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)

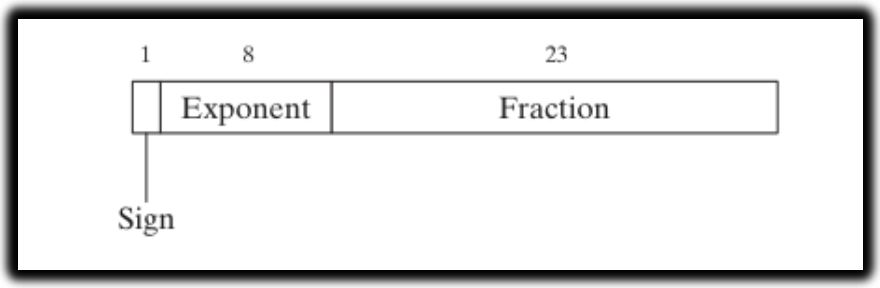
- Shows the **power of 2**
- It scales the number (makes it bigger or smaller)

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

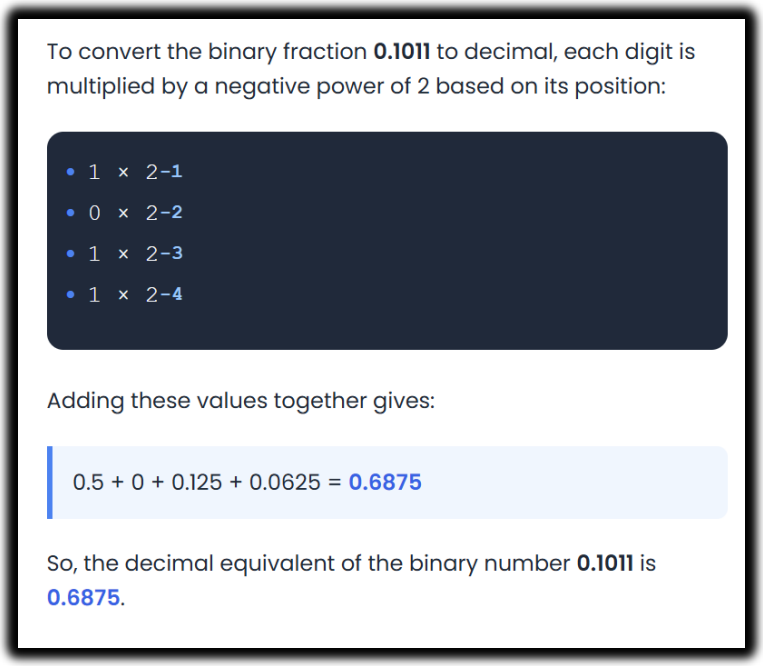
Let's discuss another table:

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

The passage below describes 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 that the significand must be adjusted so that the leading bit is 1.

For example, the significand 1.1011 is normalized by shifting the bits one position to the left, giving the normalized significand 11.011.

The next step is to encode the sign bit, exponent, and significand fields.

The **sign bit** is encoded as a single bit, with 0 representing a positive number and 1 representing a negative number.

The **exponent** is encoded as an 8-bit unsigned integer with a bias of 127. This means that the number's actual exponent must be added to 127 to get the biased exponent.

For example, the actual **exponent 5** is **encoded as** the **biased exponent 132.**

The **significand** is encoded as a 23-bit unsigned integer. The leading bit of the normalized significand is not explicitly encoded, since it is always 1.

This means that the significand 11.011 is encoded as the 23-bit integer 01000100000000000000000.

Once the sign bit, exponent, and significand fields have been encoded, they can be combined to form the complete binary IEEE short real.

The following table shows an example of how to create the IEEE representation of the floating-point number $1.101 * 2^{20}$:

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

The IEEE representation of a floating-point number can be used to store and manipulate the number in a computer.

IEEE floating-point numbers are widely used in a variety of applications, including scientific computing, graphics, and gaming.

The table you sent shows the binary value, exponent, sign bit, and significand of several different IEEE floating-point numbers. Here is a concise explanation of each row:

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

The first row shows the IEEE representation of the floating-point number -1.11. The sign bit is 1, which indicates that the number is negative. The exponent is 127, which is the largest possible exponent for a normalized floating-point number. The significand is 1101.101, which is the binary representation of the fraction 3/4.

The second row shows the IEEE representation of the floating-point number 1.10111. The sign bit is 0, which indicates that the number is positive. The exponent is 0, which means that the significand is not multiplied by 2. The significand is 1.10111, which is the binary representation of the fraction 23/16.

The other rows in the table show the IEEE representations of other floating-point numbers. The last row in the table shows the smallest possible normalized floating-point number.

The IEEE specification includes several real-number and non-number encodings, including:

Positive and negative zero:

These are two different representations of the number zero.

Denormalized finite numbers:

These are numbers that are very close to zero. They are represented using a special format that allows them to be represented with a wider range of exponents.

Normalized finite numbers:

These are all the other nonzero finite numbers that can be represented in floating-point format. They are represented using a format that allows them to be represented with a high degree of accuracy.

Positive and negative infinity:

These are representations of the concepts of positive and negative infinity.

Non-numeric values (NaN, known as Not a Number):

These are used to represent results of invalid floating-point operations.

Indefinite numbers:

These are used by the floating-point unit (FPU) as responses to some invalid floating-point operations.

Normalized and denormalized numbers

Normalized finite numbers are all the nonzero finite values that can be encoded in a normalized real number between zero and infinity. Normalized numbers have a leading bit of 1 in the significand. This allows for the greatest possible precision in representing the number.

Denormalized finite numbers are numbers that are very close to zero. They are represented using a special format that allows them to be represented with a wider range of exponents. Denormalized numbers have a leading bit of 0 in the significand. This means that some precision is lost in representing the number, but it allows for a wider range of numbers to be represented.

In the example you provided, the number:

$1.01011110000000000001111 \times 2^{-129}$

Is too small to be represented as a normalized number.

The FPU therefore denormalizes the number by shifting the binary point left 1 bit at a time until the exponent reaches a valid range.

$1.01011110000000000001111 \times 2^{-129} \rightarrow 0.10101111000000000000111 \times 2^{-128}$

$0.10101111000000000000111 \times 2^{-128} \rightarrow 0.01010111100000000000011 \times 2^{-127}$

$0.01010111100000000000011 \times 2^{-127} \rightarrow 0.00101011110000000000001 \times 2^{-126}$

After three denormalization steps, the exponent is now within the valid range. The denormalized number is now represented as $0.00101011110000000000001 \times 2^{-126}$.

This results in some loss of precision in the significand, but it allows the number to be represented.

Denormalized numbers are important because they allow the FPU to represent a wider range of numbers than it would otherwise be able to. This is important for many applications, such as scientific computing and graphics.

So, to finish off:

Converting IEEE single-precision values to decimal

The following are the steps to convert an IEEE single-precision value to decimal:

- Check the most significant bit (MSB) to determine the sign of the number. If the MSB is 1, the number is negative. If the MSB is 0, the number is positive.
- Extract the next 8 bits to get the exponent. Subtract 127 from the exponent to get the unbiased exponent. Convert the unbiased exponent to decimal.
- Extract the next 23 bits to get the significand. Add a leading 1 to the significand bits. Ignore any trailing zeros.
- Create a floating-point binary number using the significand, the sign determined in step 1, and the exponent calculated in step 2.
- Denormalize the binary number produced in step 3. To do this, shift the binary point the number of places equal to the value of the exponent. If the exponent is positive, shift the binary point to the right. If the exponent is negative, shift the binary point to the left.
- Convert the denormalized binary number to decimal using weighted positional notation.

Example:

Convert the following IEEE single-precision value to decimal:

0 10000010 01011000000000000000000

- The MSB is 0, so the number is positive.
- The unbiased exponent is 00000011, which is equal to decimal 3.
- The significand is 01011000000000000000000, which is equal to 1.01011 binary.
- The floating-point binary number is 1.01011 * 2^3.
- The denormalized binary number is 1010.11.

- The decimal value is 10 3/4, or 10.75.

POSITIVE AND NEGATIVE INFINITY

Positive and negative infinity are special floating-point values that represent the maximum positive and negative real numbers, respectively. They are used to represent results of operations that would otherwise overflow, such as dividing by zero or adding a very large number to a very small number.

NaNs are bit patterns that do not represent any valid real number. They are used to represent results of invalid floating-point operations, such as dividing zero by zero or taking the square root of a negative number.

The following table shows the specific encodings for positive infinity, negative infinity, quiet NaNs, and signaling NaNs in the IEEE floating-point standard:

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...

Note that the significand field for NaNs can be any value.

Floating-point units (FPUs) handle positive infinity, negative infinity, and NaNs in special ways.

For example, if the FPU attempts to perform an operation that would result in positive infinity or negative infinity, it will instead return the corresponding infinity value.

If the FPU attempts to perform an operation that would result in a NaN, it will instead return a NaN value and may also generate an exception.

The specific behavior of the FPU when handling positive infinity, negative infinity, and NaNs is defined by the IEEE floating-point standard.

This ensures that all floating-point units behave in a consistent manner, regardless of the manufacturer or architecture.

TABLE 12-6 Specific 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[a]

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

CONVERTING DECIMAL FRACTIONS TO BINARY REALS

Method 1

- Write the decimal fraction as a sum of fractions in the form (1/2 + 1/4 + 1/8 + ...).
- Translate each fraction in the sum to binary.
- Add the binary fractions together to get the final result.

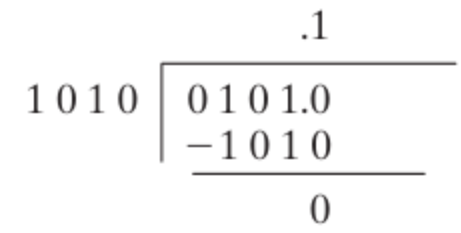
Method 2

- Convert the numerator and denominator of the decimal fraction to binary.
- Perform long division on the binary numbers.
- The quotient of the division is the binary representation of the decimal fraction.

Example

Let's use the second method to convert the decimal fraction 0.5 to binary.

- Convert the numerator and denominator of the decimal fraction to binary.
- 5 in decimal is 101 in binary.
- 10 in decimal is 1010 in binary.
- Perform long division on the binary numbers:



When 1010 binary is subtracted from the dividend the remainder is zero, and the division stops.

Therefore, the decimal fraction 5/10 equals 0.1 binary. We will call this approach the binary long division method.

Representing 0.2 in Binary

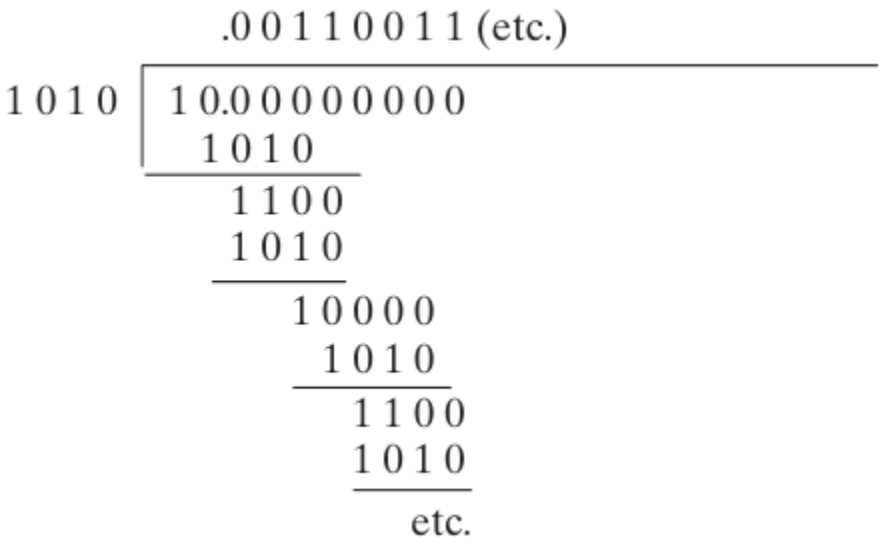
Let's convert decimal 0.2 (2/10) to binary using the binary long division method. First, we divide binary 10 by binary 1010 (decimal 10).

The first quotient large enough to use is 10000. After dividing 1010 into 10000, the remainder is 110.

Appending another zero, the new dividend is 1100. After dividing 1010 into 1100, the remain- der is 10. After appending three zeros, the new dividend is 10000.

This is the same dividend we started with. From this point on, the sequence of the bits in the quotient repeats (0011. . .), so we know that an exact quotient will not be found and 0.2 cannot be represented by a finite number of bits.

The single-precision encoded significand is **00110011001100110011001.**



==========

Examples of decimals factored as binary reals:

Decimal Fraction	Factored As...	Binary Real
1/2	1/2	.1
1/4	1/4	.01
3/4	1/2 + 1/4	.11
1/8	1/8	.001
7/8	1/2 + 1/4 + 1/8	.111
3/8	1/4 + 1/8	.011
1/16	1/16	.0001
3/16	1/8 + 1/16	.0011
5/16	1/4 + 1/16	.0101

The table above shows how to convert decimal fractions to binary reals using the first method I described above.

The first column of the table shows the decimal fraction. The second column shows the factored form of the decimal fraction.

The third column shows the binary representation of the decimal fraction.

For example, the first row of the table shows the decimal fraction 1/2. The factored form of 1/2 is 1/2. The binary representation of 1/2 is .1.

The second row of the table shows the decimal fraction 1/4. The factored form of 1/4 is 1/4. The binary representation of 1/4 is .01.

The third row of the table shows the decimal fraction 3/4. The factored form of 3/4 is 1/2 + 1/4. The binary representation of 3/4 is .11.

The rest of the rows in the table follow the same pattern.

To use the table to convert a decimal fraction to binary, simply find the decimal fraction in the first column of the table. The binary representation of the decimal fraction will be in the third column.

For example, to convert the decimal fraction 5/8 to binary, find the decimal fraction 5/8 in the first column of the table. The binary representation of 5/8 is .101.

It is important to note that not all decimal fractions can be represented exactly in binary. This is because there is an infinite number of real numbers, but only a finite number of

binary digits. As a result, some decimal fractions must be approximated when they are converted to binary.

==========

The **factored column** in the table shows how the decimal fraction can be expressed as a sum of fractions in the form (1/2 + 1/4 + 1/8 + ...). This is useful because each of the fractions in the sum can be easily converted to binary.

To factor a decimal fraction, you can use the following steps:

- Find the largest power of two that is less than or equal to the decimal fraction.
- Subtract that power of two from the decimal fraction.
- Repeat steps 1 and 2 until the decimal fraction is zero.
- The factors of the decimal fraction are the powers of two that were subtracted in each step.

For example, to factor the decimal fraction 3/4, we would do the following:

- The largest power of two that is less than or equal to 3/4 is 1/2.
- Subtracting 1/2 from 3/4 gives us 1/4.
- The largest power of two that is less than or equal to 1/4 is 1/4.
- Subtracting 1/4 from 1/4 gives us zero.
- Therefore, the factors of 3/4 are 1/2 and 1/4.

Once you have factored the decimal fraction, you can convert each of the factors to binary. To do this, **simply divide the numerator of the factor by the denominator. The quotient is the binary representation of the factor.**

For example, to convert the factor 1/2 to binary, we would divide the numerator 1 by the denominator 2. The quotient is 0.5, which is the binary representation of 1/2.

To convert the factor 1/4 to binary, we would divide the numerator 1 by the denominator 4. The quotient is 0.25, which is the binary representation of 1/4.

Once you have converted each of the factors to binary, you can add them together to get the binary representation of the decimal fraction.

For example, the binary representation of 3/4 is 0.11, which is the sum of the binary representations of 1/2 and 1/4.

==========

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

The **FPU** is a specialized hardware unit that performs floating-point calculations. It has its own set of registers called a **register stack.**

The FPU loads values from memory into the register stack, performs calculations, and stores stack values into memory.

