAL!! YOU WILL ALWAYS FIND THIS EVERYWHERE... (CLOSING CHAPTER)

- **BYTE (8 bits)**: Short form - **B**
- **SBYTE (8 bits, signed)**: Short form - **SB**
- **WORD (16 bits)**: Short form - **W**
- **SWORD (16 bits, signed)**: Short form - **SW**
- **DWORD (32 bits)**: Short form - **D**
- **SDWORD (32 bits, signed)**: Short form - **SD**
- **FWORD (48 bits)**: Short form - **FW**
- **QWORD (64 bits)**: Short form - **Q**
- **TBYTE (80 bits)**: Short form - **T**
- **REAL4 (32-bit floating-point)**: Short form - **F**
- **REAL8 (64-bit floating-point)**: Short form - **FF**
- **REAL10 (80-bit floating-point)**: Short form - **FT**

Farewell, fellow code wizards—keep building worlds with your keyboards, and may I never find your bugs... because I *will* exploit them! 😈💻