The FPU evaluates mathematical expressions in **postfix format.**

A **postfix expression** is an expression where the operands appear before the operators. For example, the postfix expression for the **infix expression (5 * 6) - 4** is **5 6 * 4 -.**

Left to Right	Stack		Action
5	5	ST (0)	push 5
5 6	5 6	ST (1) ST (0)	push 6
5 6 *	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.

The FPU uses an expression stack to hold intermediate values during the evaluation of postfix expressions. The expression stack is a **last-in-first-out (LIFO) stack.**

This means that the last value pushed onto the stack is the first value popped off the stack.

To evaluate a postfix expression, the FPU pushes the operands onto the expression stack.

Then, it pops the top two operands off the stack and performs the operation specified by the next operator in the expression. The result of the operation is pushed onto the stack.

This process continues until all of the operators in the expression have been processed.

The following table shows some examples of equivalent infix and postfix expressions:

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 - *

Note that the parentheses in the infix expressions are not necessary in the postfix expressions, because the order of operations is implied by the order of the operands.

For example, in the expression (A + B) * (C + D), the multiplication operation is performed before the addition operation, even though there are no parentheses. This is because multiplication has a higher precedence than addition.

Here is an example of how the FPU would evaluate the postfix expression A B + using its expression stack:

- Expression stack: empty
- Next operand: A
- Push A onto the expression stack
- Expression stack: A
- Next operand: B
- Push B onto the expression stack
- Expression stack: A, B
- Next operator: +
- Pop the top two operands off the expression stack and perform the addition operation
- Push the result of the addition operation onto the expression stack

- Expression stack: A + B

The FPU would then stop evaluating the expression, because there are no more operands or operators. The result of the expression is A + B.

The FPU evaluates all postfix expressions in the same way.

It pushes the operands onto the expression stack, pops the top two operands off the stack and performs the operation specified by the next operator, and pushes the result of the operation onto the stack.

This process continues until all of the operators in the expression have been processed. The final value on the expression stack is the result of the expression.

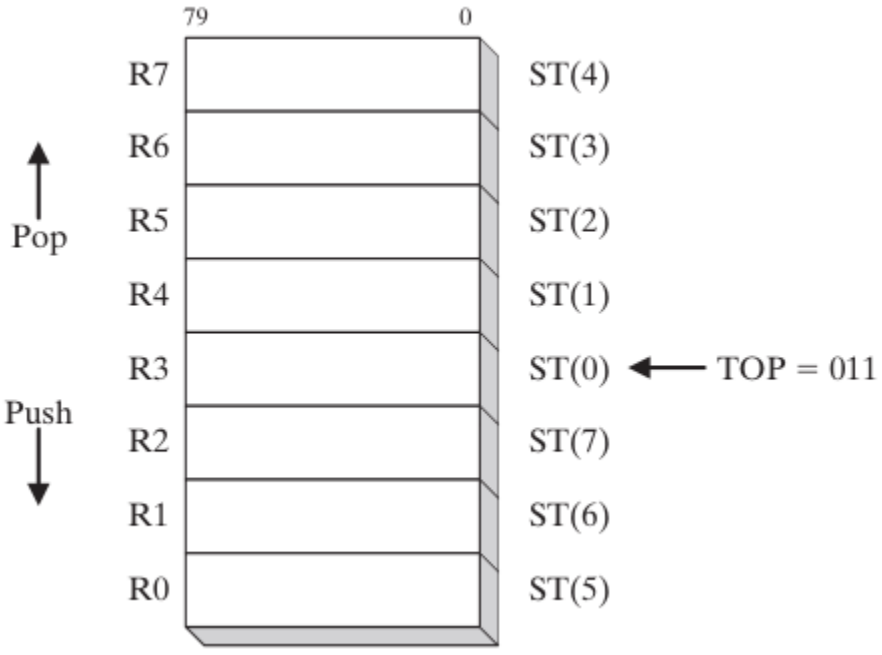
FPU DATA REGISTERS

The FPU has eight individually addressable 80-bit data registers named R0 through R7. Together, they are called a register stack.

The FPU stack works in a **last-in-first-out (LIFO)** manner. This means that the last value pushed onto the stack is the first value popped off the stack.

The top of the FPU stack is indicated by a three-bit field named TOP in the FPU status word. The register at the top of the stack is also known as ST(0).

The rest of the registers are known as ST(1), ST(2), ..., ST(7), in order.

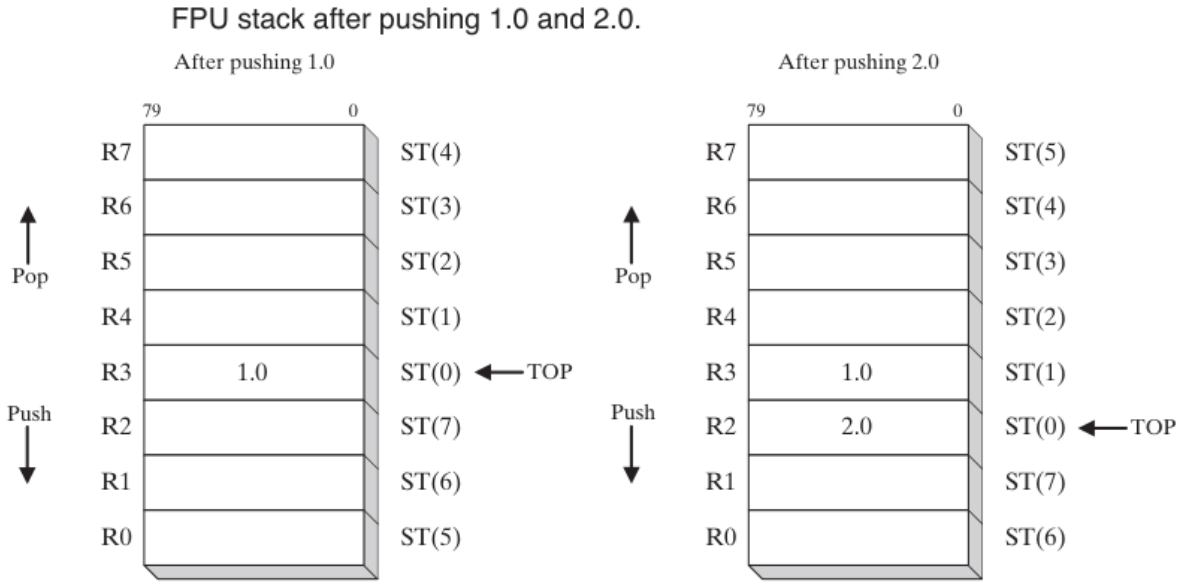


To push a value onto the FPU stack, the FPU decrements TOP by 1 and copies the value into the register identified as ST(0). If TOP equals 0 before a push, TOP wraps around to register R7.

To pop a value off the FPU stack, the FPU copies the data at ST(0) into an operand, then adds 1 to TOP. If TOP equals 7 before the pop, it wraps around to register R0.

If loading a value into the stack would result in overwriting existing data in the register stack, a floating-point exception is generated.

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



To perform a floating-point operation, the FPU pops the required operands off the stack, performs the operation, and pushes the result back onto the stack.

For example, to perform the addition operation 1.0 + 2.0, the FPU would pop the operands 1.0 and 2.0 off the stack, add them together, and push the result, 3.0, back onto the stack.

The FPU stack is a powerful tool for performing floating-point calculations. It allows the FPU to efficiently perform complex operations without having to store intermediate results in memory.

The functionality of the Floating-Point Unit (FPU) in a processor, which is responsible for handling floating-point arithmetic operations. Let's break down some of the key points:

ST(n) Notation:

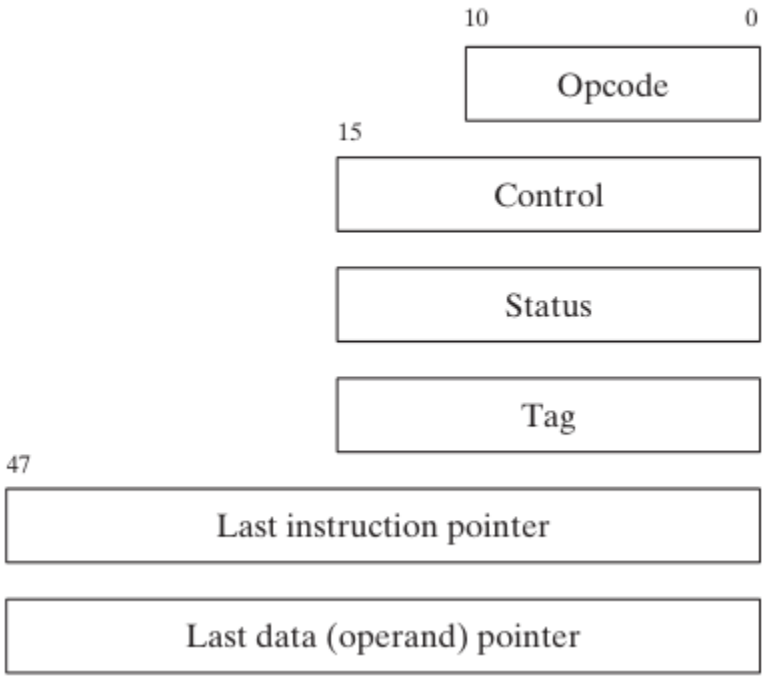
The FPU uses a notation like ST(0), ST(1), and so on to refer to stack registers. ST(0) always represents the top of the stack. This notation is used to address data within the FPU's stack-based architecture.

Floating-Point Value Format:

The floating-point values within FPU registers are stored in the IEEE 10-byte extended real format. This format is sometimes referred to as temporary real.

When the FPU stores the result of an operation in memory, it may translate it into various formats, including integers, long integers, single precision, double precision, or packed binary-coded decimal (BCD).

FPU special-purpose registers.



Special-Purpose Registers: The FPU has several special-purpose registers:

Opcode Register: Stores the opcode of the last non-control instruction executed.

Control Register: Controls the precision and rounding method used by the FPU during calculations. It can also be used to mask individual floating-point exceptions.

Status Register: Contains the top-of-stack pointer, condition codes, and information about exceptions or warnings.

Tag Register: Indicates the content of each register in the FPU data-register stack. It uses two bits per register to specify whether the register holds a valid number, zero, or 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 executed instruction.

ROUNDING IN FPU

The concept of rounding in floating-point calculations and how the Floating-Point Unit (FPU) handles it. I'll break down the key points for better understanding:

Purpose of Special-Purpose Registers: The special-purpose registers mentioned are used by operating systems to preserve the state of the FPU when switching between tasks. This state preservation is crucial for multitasking, where the CPU switches between different tasks.

Rounding in Floating-Point Calculations: The FPU aims to produce precise results from floating-point calculations, but sometimes the destination operand (the format in which the result is stored) can't represent the exact calculated result. Rounding is used to adjust the result to fit within the destination format.

Example of Rounding: Let's take an example where the precise result is **1.0111**, but the destination format can only represent three fractional bits.

Rounding can occur in two ways:

(a) Round up to the next higher value by adding 0.0001: 1.0111 → 1.100

(b) Round down to the closest value by subtracting 0.0001: 1.0111 → 1.011

Similar rounding can be applied for negative values:

(a) -1.0111 → -1.100

(b) -1.0111 → -1.011

Rounding Methods: The FPU provides four rounding methods:

- • • **Round to Nearest Even:** The result is rounded to the nearest value. If two values are equally close, the result is an even value (least significant bit = 0).
- • • **Round Down Toward -∞:** The result is rounded to a value less than or equal to the precise result.
- • • **Round Up Toward +∞:** The result is rounded to a value greater than or equal to the precise result.
- • • **Round Toward Zero (Truncation):** The absolute value of the rounded result is less than or equal to the precise result.

FPU Control Word: The FPU control word contains the RC (Rounding Control) field, which specifies the rounding method to use. The values are binary-encoded as follows:

- • • **00 binary:** Round to nearest even (default).
- • • **01 binary:** Round down toward negative infinity.
- • • **10 binary:** Round up toward positive infinity.

- • • **11 binary:** Round toward zero (truncate).

The default method is **"Round to Nearest Even"**, which is generally considered the most accurate and appropriate for most application programs.

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

Example: Rounding +1.0111.

Method	Precise Result	Rounded
Round to nearest even	1.0111	1.100
Round down toward $-\infty$	1.0111	1.011
Round toward $+\infty$	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 toward $-\infty$	-1.0111	-1.100
Round toward $+\infty$	-1.0111	-1.011
Round toward zero	-1.0111	-1.011

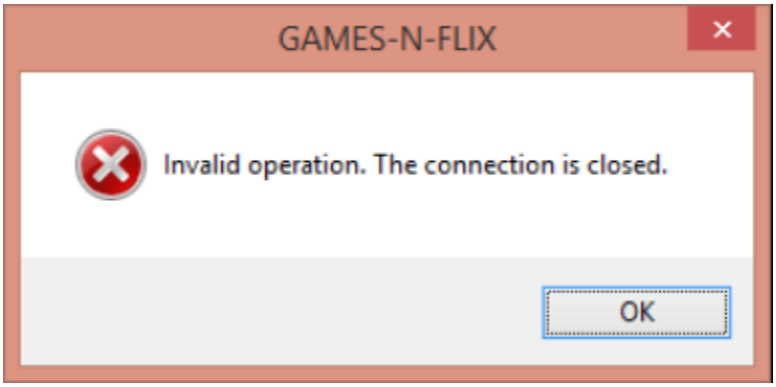
Rounding in floating-point arithmetic is essential to ensure that calculated results fit within the chosen format while minimizing error. The method chosen depends on the specific requirements of the application.

FLOATING POINT EXCEPTIONS

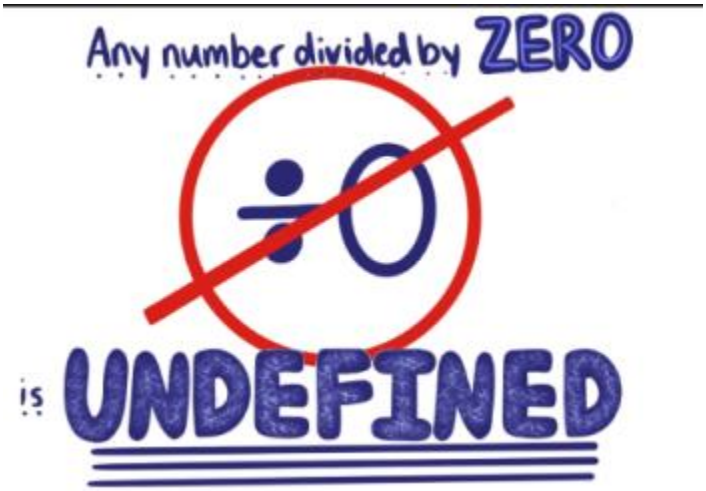
The passage below, describes floating-point exceptions and how the FPU handles them.

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

Invalid operation (#I): This exception is raised when an invalid operation is attempted, such as dividing zero by zero or taking the square root of a negative number.

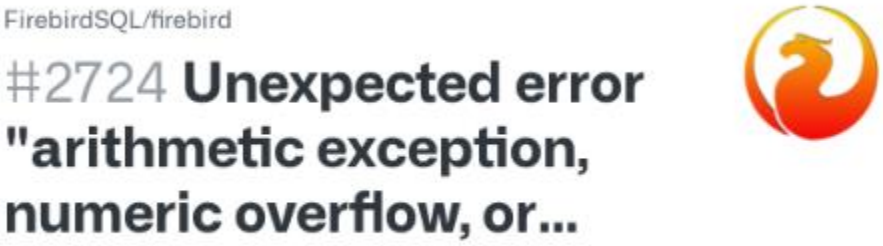


Divide by zero (#Z): This exception is raised when an attempt is made to divide by zero.

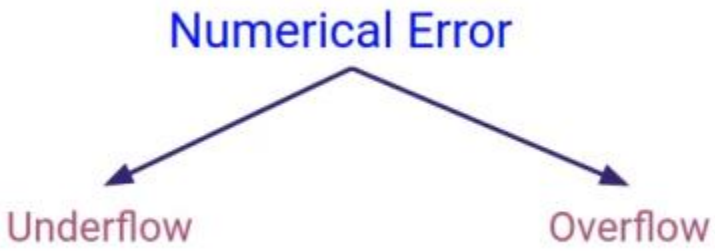


Denormalized operand (#D): This exception is raised when an attempt is made to operate on a denormalized number. A denormalized number is a floating-point number that is very close to zero.

Numeric overflow (#O): This exception is raised when the result of a floating-point operation is too large to be represented by the FPU.



Numeric underflow (#U): This exception is raised when the result of a floating-point operation is too small to be represented by the FPU.



Inexact precision (#P): This exception is raised when the result of a floating-point operation is not exact. This can happen because floating-point numbers are represented by a finite number of bits.

Each exception type has a corresponding flag bit and mask bit. When a floating-point exception is detected, the processor sets the matching flag bit.

The mask bit controls whether or not the processor automatically handles the exception. If the mask bit is set, the processor automatically handles the exception and lets the program continue. If the mask bit is clear, the processor invokes a software exception handler.

The processor's masked (automatic) responses are generally acceptable for most programs. However, custom exception handlers can be used in cases where specific responses are required by the application.

A single instruction can trigger multiple exceptions. The processor keeps an ongoing record of all exceptions occurring since the last time exceptions were cleared. After a sequence of calculations completes, you can check to see if any exceptions occurred.

Here are some examples of how floating-point exceptions can occur:

- Dividing a number by zero will raise a divide by zero exception.
- Taking the square root of a negative number will raise an invalid operation exception.
- Adding two very large numbers together may raise a numeric overflow exception.
- Subtracting two very small numbers from each other may raise a numeric underflow exception.
- Multiplying two numbers together may raise an inexact precision exception if the result is too large to be represented by the FPU.

Floating-point exceptions can cause unexpected behavior in programs. It is important to be aware of the different types of floating-point exceptions and how to handle them.

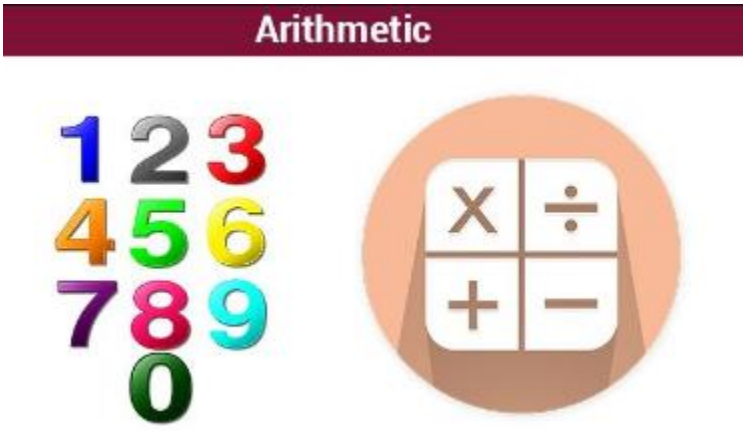
You can use the FPU's mask bits to control how the processor handles floating-point exceptions. You can also write custom exception handlers to handle floating-point exceptions in a specific way.

FLOATING-POINT INSTRUCTION SET

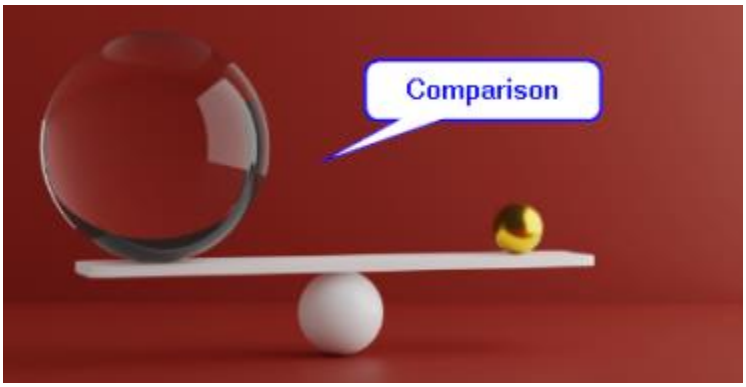
The floating-point instruction set is a set of instructions that the FPU uses to perform floating-point calculations. The instruction set is divided into the following categories:

Data transfer: These instructions move data between the FPU registers and memory.

Basic arithmetic: These instructions perform basic floating-point operations such as addition, subtraction, multiplication, and division.



Comparison: These instructions compare two floating-point numbers and return a result indicating whether one number is greater than, equal to, or less than the other number.



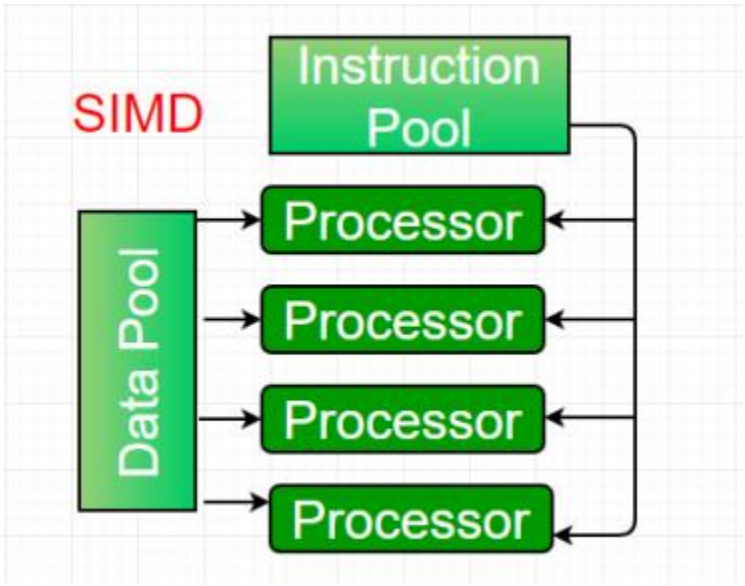
Transcendental: These instructions perform transcendental functions such as sine, cosine, and tangent.

Load constants: These instructions load predefined constants into the FPU registers.

x87 FPU control: These instructions control the behavior of the FPU, such as the rounding mode and the calculation precision.

```
const int mask = 0x0c00

asm(
    "       fldcw                   \n"
    "       fstcw                   \n"
    "       finit                   \n"
    "       fldl    %[mask]         \n"
    "       //& together           \n"
    //set bits 10 & 11

);
```

x87 FPU and SIMD state management: These instructions manage the state of the FPU, such as saving and restoring the FPU register stack.



Floating-point instruction names begin with the letter F.

The second letter of the instruction mnemonic indicates how a memory operand is to be interpreted: **B indicates a BCD operand, and I indicates a binary integer operand.** If neither is specified, the memory operand is assumed to be in real-number format.

Floating-point instructions can have **zero operands, one operand, or two operands**.

If there are two operands, one must be a floating-point register. There are no immediate operands, but certain predefined constants can be loaded into the stack.

Integer operands must be loaded into the FPU from memory before they can be used in floating-point calculations. When storing floating-point values into integer memory operands, the values are automatically truncated or rounded into integers.

To initialize the FPU, you can use the **FINIT instruction.**

The FINIT instruction sets the FPU control word to **037Fh**, which masks all floating-point exceptions, sets rounding to nearest even, and sets the calculation precision to 64 bits.

It is recommended to **call FINIT at the beginning of your programs** so that you know the starting state of 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

When defining memory operands for FPU instructions, you must use one of these data types.

For example, to load a floating-point variable into the FPU stack, the variable must be defined as REAL4, REAL8, or REAL10.

Here is an example of how to use the REAL8 data type to define a floating-point variable and load it into the FPU stack:

```
; 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.
```

This code will load the value stored in bigVal onto the FPU stack, making it ready for further floating-point operations.

The fld instruction loads the value of the variable bigVal into the FPU stack. The bigVal variable is defined as REAL8, so the fld instruction knows how to interpret the data in memory.

The floating-point data types supported by MASM are used in a wide variety of applications, including graphics, scientific computing, and 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:

```
.data
    dblOne    REAL8 234.56
    dblTwo    REAL8 10.1
    intVal    DWORD 100

.code
    ; Load the value of dblOne onto the FPU stack.
    fld     QWORD PTR [dblOne]

    ; Load the value of dblTwo onto the FPU stack.
    fld     QWORD PTR [dblTwo]

    ; Load the integer value intVal onto the FPU stack, preserving its sign.
    fild    DWORD PTR [intVal]

    ; Load the mathematical constant p onto the FPU stack.
    fldpi

    ; Load the constant 1.0 onto the FPU stack.
    fld1
```

In this section, you're defining data variables and loading them onto the FPU (Floating-Point Unit) stack:

dblOne and **dblTwo** are double-precision floating-point variables with values 234.56 and 10.1, respectively. These values are stored in 64 bits each.

intVal is a 32-bit integer variable with a value of 100.

Now, let's explain the code part:

fld QWORD PTR [dblOne]: This instruction loads the value of dblOne (234.56) onto the FPU stack. QWORD PTR indicates that you're accessing a 64-bit value in memory. The fld instruction is used for loading floating-point values.

fld QWORD PTR [dblTwo]: This instruction loads the value of dblTwo (10.1) onto the FPU stack, similar to the previous instruction.

fild DWORD PTR [intVal]: This instruction is different. It loads the 32-bit integer value intVal (100) onto the FPU stack. The fild instruction is used to load integers and preserves the sign. It converts the integer into a floating-point format in the FPU.

fldpi: This instruction loads the mathematical constant π (pi) onto the FPU stack. It's a predefined constant. (we described it above)

fld1: This instruction loads the constant 1.0 onto the FPU stack. It's a predefined constant representing the floating-point value 1.0.

FST AND FSTP INSTRUCTIONS

Here is a concise explanation of the FST and FSTP instructions:

FST (Store Floating-Point Value):

- Copies a floating-point operand from the top of the FPU stack to memory OR
- Stores the value on top of the FPU stack (ST(0)) to memory.
- Supports memory operand types (REAL4, REAL8, REAL10) and can store to another FPU register.
- Does not pop the stack.

The FST instruction in x86 assembly language serves a dual purpose. It can be used to copy a floating-point operand from the top of the FPU stack (ST(0)) to memory. Additionally, it can store the value at the top of the FPU stack directly to memory.

FST supports various memory operand types such as REAL4, REAL8, and REAL10, and it can also be used to store values to another FPU register. Crucially, FST does not pop the stack, leaving the values on the stack intact for further use in calculations.

FSTP (Store Floating-Point Value and Pop):

- Copies the value in ST(0) to memory and pops ST(0) off the stack.
- Stores the value on top of the FPU stack (ST(0)) to memory and pops the stack.
- Supports the same memory operand types as FST.
- Physically removes values from the stack, changing the TOP pointer.

In contrast, the FSTP instruction not only copies the value in ST(0) to memory but also pops (removes) it from the FPU stack. This dual functionality makes it particularly useful for efficient stack management. Similar to FST, FSTP supports the same memory operand types and can store to other FPU registers.

By physically removing values from the stack, FSTP changes the TOP pointer, effectively "popping" the value from the stack, making it a preferred choice when you no longer need the value in the stack for subsequent operations.

Here's a code example illustrating the use of these instructions:

```
.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 code, you're working with the FPU (Floating-Point Unit) to load values onto the FPU stack and then store them in memory using the FST and FSTP instructions. Let's delve into the details without code boxes:

You begin by defining four data variables:

- **dblOne** and **dblTwo** are double-precision floating-point variables with values 234.56 and 10.1, respectively.
- **dblThree** and **dblFour** are initialized as 0.0 but will be used to store values from the FPU stack.

The fld instruction is used to load values onto the FPU stack. You load the value of dblOne (234.56) into ST(0) and then dblTwo (10.1) into ST(0), effectively pushing the previous value down in the stack (ST(1) = 234.56).

You then use the fst instruction to store the value at the top of the FPU stack (ST(0)) into memory without popping the value from the stack. You store this value in dblThree and dblFour, so both variables receive the value 10.1.

To reset the values for demonstration, you reload dblOne and dblTwo onto the FPU stack.

The fstp instruction is employed to store the value from the top of the FPU stack (ST(0)) into memory and simultaneously pop it off the stack.

This effectively removes the value from ST(0) and stores it in dblThree and dblFour. The first fstp stores 10.1 in dblThree, while the second stores 234.56 in dblFour.

The stack's top pointer (TOP) is incremented after each fstp, shifting the remaining values up.

In summary, this code illustrates how the FST and FSTP instructions allow you to move values between the FPU stack and memory.

FST stores the value in memory without altering the stack, while FSTP stores the value in memory and pops it from the stack.

These instructions are essential for efficient floating-point data handling in assembly programming.

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.

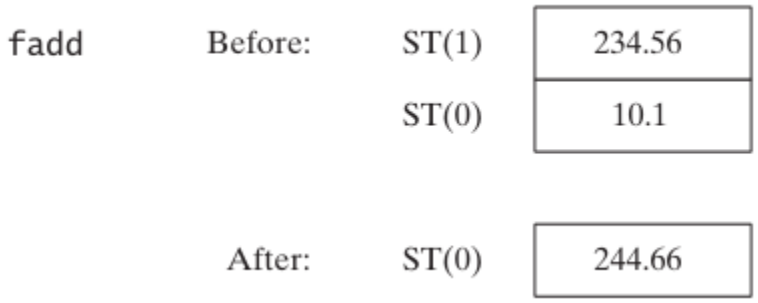
FABS, on the other hand, computes the absolute value by clearing the sign of the value in ST(0).

FADD (Add)

FADD can be used in different formats:

With no operands

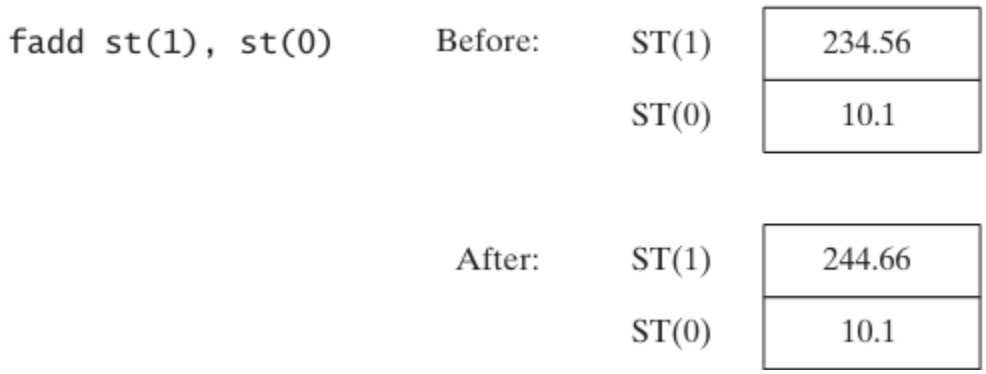
It adds ST(0) to ST(1), storing the result in ST(1). It then pops ST(0), leaving the result on top of the stack.



With a memory operand (REAL4 or REAL8)

It adds the value in memory to the value in ST(0).

With a register number (i), it adds the value in ST(i) to ST(0).



The format **"FADD ST(i), ST(0)"** adds ST(0) to ST(i), effectively **swapping their positions.**

Memory Operand with FADD

When FADD is used with a memory operand, it adds the value stored in memory to ST(0), the top of the FPU stack.

For instance, **"fadd mySingle"** adds the value stored in the memory location "mySingle" to ST(0), effectively increasing its value.

```
.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)
```

In this example, we have defined two floating-point values, mySingle and myDouble, in memory.

We load myDouble onto the FPU stack using the fld instruction, and then we use fadd to add the value of mySingle to the value in ST(0).

Finally, we store the result in the memory location named result using fstp. The program then exits.

Please note that this code is provided for educational purposes and assumes a Linux environment.

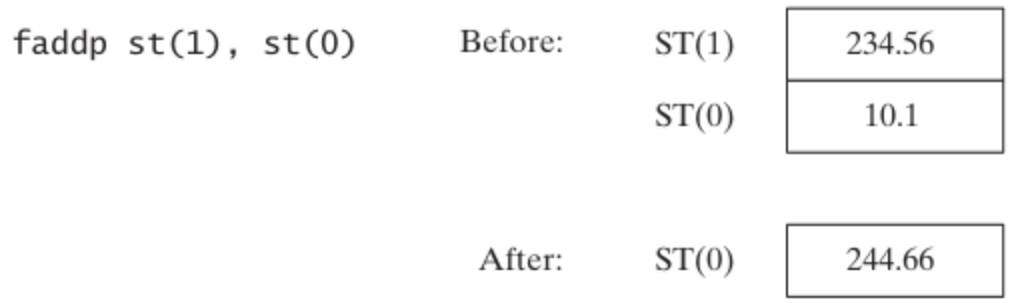
The system call (int 0x80) is specific to Linux, and you may need to adjust it for other operating systems. Additionally, make sure to use the appropriate assembly syntax for your assembler and platform.

FADDP (Add with Pop)

FADDP, the **"add with pop"** instruction, performs an addition operation and then pops the value in ST(0) from the stack.

This is particularly useful for efficiently managing the stack.

The syntax for FADDP is **"FADDP ST(1), ST(0)"**, indicating that the result of the addition is stored in ST(1), and ST(0) is removed from the stack.



FIADD (Add Integer)

The FIADD instruction is used to add an integer value to ST(0) after converting the source operand to double-extended precision floating-point format.

It **supports two operand types: m16int** and **m32int**, which refer to 16-bit and 32-bit integers, respectively.

For example, **"fiadd myInteger"** adds the value stored in the memory location **"myInteger"** to ST(0) after converting it to a floating-point format.

```
.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:

- We define an integer value, myInteger, in memory and a memory location named result to store the result.
- We use the finit instruction to initialize the FPU.
- We load the integer value myInteger onto the FPU stack and convert it to a floating-point value using fild.
- Then, we use fadd to add the value in ST(0) to ST(1), which effectively adds the value of myInteger to the value in ST(0).
- Finally, we store the result in the result memory location using fstp.

The system call (int 0x80) is specific to Linux, and you may need to adjust it for Windows or other operating systems. Make sure to use the appropriate assembly syntax for your platform.

FSUB (Floating-Point Subtract):

The FSUB instruction performs subtraction in the FPU. It subtracts a source operand from a destination operand, storing the difference in the destination operand.

The destination is always an FPU register (ST(0)), and the source can be either an FPU register or memory. It supports the same memory operand types as FADD.

The operation of FSUB is similar to that of FADD, except it performs subtraction.

Without Operands

For example, if used **without operands**, it subtracts ST(0) from ST(1), temporarily storing the result in ST(1). ST(0) is then popped from the stack, leaving the result on top.

Examples:

- fsub mySingle: This instruction subtracts the value in mySingle from ST(0), effectively performing ST(0) -= mySingle.
- fsub array[edi*8]: Here, the value stored in array[edi*8] is subtracted from ST(0). This operation doesn't pop the stack, leaving the result in ST(0).

FSUBP (Floating-Point Subtract with Pop)

The FSUBP instruction is similar to FSUB but with an additional step.

It performs the subtraction and then pops (removes) ST(0) from the stack.

This means that after executing FSUBP, the result is left in ST(0), and the 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 ST(1), ST(0): This instruction subtracts ST(0) from ST(1) and leaves the result in ST(0) while removing the old ST(0) from the stack.

FISUB (Floating-Point Subtract Integer):

The FISUB instruction is used to subtract an integer from a floating-point value in ST(0).

It first converts the source operand (an integer) to double-extended precision floating-point format before performing the subtraction.

This allows you to perform arithmetic operations involving 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
```

These code example show how to use FISUB to subtract an integer from an FPU register and FSUBP to subtract and pop the stack, as well as the explanation you provided for fsubp ST(1), ST(0).

Examples:

fisub m16int: This instruction subtracts the 16-bit integer from ST(0) after converting it to a floating-point format.

fisub m32int: Similarly, this instruction subtracts the 32-bit integer from ST(0) after converting it to a floating-point format.

```
.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.

FMUL (floating-point multiply)

The FMUL (floating-point multiply) instruction is used to multiply a source operand by a destination operand, and the result is stored in the destination operand, which is always an FPU register.

The source can be either an FPU register or a memory operand, similar to FADD and FSUB:

FMUL:

FMUL can be used **with no operands**, in which case it multiplies ST(0) by ST(1), and the result is temporarily stored in ST(1). ST(0) is then popped from the stack, leaving the product in ST(0).

When used **with a memory operand**, FMUL multiplies ST(0) by the value stored in the memory operand.

FMULP (floating-point multiply with pop) is similar to FMUL, but it pops ST(0) from the stack after performing the multiplication.

This means that after executing FMULP, the result is left in ST(0), and the stack pointer is adjusted accordingly.

FIMUL (floating-point integer multiply) is identical to FIADD, except it performs multiplication instead of addition.

It converts the source operand (an integer) to double-extended precision floating-point format before performing the multiplication.

Here's an example in assembly code:

```
.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
```

Let's explain the provided code:

Variable Initialization:

In the .data section, we define several variables. mySingle is set to a single-precision floating-point value (3.0), my16BitInt to a 16-bit integer (2), my32BitInt to a 32-bit integer (4), and result is initialized to store the results of the various multiplications.

Single-Precision Floating-Point Multiplication:

The first part of the code demonstrates single-precision floating-point multiplication. We load the value stored in mySingle onto the FPU stack using fld.

This places the single-precision value in the top of the FPU stack, which is ST(0). We then use the fmul instruction to multiply the value in ST(0) (the value from mySingle) by itself.

The result remains in ST(0). Finally, the fstp instruction is used to store the result in the result variable.

Resetting the FPU Stack:

We reset the FPU stack to a different value (my32BitInt) using another fld instruction. This prepares the FPU stack for the next multiplication operation.

Single-Precision Floating-Point Multiplication (Again):

Similar to the first part, we load a single-precision value onto the FPU stack, this time using fld to place a new value into ST(0).

We then use the fmul instruction to multiply the value in ST(0) by the new single-precision value. The result remains in ST(0).

Once again, the fstp instruction is used to store the result in the result variable.

Resetting the FPU Stack Again:

We reset the FPU stack to a different value (my16BitInt) using yet another fld instruction.

This prepares the FPU stack for the next multiplication operation.

Integer Multiplication with FIMUL:

In this part, we load a 16-bit integer value onto the FPU stack using fld. We then load another single-precision value onto the FPU stack.

However, instead of using fmul as in the previous scenarios, we use the fimul instruction with WORD PTR [esp].

This performs integer multiplication between the 16-bit integer in ST(0) and the single-precision value, and the result remains in ST(0).

Finally, the fstp instruction is used to store the result in the result variable.

Completion and Result Storage:

The code concludes with the result of the integer multiplication stored in the result variable.

This code showcases various scenarios of floating-point and integer multiplication operations in the x86 FPU, demonstrating how to load values, perform multiplications, and store the results for different data types and operand combinations.

FDIV (Floating-Point Division):

The FDIV instruction is used to perform division operations within the x86 FPU.

It divides the value in a destination operand by the value in a source operand, with the result stored in the destination operand.

The destination operand must be an FPU register, while the source operand can be either an FPU register or a memory location.

The syntax is similar to other FPU arithmetic instructions, supporting both memory and register operands.

For example, **fdiv ST(0), ST(i)** divides ST(i) by ST(0), and **fdiv m64fp** divides ST(0) by the value in memory specified by m64fp.

Handling special cases is essential when dealing with division.

For instance, division by zero results in a divide-by-zero exception, so it's crucial to ensure that the divisor is not zero.

Special situations arise when **dividing by infinity**, **zero, or NaN (Not-a-Number)**, each governed by specific rules as detailed in the Intel Instruction Set Reference manual.

FIDIV (Floating-Point Divide Integer):

The FIDIV instruction is used to perform integer division within the x86 FPU.

Before the division operation, it converts the source operand (an integer) to double-extended precision floating-point format, ensuring compatibility with the FPU's floating-point arithmetic.

Syntax examples include **fidiv m16int** for dividing a 16-bit integer and **fidiv m32int** for dividing a 32-bit integer.

The FIDIV instruction is particularly useful when you need to perform arithmetic operations involving integers and floating-point numbers in the FPU.

Now, let's provide a code that showcases various division scenarios, including division by different data types and operand combinations.

```
.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
```

Let's describe the provided assembly code in paragraphs to understand its functionality.

In the provided assembly code, we are performing various division operations using the x86 Floating-Point Unit (FPU).

These operations involve both floating-point and integer division. The code showcases different scenarios of division and stores the results in the quotient variable.

The code begins by defining some data elements in the .data section. It sets up two floating-point variables, dblOne and dblTwo, with respective values.

Additionally, it defines an integer variable, intDivisor, and an empty variable named quotient, which will hold the results of the division operations.

The .code section is where the actual division operations are performed. First, it loads the value of dblOne onto the FPU stack using the fld (load) instruction.

This value represents a double-precision floating-point number.

Next, it performs single-precision floating-point division by dividing ST(0) (top of the stack) by the value of dblTwo using the fdiv (division) instruction.

The result is then stored in the quotient variable using fstp (store and pop).

The code resets the FPU stack by loading dblOne again. This time, it performs double-precision floating-point division by dividing ST(0) by dblTwo.

This demonstrates division with larger precision.

Lastly, the code performs integer division using the FIDIV (integer divide) instruction.

It loads the value of intDivisor onto the FPU stack and divides ST(0) by this integer value. The result is then stored in the quotient variable.

In summary, the code illustrates various division scenarios using the x86 FPU.

It covers single-precision and double-precision floating-point division as well as integer division, providing a comprehensive example of division operations within the FPU.

The results are stored in the quotient variable, making it a versatile demonstration of division in assembly language.

COMPARING FLOATING-POINT VALUES

The passage below describes how to compare floating-point values and branch to a label based on the conditions.

To compare floating-point values, you can use the **FCOM instruction.**

The **FCOM instruction** compares the value in **ST(0) to the source operand**, which can be a memory operand or an FPU register.

After executing FCOM, you can use the **FNSTSW instruction** to move the FPU status word into AX.

The FPU status word contains the condition codes, which indicate the results of the comparison.

Once the **condition codes are in AX**, you can use the SAHF instruction to copy AH into the EFLAGS register.

The **EFLAGS register** contains the CPU status flags, including the Zero, Parity, and Carry flags.

Finally, you can use **conditional jump instructions to branch** to a label based on the condition codes.

For example, the following code branches to the label greater_than if the value in ST(0) is greater than the value in 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
```

In this code, the greater_than label marks the point where you'll continue if the result of the comparison (using FCOM) is greater.

If the result is not greater, you can place your code for handling that case in the section labeled "Your code for handling the case where the comparison result was not greater goes here."

This code sequence loads two values from the FPU stack, compares them using FCOM, transfers the FPU status word into the AX register (to obtain the FPU condition codes), copies the relevant condition code to EFLAGS using SAHF, and then uses the JG instruction to check if the result was greater, jumping to the greater_than label accordingly.

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

The comparison is unordered if either operand is a NaN (Not a Number) or if the two operands have different signs and one of them is zero.

The ability to compare floating-point values and branch to a label based on the conditions is essential for many applications, such as graphics, scientific computing, and financial modeling.

FCOM and FCOMI instruction

Let's describe the FCOMI instruction and how to use it to branch to a label based on the condition codes after comparing two floating-point values.

The FCOMI instruction is a P6 family instruction that compares floating-point values and sets the Zero, Parity, and Carry flags directly. This eliminates the need to use the FNSTSW and SAHF instructions to move the FPU status word into AX and copy AH into the EFLAGS register.

To use the FCOMI instruction, you simply pass the two floating-point values that you want to compare as operands. The FCOMI instruction will then set the Zero, Parity, and Carry flags based on the result of the comparison.

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*)

The unordered condition is a special case that occurs when either operand is a NaN (Not a Number) or if the two operands have different signs and one of them is zero.

In this case, the condition codes are set to 111, which does not correspond to any conditional jump instruction.

Once the condition codes have been set by the FCOMI instruction, you can use conditional jump instructions to branch to a label based on the result of the comparison.

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

```
; Load the value of Y onto the FPU stack
fld Y           ; ST(0) = Y

; Load the value of X onto the FPU stack, creating a stack with X on top and Y below
fld X           ; ST(0) = X, ST(1) = Y

; Compare the values on the FPU stack (X and Y)
fcomi ST(0), ST(1) ; Compare ST(0) to ST(1)

; Jump to label L1 if not below (if X is not less than Y), skipping the next instruction
jnb L1          ; Jump if not below (ST(0) not < ST(1)?), skip to L1

; If the jump condition is not met, set the integer N to 1
mov N, 1        ; N = 1

; Label L1 for reference
L1:
```

Here's a more detailed explanation of the code snippet:

fld Y: This instruction loads the value of Y (the second floating-point number) onto the FPU stack. As a result, ST(0) now contains the value of Y.

fld X: The next instruction loads the value of X (the first floating-point number) onto the FPU stack. This action shifts Y into ST(1), and X occupies ST(0), making it ready for comparison.

fcomi ST(0), ST(1): The fcomi instruction performs a comparison between the values in ST(0) and ST(1). In this case, it compares X (ST(0)) to Y (ST(1)). If ST(0) is not less than ST(1) (in other words, X is not less than Y), it sets the carry flag (CF) to 1.

jnb L1: The jnb (jump if not below) instruction checks the carry flag (CF) and transfers control to the label L1 if CF is not set. In this context, it means that if X is not less than Y, the code will skip the next instruction and move to the label L1.

mov N, 1: When the comparison determines that X is indeed less than Y, the code sets the integer N to 1. This assignment means that N will take the value 1 when the condition is met.

L1:: This label serves as a marker in the code, allowing you to return to this point if needed. In this particular scenario, it signifies the end of the conditional block.

The code snippet effectively compares two floating-point values (X and Y) and, depending on the result, sets the integer N to 1 if X is less than Y.

It demonstrates how the FPU condition codes, particularly the carry flag, can be used to control program flow based on floating-point comparisons.

FCOM compares two floating-point values on the FPU stack and sets condition flags based on the result.

Here's an example that illustrates how to perform a comparison between two double-precision floating-point numbers, X and Y, and then set an integer N based on the result. In this case, if X is less than Y, N is set to 1.

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

.code
    ; if( X < Y )
    ;
    N = 1
    fld X   ; Load X into ST(0)
    fcomp Y   ; Compare ST(0) to Y
    fnstsw ax   ; Move status word into AX
    sahf   ; Copy AH into EFLAGS
    jnb L1   ; X not < Y? Skip
    mov N, 1   ; N = 1
    L1:
```

This code snippet demonstrates how the FPU condition codes are set and how conditional jumps are used to control program flow based on the comparison results.

An improvement to consider is the usage of the FCOMI instruction, available on newer processors like the Intel P6 family (e.g., Pentium Pro and Pentium II).

FCOMI is a more efficient instruction that directly compares two values on the FPU stack and sets condition flags. It's designed for improved performance in comparison operations compared to FCOM.

The FCOMI instruction can perform floating-point comparisons and set the Zero, Parity, and Carry flags directly. Here's 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

The passage you sent describes how to compare floating-point values for equality in assembly language.

The proper way to compare floating-point values for equality is to take the **absolute value of their difference, |x - y|**, and compare it to a small user-defined value called **epsilon.**

This is because floating-point numbers are represented approximately in computers, and even small rounding errors can accumulate over time.

The following code in assembly language compares two floating-point values, val2 and val3, for equality 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 is designed to compare two floating-point values, val2 and val3, for equality while considering a tolerance level defined as epsilon.

The purpose of this code is to address a common challenge when working with floating-point numbers, which is that due to precision limitations, exact equality comparisons may not yield the expected results.

Therefore, it's a common practice to compare values within a certain tolerance range instead.

Here's a step-by-step explanation of the code:

Data Definitions: In the .data section, we define three variables:

- • • **epsilon:** This is set to 1.0E-12, representing the tolerance level for considering two values as equal.
- • • **val2:** This variable holds the first value for comparison, initialized with 0.0.
- • • **val3:** This variable holds the second value, which is considered equal to val2 and is set to 1.001E-13.

Code Section: In the .code section, the actual comparison is performed.

- • • **fld epsilon:** The fld instruction loads the value of epsilon onto the FPU stack. This value represents the maximum allowable difference between two values for them to be considered equal.
- • • **fld val2:** The next instruction loads the value of val2 onto the FPU stack. This sets up the first value for comparison.
- • • **fsub val3:** Here, we subtract the value of val3 from the value of val2. This computes the difference between the two values.
- fabs: The fabs instruction computes the absolute value of the difference obtained in the previous step. This ensures that we're only looking at the magnitude of the difference, regardless of its sign.
- • • **fcomi ST(0), ST(1):** The fcomi instruction performs the actual comparison between the top of the FPU stack (ST(0)) and the next value (ST(1)). It sets certain condition flags based on the result of this comparison. Specifically, it sets the Carry Flag (CF) and Zero Flag (ZF) based on the relationship between the two values.
- • • **ja skip:** The ja instruction is a conditional jump that stands for "jump if above." It checks the CF and ZF flags to determine whether the absolute difference (ST(0)) is greater than the second value (ST(1)), considering the defined tolerance. If the jump condition is met (i.e., if CF=1 and ZF=0), it means the absolute difference is greater than epsilon, and we jump to the skip label.
- • • **mWrite <"Values are equal", 0dh, 0ah>:** If the ja condition is not met, it implies that the absolute difference is within the tolerance range defined by epsilon, and the values are considered equal.

- In this case, the code proceeds to display the message "Values are equal." The mWrite instruction is not a standard x86 assembly instruction but is used here to represent a placeholder for a function that would display the message.
- The 0dh and 0ah are carriage return and line feed characters for formatting.
- • • **skip::** This label is where the code jumps to if the absolute difference is greater than epsilon. It serves as the point to continue execution after the comparison.

In summary, this code demonstrates how to compare floating-point values with a tolerance level (epsilon) to determine if they are equal within a certain range.

It accounts for the precision limitations of floating-point arithmetic and is a common practice when dealing with real-world numerical computations.

The code works summary:

- Load the tolerance value, epsilon, onto the FPU stack.
- Load the first value, val2, onto the FPU stack.
- Subtract the second value, val3, from val2.
- Take the absolute value of the difference.
- Compare the absolute value of the difference to the tolerance value.
- If the absolute value of the difference is greater than the tolerance value, jump to the skip label.
- Otherwise, display the message "Values are equal".
- This code will display the message "Values are equal" if the difference between val2 and val3 is less than or equal to the tolerance value, epsilon. Otherwise, the program will skip over the mWrite instruction and continue execution.

It is important to note that the choice of an appropriate tolerance value depends on the specific application.

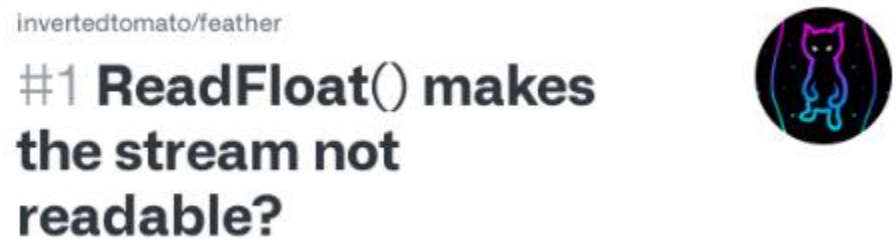
For example, in a graphics application, a tolerance value of $\mathbf{1.0E^{-6}}$ might be sufficient.

However, in a financial modeling application, a tolerance value of $\mathbf{1.0E^{-12}}$ or less might be required.

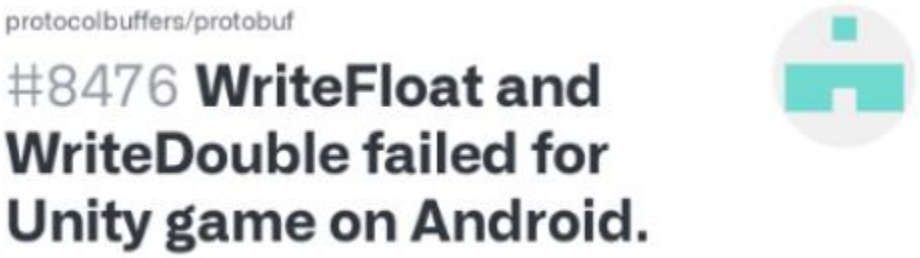
FLOATING-POINT INPUT-OUTPUT PROCEDURES

The passage you sent describes two procedures for floating-point input/output in assembly language:

- • • **ReadFloat:** Reads a floating-point value from the keyboard and pushes it on the floating-point stack.



- • • **WriteFloat:** Writes the floating-point value at ST(0) to the console window in exponential format.



The ReadFloat procedure accepts a wide variety of floating-point formats, including:

35

+35.

-3.5

.35

3.5E5

3.5E005

-3.5E+5

3.5E-4

+3.5E-4

The WriteFloat procedure writes the floating-point value at ST(0) to the console window in exponential format.

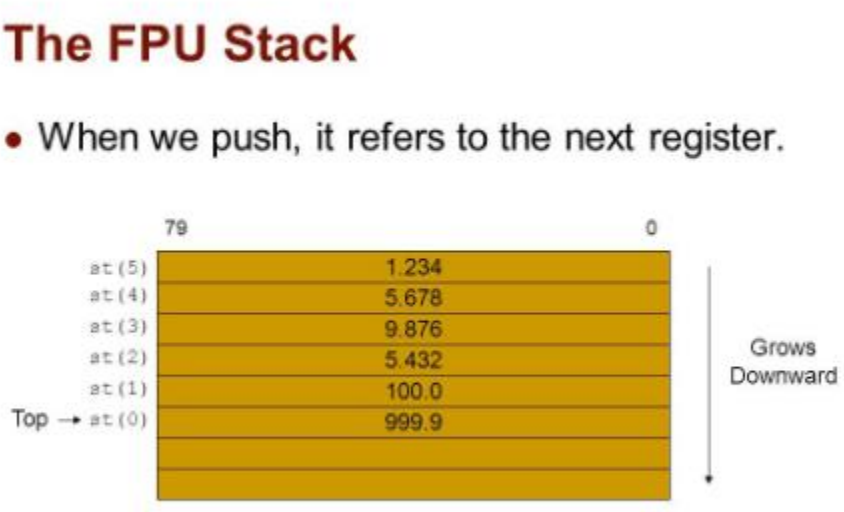
Here's the example program that demonstrates the use of the ReadFloat and WriteFloat procedures in assembly language. This program pushes two floating-point values onto the FPU stack, displays the FPU stack, takes user input for two values, multiplies them, and displays their product. I'll provide you with the assembly code:

```
; 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
```

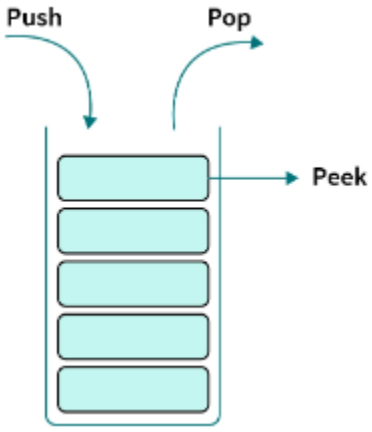
This assembly code is designed to demonstrate the use of floating-point input and output procedures while performing basic arithmetic operations on these floating-point values using the FPU (Floating-Point Unit). Let's break down the code step by step.

Initialization (finit): The program begins by initializing the FPU using the finit instruction. This is a necessary step to prepare the FPU for floating-point operations.

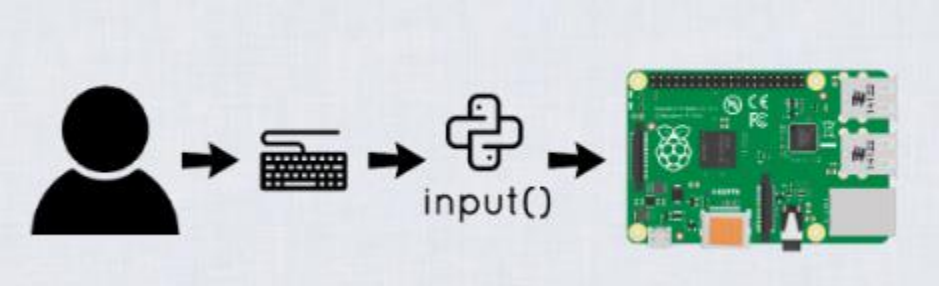
Pushing Values onto the FPU Stack: Two floating-point values are pushed onto the FPU stack. These values are stored in memory as first and second. The fld (floating-point load) instructions are used to load these values onto the FPU stack. fld first pushes the value of first onto the stack, and fld second pushes the value of second onto the stack.



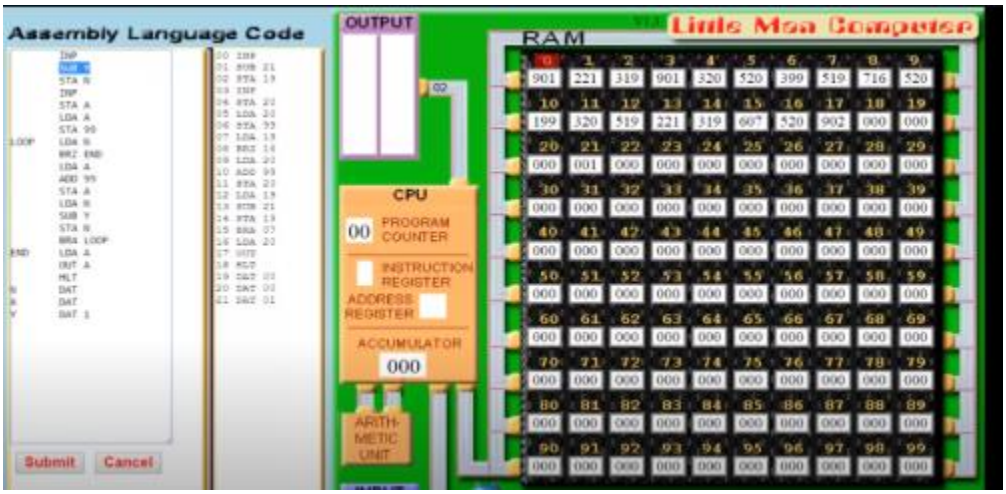
Displaying the FPU Stack: After pushing the values onto the FPU stack, the program calls a custom procedure called ShowFPUStack. This procedure is responsible for displaying the contents of the FPU stack. It helps visualize the values stored on the stack at this point.



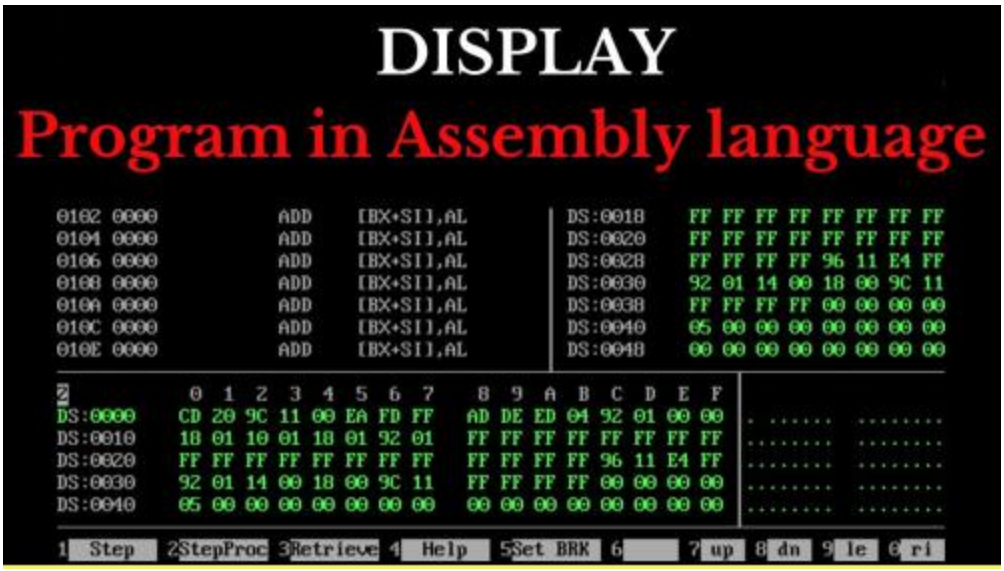
User Input: The program prompts the user to enter two real numbers. It uses the mWrite function to display the input prompt. Then, it calls the ReadFloat procedure, which reads the user's input as a floating-point number. This process is repeated for the second number.



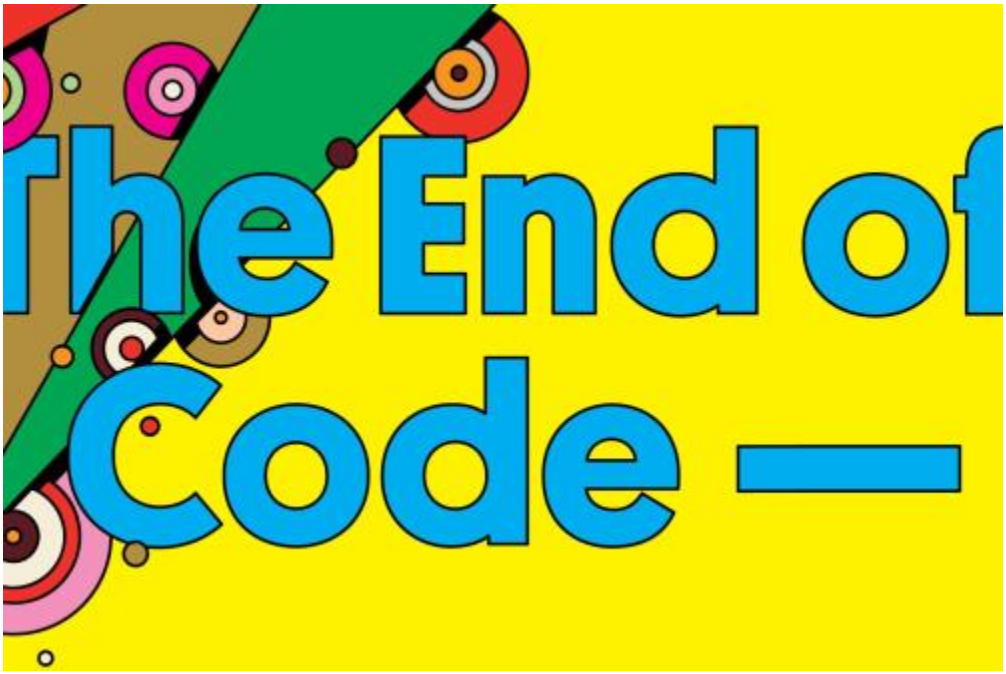
Multiplication (fmul): Once both user inputs are on the FPU stack, the program uses the fmul instruction to multiply these values. The fmul instruction multiplies the value on top of the stack (ST(0)) by the next value (ST(1)) and stores the result in ST(0). In this case, it effectively calculates the product of the two numbers entered by the user.



Displaying the Result: After the multiplication is performed, the program uses mWrite to display the text "Their product is: " to the console. Then, it calls the WriteFloat procedure to display the result of the multiplication in exponential format.



End of Program: Finally, the program adds a line break using Crlf for a clean console output and exits.



The primary purpose of this program is to showcase the handling of floating-point values, input, and output, as well as basic arithmetic operations on these values using the FPU.

The FPU stack is crucial in managing these floating-point values during the operations, and the code illustrates the sequence of actions involved in working with the FPU for floating-point calculations.

EXCEPTION SYNCHRONIZATION

The passage below describes the potential problem of floating-point exceptions in concurrent systems and how to use the WAIT and FWAIT instructions to solve it.

In a concurrent system, two or more tasks can execute at the same time. This can be a problem for floating-point exceptions because the **FPU and the CPU are separate units.**

If an unmasked **floating-point exception occurs while the CPU is executing an integer or system instruction**, the exception will not be handled until the next floating-point instruction or the FWAIT instruction is executed.

This can lead to problems if the floating-point instruction that caused the exception is followed by an integer or system instruction that modifies the same memory operand as the floating-point instruction.

For example, the following code could cause a problem:

```
445 .data
446     intVal DWORD 25   ; Define an integer value
447
448 .code
449     fild intVal        ; Load the integer value into ST(0)
450     inc intVal         ; Increment the integer value
```

This code loads the integer value stored in intVal into the FPU stack register ST(0) and then increments the integer value by 1.

Please note that in this context, the inc instruction increments the integer value stored in memory at the address specified by intVal.

If an unmasked floating-point exception occurs while the FILD instruction is executing, the exception will not be handled until the INC instruction is executed.

If the INC instruction modifies the same memory operand as the FILD instruction, the exception handler will not be able to access the correct value of the memory operand.

The WAIT and FWAIT instructions can be used to solve this problem.

Both instructions force the processor to check for pending, unmasked floating-point exceptions before proceeding to the next instruction.

This ensures that the exception handler will have a chance to execute before any integer or system instructions that modify the same memory operand as the floating-point instruction that caused the exception.

To solve the problem in the example code above, you could add a WAIT or FWAIT instruction after the FILD instruction. For example:

```
.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, the comments are placed next to the relevant instructions.

The FILD instruction loads the integer into the FPU stack, followed by the FWAIT instruction to ensure any pending exceptions are handled.

Then, the INC instruction increments the integer value in memory. This sequence of instructions allows for proper exception handling before modifying the shared memory operand.

This would ensure that the exception handler would have a chance to execute before the INC instruction is executed.

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
```

In this code, we calculate valD as the result of the expression -valA + (valB * valC).

We start by loading valA onto the FPU stack and negating it using fchs.

Then, we load valB into ST(0) and multiply it by valC, resulting in the product in ST(0).

Finally, we add the two values on the stack and store the result in valD.

The fstp instruction pops the result from the stack and stores it in valD.

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)
```

In this code, we are calculating and displaying the sum of an array of double-precision real numbers.

The size of the array is defined by the symbolic constant ARRAY_SIZE, which has a value of 20.

This constant makes the code more maintainable, allowing us to change the array size in a single place if needed.

We initialize a loop counter esi to 0 and push 0.0 onto the FPU stack using fldz. This will be our accumulator for the sum of the array elements.

The loop counter ecx is set to the value of ARRAY_SIZE, which determines the number of elements in the array.

We enter a loop labeled as L1, where we perform the following operations for each element of the array:

Load the current array element into the FPU stack using fld sngArray[esi].

Add the value in the FPU stack to the accumulator (ST(0)) using fadd. This operation also pops the top of the stack.

Move to the next element by adding the size of a REAL8 (8 bytes) to esi using add esi, TYPE REAL8.

The loop continues until all elements of the array have been processed.

After the loop, we call WriteFloat to display the sum of the array elements 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.

```
535 .data
536     valA REAL8 25.0
537     valB REAL8 36.0
538
539 .code
540     fld valA    ; Load valA onto the FPU stack
541     fsqrt       ; Replace ST(0) with the square root of valA
542     fld valB    ; Load valB onto the FPU stack
543     fsqrt       ; Replace ST(0) with the square root of valB
544     fadd        ; Add the two square roots and leave the result in ST(0)
```

We use the FPU instructions to perform the calculation:

fld valA: This instruction loads the value of valA onto the FPU stack, making it the top element (ST(0)) on the stack.

fsqrt: This instruction replaces the value in ST(0) with its square root, so now ST(0) contains the square root of valA.

fld valB: We load the value of valB onto the FPU stack, replacing the previous value in ST(0).

fsqrt: This instruction calculates the square root of the new value in ST(0), which is the square root of valB.

fadd: Finally, we add the two square roots together, and the result is left in ST(0). This means ST(0) now contains the sum of the square roots of valA and valB.

This code efficiently calculates the sum of the square roots of valA and valB, utilizing the FPU instructions to load, compute square roots, and add the results. The final sum is stored 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.

```
550 .data
551     array REAL4 6.0, 2.0, 4.5, 3.2
552
553 .code
554     fld dword ptr [array]     ; Load the first number of the first pair
555     fmul dword ptr [array+4]  ; Multiply with the second number of the first pair
556     fld dword ptr [array+8]   ; Load the first number of the second pair
557     fmul dword ptr [array+12]  ; Multiply with the second number of the second pair
558     fadd                      ; Add the two products in ST(0)
```

fld array: This instruction loads the first number from the "array" into ST(0), which is the top of the FPU stack.

fmul [array+4]: Here, we multiply the value at the memory location array+4 with the value in ST(0). This corresponds to multiplying the first pair (6.0 * 2.0) and leaving the result in ST(0).

fld [array+8]: We load the second number from the second pair (4.5) into ST(0).

fmul [array+12]: This instruction multiplies the value at memory location array+12 (which is 3.2) with the value in ST(0). This is equivalent to multiplying the second pair (4.5 * 3.2) and leaves the result in ST(0).

fadd: Finally, we add the two results in ST(0), which represents the dot product (6.0 * 2.0) + (4.5 * 3.2).

MIXED-MODE ARITHMETIC

The passage below describes mixed-mode arithmetic in assembly language.

Mixed-mode arithmetic is arithmetic involving both integers and reals.

The Intel instruction set provides instructions that promote integers to reals and load the values onto the floating-point stack.

To perform mixed-mode arithmetic in assembly language, you must first use an instruction to promote the integer to a real.

For example, the following instruction promotes the integer in the N register to a real and loads it onto the floating-point stack:

```
fild N
```

Once you have finished performing arithmetic operations on the real, you can use an instruction to store it back to memory as an integer.

For example, the following instruction stores the real in the ST(0) register to the Z integer variable:

```
fist Z
```

It is important to note that the FIST instruction rounds the real to the nearest integer. If you need to truncate the real to an integer, you can use the following code:

```
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
```

In this code, we first store the FPU control word in the variable ctrlWord.

Then, we modify the **RC field** in ctrlWord to set the **rounding mode to truncate.**

After changing the **rounding mode**, we perform the desired arithmetic operations.

After completing the arithmetic operations, we store the FPU control word again and reset the rounding mode to the default setting (round to nearest) by using a bitwise AND operation to clear the bits that control the rounding mode.

Finally, we 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:

```
590 .data
591     N SDWORD 20
592     X REAL8 3.5
593     Z REAL8 ?
594
595 .code
596     fild N    ; Load the integer into ST(0)
597     fadd X    ; Add the memory value (X) to ST(0)
598     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 control word's RC field allows you to specify the rounding mode. You can use the **FSTCW instruction to store the control word in a variable**, **modify the RC field** to change the rounding mode (e.g., truncate or round to nearest), and **then use the FLDCW instruction to load the modified control word back into the FPU.** This allows you to control how rounding is performed during floating-point operations.

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: This includes the Irvine32 library, which provides various I/O and utility functions for assembly language programming.

This section is used for defining data variables.

N SDWORD 20: Declares a signed doubleword (32-bit) integer variable named N with an initial value of 20.

X REAL8 3.5: Declares an 8-byte (64-bit) double-precision real variable named X with an initial value of 3.5.

Z REAL8 ?: Declares another double-precision real variable named Z without an initial value. It will store the result of the computation.

ctrlWord WORD 0: Declares a 16-bit variable named ctrlWord to store the FPU control word.

FPU Initialization:

finit: Initializes the FPU (Floating-Point Unit), ensuring a clean start for FPU operations.

Storing the FPU Control Word:

fstcw ctrlWord: Stores the current FPU control word in the variable ctrlWord. This allows us to modify the control word without affecting the system's default FPU settings.

Modifying the Rounding Mode:

or ctrlWord, 110000000000b: This binary OR operation modifies bits 10 and 11 of the control word to set the rounding control (RC) to truncate. Truncate mode discards the fractional part when converting real numbers to integers.

Mixed-Mode Arithmetic:

fild N: Loads the integer N into the FPU stack, promoting it to a double-precision real value in ST(0).

fadd X: Adds the real X to the value in ST(0), resulting in a mixed-mode addition. The result remains in ST(0).

fstp Z: Stores the result from ST(0) into the double-precision real variable Z.

Displaying the Result:

mov edx, OFFSET Z: Prepares the address of the Z variable for displaying the result.

call WriteFloat: Calls the WriteFloat procedure to display the result stored in Z.

Resetting the Rounding Mode:

and ctrlWord, 001111111111b: This binary AND operation resets bits 10 and 11 of the control word to their default values, effectively setting the rounding mode back to round to nearest even.

Program Exit:

exit: Exits the program.

In summary, this program demonstrates the following concepts:

Changing the FPU control word to set the rounding mode for FPU operations.

Performing mixed-mode arithmetic by adding an integer and a double-precision real value using FPU instructions.

Controlling the rounding mode to ensure the desired behavior for the arithmetic operations.

The program calculates the result of mixed-mode arithmetic and displays it, considering the specified rounding mode, and then resets the rounding mode to its default state before exiting.

UNMASKING FLOATING-POINT EXCEPTIONS

The passage below describes how to mask and unmask floating-point exceptions in assembly language.

Floating-point exceptions are masked by default, which means that when a floating-point exception is generated, the processor assigns a default value to the result and continues executing code.

To unmask a floating-point exception, you must clear the appropriate bit in the FPU control word. The following code unmasks the divide by zero exception:

```
.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
```

This code stores the current FPU control word in ctrlWord, then clears the appropriate bit (bit 2) to unmask the "Divide by zero" exception.

Finally, it loads the modified control word back into the FPU, ensuring that the exception is unmasked for subsequent operations.

Once the divide by zero exception is unmasked, the processor will try to execute an appropriate exception handler.

If no exception handler is installed, the processor will display an error message and terminate 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)
```

This code snippet intentionally divides by zero, which will generate an unmasked exception. Here's how it works step by step:

fild val1: Loads a value from the memory location val1 into the FPU's top stack register ST(0).

fdiv val2: Attempts to divide the value in ST(0) by the value stored in val2. Since the divisor is zero, this operation triggers a "Divide by zero" exception, which is unmasked as per the previous instructions.

fst val2: This instruction is intended to store the result back into the memory location val2. However, it won't be reached due to the unmasked exception generated in the previous step.

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

Q: Write an instruction that loads a duplicate of ST(0) onto the FPU stack.

A: You can load a duplicate of ST(0) onto the FPU stack with the fild ST(0) instruction.

Q: If ST(0) is positioned at absolute register R6 in the register stack, what is the position of ST(2)?

A: When ST(0) is at absolute register R6, ST(2) is positioned at absolute register R4 in the register stack.

Q: Name at least three FPU special-purpose registers.

A: Three FPU special-purpose registers are FSW (Floating-Point Status Word), FSTP (Floating-Point Control), and FTW (Floating-Point Tag Word).

Q: When the second letter of a floating-point instruction is B, what type of operand is indicated?

A: When the second letter of a floating-point instruction is B, it indicates that the instruction operates on BCD (Binary Coded Decimal) operands.

Q: Which floating-point instructions accept immediate operands?

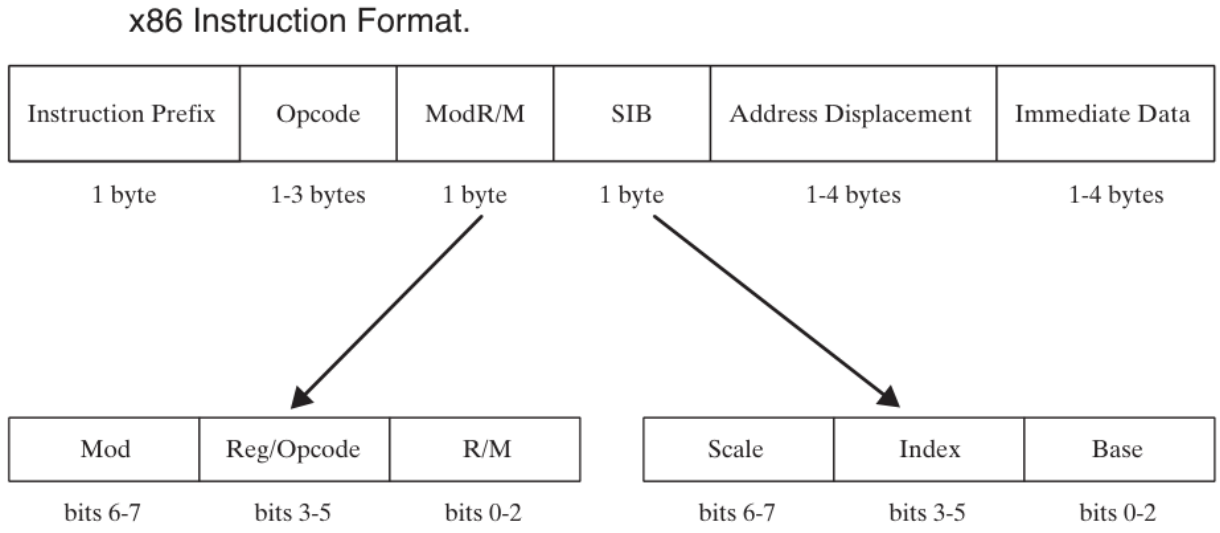
A: Some floating-point instructions that accept immediate operands (constants) include fld1 (load +1.0), fldz (load +0.0), fldpi (load pi), fldl2t (load the base-2 logarithm of 10), and fldl2e (load the base-2 logarithm of e).

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

The instruction prefix byte is a single byte that can be used to override the default operand sizes. For example, the REX prefix byte can be used to override the default operand size from 16 bits to 32 bits.

Opcode

The opcode is a single byte or multiple bytes that identifies a specific variant of an instruction. For example, the opcode for the ADD instruction is 01.

Mod R/M field

The Mod R/M field is a single byte that identifies the addressing mode and operands. The first two bits of the Mod R/M field indicate the addressing mode, and the remaining bits indicate the operands.

Scale index byte (SIB)

The scale index byte (SIB) is a single byte that is used to calculate offsets of array indexes. The SIB byte is used in conjunction with the base and index registers to calculate the effective address of an operand.

Address displacement field

The address displacement field is a single byte or multiple bytes that 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

The immediate data field is a single byte or multiple bytes that holds constant operands. Immediate data operands are used in instructions such as MOV and ADD.

NB: Not all instructions contain all of the fields described above. For example, the ADD instruction only contains the opcode and Mod R/M field.

The x86 machine instruction format is a complex topic, but it is important to understand the basics of the format in order to write assembly language code.

MOD FIELD

The MOD field values shown in the table below are used to specify the addressing mode for the operand in the Mod R/M field. The MOD field is two bits wide, and the possible values are:

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

In the context of x86 assembly language instruction encoding, these are the meanings of the Mod (Mode) values along with their corresponding Displacement (DISP) representations:

Mod 00: DISP = 0

This mode implies that the displacement (address offset) is 0.

Disp-low and disp-high bytes are absent in the encoding, unless the R/M (Register/Memory) field equals 110, in which case the displacement is 32 bits long.

Mod 01: DISP = disp-low, sign-extended to 16 bits; disp-high is absent

In this mode, the displacement is included and consists of the disp-low byte sign-extended to 16 bits.

The disp-high byte is absent.

Mod 10: DISP = disp-high and disp-low are used

In this mode, both the disp-high and disp-low bytes are used to form the displacement.

The full 32-bit displacement is present.

Mod 11: R/M field contains a register number

In this mode, the R/M field does not indicate memory addressing but instead contains a register number.

No displacement is used in this case.

These modes are used to specify how memory operands are addressed and whether or not displacement values are included in the instruction encoding. The specifics of how these modes are used can vary depending on the particular instruction and addressing needs.

If mod field is 11:

If the MOD field is 11, then the R/M field contains the address of the operand. Otherwise, the R/M field contains a register code, and the operand address is calculated using the following formula:

```
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.

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

Let's continue...

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

[a]D16 indicates a 16-bit displacement.

I was using the table above, but it seems this is the new table:

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

The table on 16-bit R/M field values (for MOD = 10) shows the values of the R/M field for each mode. The R/M field is a single byte, and the first two bits of the R/M field indicate the addressing mode, and the remaining bits indicate the operands.

Here is an explanation of each row in the table:

The D16 in the table refers to a 16-bit displacement. The displacement is added to the base register or index register (or both) to calculate the effective address of the operand.

For example, the instruction **MOV EAX, [BX + 10]** has an R/M value of 000. This means that the operand address is calculated by adding the 16-bit displacement 10 to the BX register.

The table also shows that the R/M value 111 is the same as the R/M value 000. This is because the R/M field is only 3 bits wide, so there are only 8 possible values. The R/M value 111 is used to indicate that the base register is BX.

The 16-bit R/M field table is a useful reference for understanding how addressing modes are used in assembly language instructions.

SINGLE BYTE INSTRUCTIONS

The simplest type of instruction is one with either no operand or an implied operand. Such instructions require only the opcode field, the value of which is predetermined by the processor's instruction set.

Table below lists a few common single-byte instructions. It might appear that the INC DX instruction slipped into the table by mistake, but the designers of the instruction set decided to supply unique opcodes for certain commonly used instructions. As a consequence, register increments are optimized for code size and execution speed.

Instruction	Opcode
AAA	37
AAS	3F
CBW	98
LODSB	AC
XLAT	D7
INC DX	42

The table of single-byte instructions that you provided lists the following instructions:

- • • **AAA:** ASCII adjust for addition
- • • **AAS:** ASCII adjust for subtraction

- • • **CBW:** Convert byte to word
- • • **LODSB:** Load byte from string
- • • **XLAT:** Translate
- • • **INC DX:** Increment DX

The **AAA and AAS instructions** are used to adjust the carry flag after performing addition or subtraction operations on ASCII characters.

The **CBW instruction** converts a byte to a word by filling the upper 8 bits of the word with the sign bit of the byte.

The **LODSB instruction** loads a byte from the current position in the string pointer (SI).

The **XLAT instruction** translates a byte using the ASCII translation table in memory.

The **INC DX instruction** increments the DX register by one.

The **INC DX instruction** is included in the table of single-byte instructions because it is a commonly used instruction that is optimized for code size and execution speed.

The designers of the instruction set decided to supply unique opcodes for certain commonly used instructions, such as INC DX, in order to improve the performance of programs that use these instructions frequently.

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

```
687 MOV DX, 10
688 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. By understanding how to use single-byte instructions, you can write efficient and effective assembly language code.

MOV IMMEDIATE TO REGISTER

The passage you sent describes how to encode MOV instructions that move immediate values to registers in x86 assembly language.

The encoding format for a MOV instruction that moves an immediate word into a register is:

```
B8 + rw dw
```

where:

B8 is the opcode for moving an immediate value to a register.

rw is the register code for the destination register.

dw is the immediate word operand, low byte first.

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:

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

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.

For example, the following instruction moves the immediate value 1 to the AX register:

MOV AX, 1

The register code for AX is 0, so the opcode for this instruction is B8 + 0 = B8.

The immediate operand is 1, so the machine code for this instruction is B8 01 00.

The machine instruction is B8 01 00 (in hexadecimal).

Encoding Steps:

The opcode for moving an immediate value to a 16-bit register is B8.

The register number for AX is 0, so 0 is added to B8.

The immediate operand (0001) is appended to the instruction in little-endian order (01, 00).

Another example is the following instruction, which moves the immediate value 1234h to the BX register:

MOV BX, 1234h

The register code for BX is 3, so the opcode for this instruction is B8 + 3 = BB.

The immediate operand bytes are 34 12, so the machine code for this instruction is BB 34 12.

The machine instruction is BB 34 12.

Encoding Steps:

The opcode for moving an immediate value to a 16-bit register is B8.

The register number for BX is 3, so add 3 to B8, producing opcode BB.

The immediate operand bytes are 34 12.

You can practice encoding MOV immediate instructions by hand to improve your skills.

Once you have encoded an instruction, you can check your work by inspecting the code generated by MASM in a **source listing file.**

PUSH CX:

The machine instruction is 51.

Encoding Steps:

The opcode for PUSH with a 16-bit register operand is 50.

The register number for CX is 1, so add 1 to 50, producing opcode 51.

REGISTER MODE INSTRUCTIONS

If you want to understand this subtopic better, check the last line of this subtopic, go to that reference.

The passage you sent describes how to encode MOV instructions that use register operands in x86 assembly language.

The encoding format for a MOV instruction that moves a value from one register to another is

```
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.

The Mod R/M byte contains a 3-bit identifier for each register operand.

The Mod R/M byte is made up of three fields (mod, reg, and r/m). A Mod R/M value of D8, for example, contains the following fields:

mod	reg	r/m
11	011	000

The **Mod field is 11**, indicating that the r/m field contains a register number.

The **reg field is 011**, indicating that the source operand is the BX register.

The **r/m field is 000**, indicating that the destination operand is the AX register.

Therefore, the machine code **89 D8** is for the instruction **MOV AX, BX,** which moves the value in the BX register to the AX register.

The register codes are listed in the following table:

R/M	Register	R/M	Register
000	AX or AL	100	SP or AH
001	CX or CL	101	BP or CH
010	DX or DL	110	SI or DH
011	BX or BL	111	DI or BH

For example, the following instruction moves the value in the BX register to the AX register:

MOV AX, BX

The register code for BX is 3 and the register code for AX is 0, so the machine code for this instruction is 89 D8.

Another example is the following instruction, which moves the value in the CL register to the AL register:

MOV AL, CL

The register code for CL is 1 and the register code for AL is 0, so the machine code for this instruction is 89 C8.

Sample MOV Instruction Encodings, Register Operands.

Instruction	Opcode	mod	reg	r/m
mov ax,dx	8B	11	000	010
mov al,dl	8A	11	000	010
mov cx,dx	8B	11	001	010
mov cl,dl	8A	11	001	010

You can practice encoding MOV instructions that use register operands by hand to improve your skills. Once you have encoded an instruction, you can check your work by inspecting the code generated by MASM in a source listing file.

==========

To calculate the value 89 D8 from the R/M fields, you need to follow these steps:

Identify the addressing mode. The Mod field is 11, which indicates that the addressing mode is register direct.

Identify the source operand. The reg field is 011, which indicates that the source operand is the BX register.

Identify the destination operand. The r/m field is 000, which indicates that the destination operand is the AX register.

Encode the opcode. The opcode for the MOV instruction is 89.

Encode the Mod R/M byte. The Mod R/M byte is encoded as follows:

- **Mod field: 11 (register direct)**
- **reg field: 011 (BX register)**
- **r/m field: 000 (AX register)**

Therefore, the machine code for the instruction MOV AX, BX is 89 D8.

Here is a table that summarizes the encoding of the MOV instruction:

Opcode	Mod R/M byte	Instruction
89	00 /r	MOV register, register
89	01 /r	MOV register, memory
89	10 /r	MOV memory, register
89	11 /r	MOV memory, memory

The reg field and r/m field of the Mod R/M byte identify the source and destination operands, respectively. The Mod field indicates the addressing mode for the operand.

So, this only applies to mov instruction?

Yes, the encoding scheme you described only applies to the MOV instruction. Other instructions may have different encoding schemes.

For example, the ADD instruction has the following encoding scheme:

Opcode	Mod R/M byte	Instruction
01	00 /r	ADD register, register
01	01 /r	ADD register, memory
01	10 /r	ADD memory, register
01	11 /r	ADD memory, memory

The ADD instruction is similar to the MOV instruction, but it adds the source operand to the destination operand instead of moving the source operand to the destination operand.

Different instructions have different encoding schemes because they perform different operations. The encoding scheme for each instruction is specified in the **Intel instruction set reference.**

PROCESSOR OPERAND-SIZE PREFIX

The **operand-size prefix (66h)** is used to override the default segment attribute for the instruction it modifies.

This was necessary because the 8088/8086 instruction set used almost all 256 possible opcodes to handle instructions using 8- and 16-bit operands.

When Intel introduced 32-bit processors, they needed to find a way to invent new opcodes to handle 32-bit operands, yet retain compatibility with older processors.

For programs targeting 16-bit processors, a prefix byte was added to any instruction that used 32-bit operands.

For programs targeting 32-bit processors, 32-bit operands were the default, so a prefix byte was added to any instruction using 16-bit operands. Eight-bit operands need no prefix.

Here is an example of how to use 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 language, the Memory Mode instructions are used to access memory operands with various addressing modes.

These instructions involve loading or storing data between registers and memory locations, with different combinations of registers and memory addresses.

The Mod field in the Mod R/M byte specifies the addressing mode for the memory operand.

Here are some key points about Mod Memory Mode instructions:

Mod R/M Byte: These instructions rely on the Mod R/M byte, which is a part of the instruction encoding, to specify the addressing mode and operands. The Mod field (bits 6-7) in the Mod R/M byte defines the addressing mode.

Mod Field: The Mod field can take on various values:

- **00:** Register-indirect addressing mode. The memory operand is accessed via a register.
- **01:** Displacement-only addressing mode. An immediate value (displacement) is added to the base register.
- **10:** SIB (Scale-Index-Base) addressing mode. This is typically used for complex addressing calculations.
- **11:** Register mode. The operands are registers, not memory.
- **Registers and Memory:** Mod Memory Mode instructions can involve different combinations of registers and memory locations, making them versatile for reading from or writing to memory.

Code Explanation:

Now, let's go over the code provided in the earlier example:

```
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]: This instruction loads the value from the memory address pointed to by SI into the AX register. In the Mod R/M byte, it uses Mod 00 (register-indirect addressing mode) and R/M 101 (SI is the offset address).

mov [si], al: This instruction stores the value in AL into the memory address pointed to by SI. It also uses Mod 00 and R/M 101, as it's effectively performing a register-indirect memory operation using SI as the offset.

add bx, 10h: This instruction adds 10h to the BX register.

mov bx, [bx]: This instruction loads the value from the memory address [BX + 10h] into the BX register. It uses Mod 00 (register-indirect addressing mode) and R/M 010 (BX is the base address with a displacement).

mov [bx + si], cx: This instruction stores the value in CX into the memory address [BX + SI]. It uses Mod 00 (register-indirect addressing mode) and R/M 110 (BX+SI is used as the offset).

These examples illustrate how to use Mod Memory Mode instructions to access memory operands using different addressing modes. The specific Mod R/M byte values indicate the addressing mode and operands used in each instruction.

REPEATING FOR SOME ADDITIONAL INFO:

The Mod R/M byte is a powerful tool that allows x86 assembly language programmers to specify a wide variety of memory-addressing modes.

The Mod field of the Mod R/M byte indicates the group of addressing modes, and the R/M field identifies the specific addressing mode within the group.

The following table shows the Mod R/M bytes for Mod 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]:

This instruction loads the value from the memory address pointed to by SI into the AX register.

Mod 00 indicates register-indirect addressing mode, and R/M 101 signifies SI as the offset address.

The value at memory address [SI] is read into AX.

mov [si], al:

This instruction stores the value in AL into the memory address pointed to by SI.

It uses Mod 00 (register-indirect addressing mode) and R/M 101 (SI is the offset address).

The content of AL is written to the memory location pointed to by SI.

add bx, 10h:

BX is loaded with the immediate value 10h.

The add instruction uses an immediate value to add 10h to the value in BX.

mov bx, [bx]:

This instruction loads the value from the memory address [BX + 10h] into the BX register.

Mod 00 signifies register-indirect addressing, and R/M 010 indicates the use of BX as the base address with a displacement of 10h.

BX now contains the value from memory address [BX + 10h].

mov [bx + si], cx:

This instruction stores the value in CX into the memory address [BX + SI].

Mod 00 indicates register-indirect addressing, and R/M 110 implies that both BX and SI are used as offsets.

The content of CX is written to the memory location at the address [BX + SI].

These examples illustrate how the Mod Memory Mode instructions enable the movement of data between registers and memory locations using different addressing modes.

The Mod R/M byte, as indicated in the comments, is key in specifying 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

Explanation

The Mod R/M byte is a complex topic, but it is essential to understand how to use it in order to write efficient and effective x86 assembly language code. The following is a brief explanation of how to calculate the Mod R/M byte for each of the MOV instructions in the question:

MOV AX, @DATA

The @DATA operand is a symbolic address that represents the global variable DATA. The Mod R/M byte for this instruction is 00 00 00. This is because the DATA operand is a 32-bit absolute address, and the Mod R/M byte for 32-bit absolute addresses is 00 00 00.

MOV DS, AX

The DS register is a segment register that stores the base address of the data segment. The Mod R/M byte for this instruction is 8E D8. This is because the DS register is a special register, and the Mod R/M byte for special registers is 8E. The D8 in the Mod R/M byte indicates that the destination operand is the DS register.

MOV AX, BX

The AX and BX registers are general-purpose registers. The Mod R/M byte for this instruction is 00 10. This is because the BX register is register number 3, and the Mod R/M byte for register-to-register moves is 00. The 10 in the Mod R/M byte indicates that the source operand is the BX register.

MOV BL, AL

The BL and AL registers are general-purpose registers. The Mod R/M byte for this instruction is 00 10. This is because the AL register is register number 0, and the Mod R/M byte for register-to-register moves is 00. The 10 in the Mod R/M byte indicates that the source operand is the AL register.

MOV AL, [SI]

The AL register is a general-purpose register, and the [SI] operand is a memory operand. The Mod R/M byte for this instruction is 00 36. This is because the SI register is register number 6, and the Mod R/M byte for memory operands is 00. The 36 in the Mod R/M byte indicates that the base register is the SI register.

MOV MYBYTE, AL

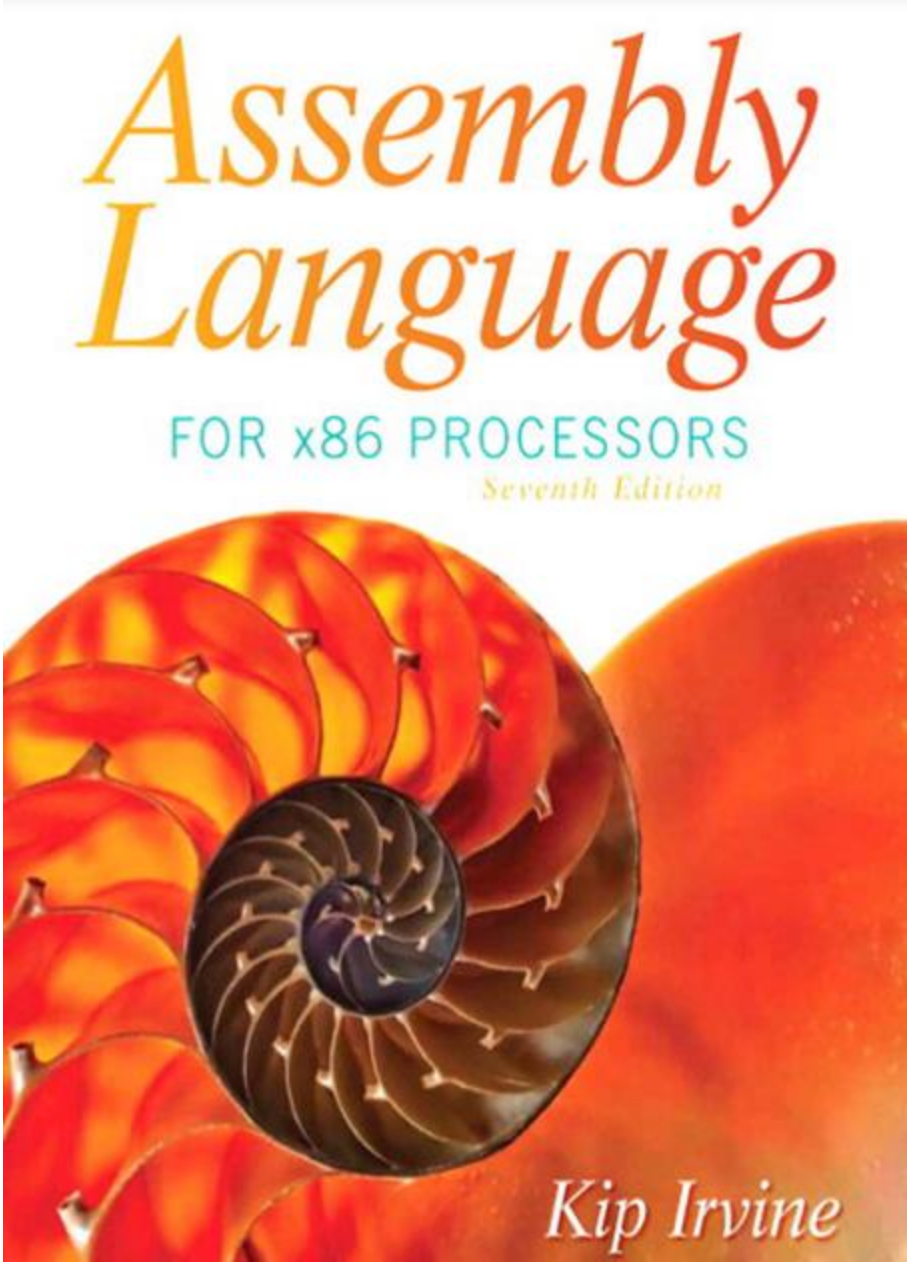
The MYBYTE operand is a global variable, and the AL register is a general-purpose register. The Mod R/M byte for this instruction is 88 44 01. This is because the MYBYTE operand is a memory operand, and the Mod R/M byte for memory operands is 00. The 44 in the Mod R/M byte indicates that the destination operand is the MYBYTE operand. The 01 in the Mod R/M byte indicates that the destination operand is a byte.

MOV MYWORD, AX

The MYWORD operand is a global variable, and the AX register is a general-purpose register. The Mod R/M byte for this instruction is 89 84 01. This is because the MYWORD operand is a memory operand, and the Mod

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

This is the book!

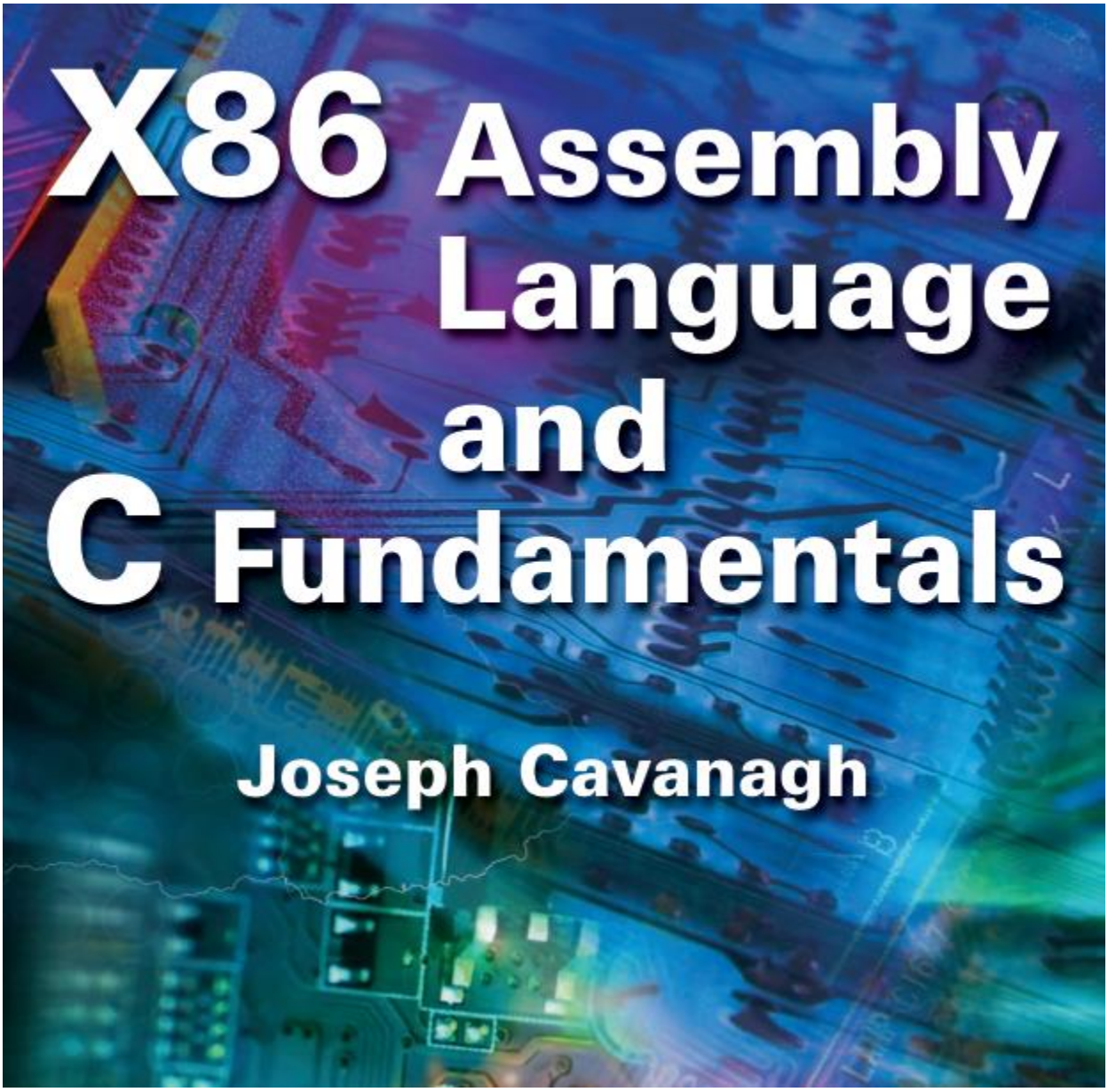


Reasons why did I decide to skip BIOS level programming?

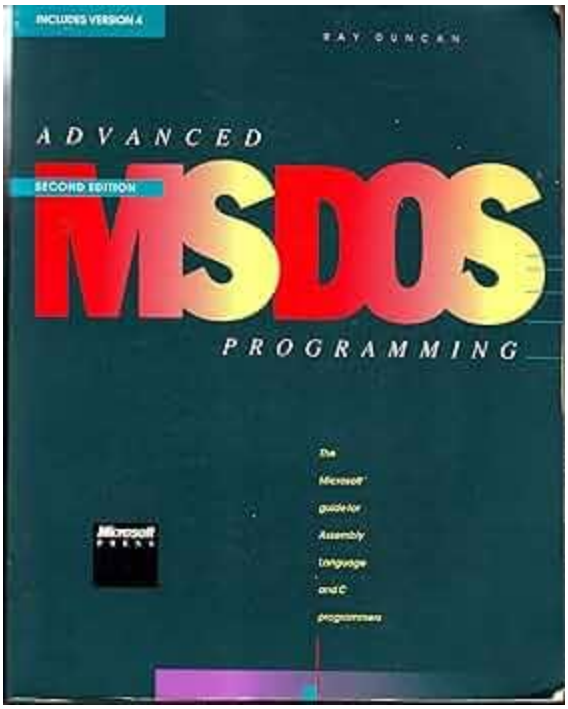
- Not just BIOS level programming, but BIOS Level programming in ASSEMBLY. 💀💀That's a cocktail for death!
- I heard BIOS level programming is like trying to read hieroglyphs from an ancient computer tomb. I'll stick to modern code, no thank you.
- My computer programming philosophy: "No BIOS level, no problem!" 🚀

The rest of the topics are super advanced and I haven't really got a grasp of assembly to a level that I can deal with the following topics that remain:

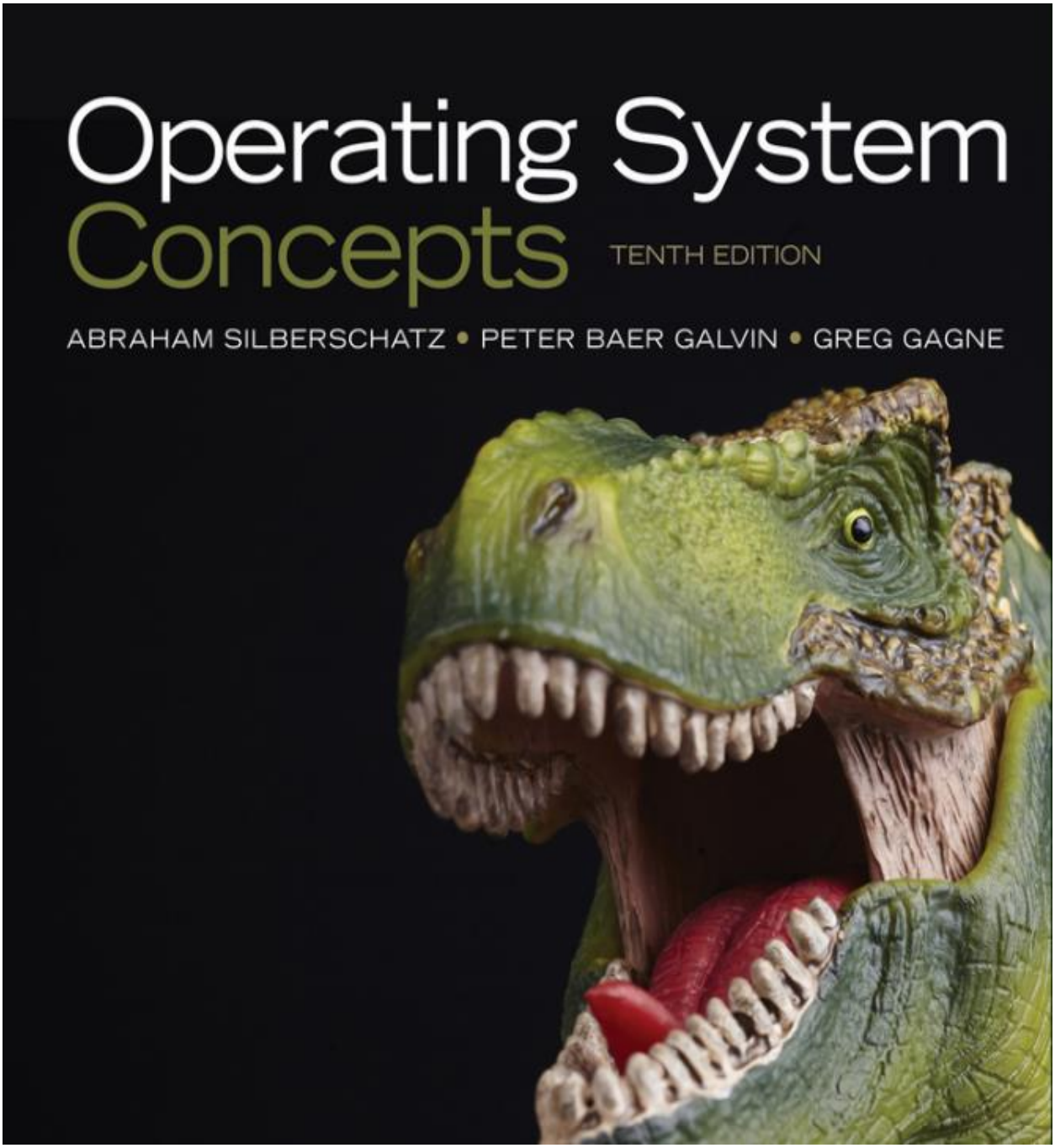
Assembly with C/C++: I will use this book later on in life ...😄



16-bit MS-DOS programming, I don't think I need that topic right now... or even in the future. It's a topic to learn for fun maybe...



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



What about MS-DOS??

- Coz Advanced MS-DOS programmers never get lost; they just change directories! Getting lost is a necessity in life for me
- Why did the advanced MS-DOS programmer bring a ladder to work? To reach the higher memory!
- Advanced MS-DOS programmers don't play hide and seek; they play "dir and find"!

- Why did the programmer use MS-DOS for his love letter? Because he wanted it to be "SYS-tematically" romantic!
- Advanced MS-DOS programmers never argue; they just "batch" their differences! patch .... 💀
- MS-DOS programmers tell the best bedtime stories: "Once upon a time in a directory far, far away..."

==========

Greetings, Future Code Conjurers,

As I prepare to take my leave, let's reminisce about the wild ride we've had in the enchanting realm of assembly programming and operating systems. It's been a journey filled with more 0s and 1s than a binary-themed disco party, but we've managed to decode the secrets of this digital universe. 🕺💃

Within my treasure chest of meticulously updated notes and code, updated @ 2023, you'll find the keys to the kingdom of coding. It's like having a GPS for the labyrinth of algorithms and a magic wand to summon those elusive syscalls. 🗝️🌐

Now, as you venture into the unpredictable landscape of real-world programming, remember that unlike the sterile environment of your IDE, reality is like a zoo. It's got bugs of all shapes and sizes, crashes that pop up like surprise parties, and sometimes, even blue screens, which are like the ultimate "You shall not pass" barriers. 🚫🚫

But fear not! With this repository of knowledge, you'll wield debugging powers that even Thor's hammer would envy. 💪🔨 And should you ever feel stuck, just remember: Ctrl + Z is your magic "Undo" spell, and Google is your friendly wizard mentor.

As you continue this epic adventure, may your code compile smoother than butter on a hot pancake, and may your systems run as smoothly as a well-oiled robot doing the moonwalk. 🤖🧈🤛

Now, as I make my exit, I can't help but grin at the thought of all the fantastic and, dare I say, "punny" code you'll create. Your journey is just beginning, and the tech universe is your playground. So go forth and code like it's the last line of defense against a robot uprising! 🌍🦾

Farewell, my fellow jesters of Java, sorcerers of C, and assembly alchemists. The tech world is your oyster; go forth and code like a wizard casting spells.

My fellow wizards and tech explorers, until we meet again in the binary dreamscape of ones and zeros, keep smiling, keep coding, and may the tech odds be ever in your favor. 😄👨‍💻👨‍💻🚀

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**

A data definition statement in assembly language is used to reserve memory for a variable and to specify its data type. The general syntax for a data definition statement is as follows:

[label] directive value

Where:

- **label** is an optional name for the variable.
- **directive** is the data type of the variable.
- **value** is the initial value of the variable.

For example, the following data definition statement reserves 4 bytes of memory for a variable named count and initializes it to the value 12345:

count DWORD 12345

int count = 12345;

So, **label is variable, directive is the datatype, and value is the value.**

The following are some other examples of data definition statements in assembly language:

message DB "Hello, world!"

age BYTE 25

salary SDWORD 100000

The **DUP operator takes two arguments: a count and a value.** The count is an integer expression that specifies the number of times to duplicate the value.