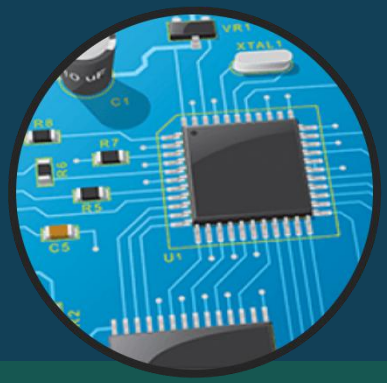
Floating-Point Processing and Instruction Encoding



Topic # 11

Fall 2019

[ref]: Irvine, Kip R. Assembly Language for x86 Processors

Engr. Taufique-ur-Rehman taufique.rehman@seecs.edu.pk
Muhammad Imran Abeel imran.abeel@seecs.edu.pk

Contents

- Floating-Point Binary Representation
- Floating-Point Unit
- x86 Instruction Encoding

Irvine, Kip R. Assembly Language for x86 Processors

Floating-Point Binary Representation

- IEEE Floating-Point Binary Reals
- The Exponent
- Normalized Binary Floating-Point Numbers
- Creating the IEEE Representation
- Converting Decimal Fractions to Binary Reals

IEEE Floating-Point Binary Reals

Types

Single Precision

32 bits: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional part of the significand.

Double Precision

64 bits: 1 bit for the sign, 11 bits for the exponent, and 52 bits for the fractional part of the significand.

Double Extended Precision

80 bits: 1 bit for the sign, 16 bits for the exponent, and 63 bits for the fractional part of the significand.

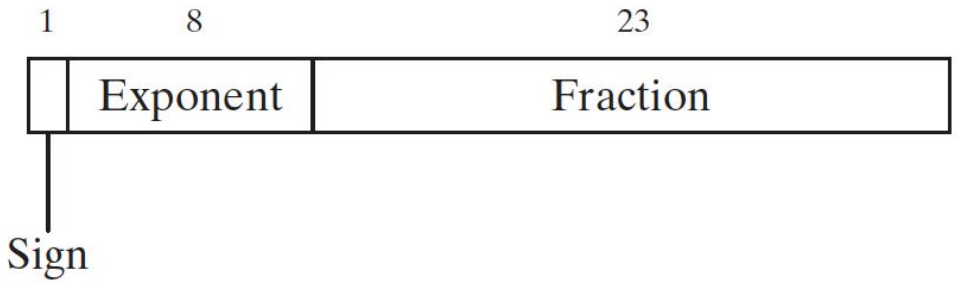
Single Precision

Approximate normalized range:

2^{-126} to 2^{127}

Also called a *short real*

FIGURE 12–1 Single-precision format.



Components of a Single-Precision Real

Sign

1 = negative, 0 = positive

Significand

decimal digits to the left & right of decimal point

weighted positional notation

Example:

$$123.154 = (1 \times 10^{2}) + (2 \times 10^{1}) + (3 \times 10^{0}) + (1 \times 10^{-1}) + (5 \times 10^{-2}) + (4 \times 10^{-3})$$

Exponent

unsigned integer

integer bias (127 for single precision)

Components of a Single-Precision Real

$$\mathbf{11.1011}$$

$$= (1 \times 2^{1}) + (1 \times 2^{0}) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$$

Sum of fractions

$$\mathbf{.1011} = \frac{1}{2} + \frac{0}{4} + \frac{1}{8} + \frac{1}{16}$$

$$= \frac{\mathbf{11}}{\mathbf{16}}$$

- The Decimal Numerator (11) represents the binary bit pattern 1011
- If **e is the number of significant bits to the right of the binary point**, the decimal denominator is 2e
- In our example, **e = 4**, so 2e = 16

Decimal Fractions vs Binary Floating-Point

$$11.1011$$
$$= (1 \times 2^{1}) + (1 \times 2^{0}) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) + (1 \times 2^{-4})$$

$$.1011 = \frac{1}{2} + \frac{0}{4} + \frac{1}{8} + \frac{1}{16} = \frac{11}{16}$$

Table 12-2 Examples: Translating Binary Floating-Point to Fractions.

Binary Floating-Point	Base-10 Fraction
11.11	3 3/4
101.0011	5 3/16
1101.100101	13 37/64
0.00101	5/32
1.011	1 3/8
0.00000000000000000000001	1/8388608

Table 12-3 Binary and Decimal Fractions.

Binary	Decimal Fraction	Decimal Value
.1	1/2	.5
.01	1/4	.25
.001	1/8	.125
.0001	1/16	.0625
.00001	1/32	.03125

The Exponent

- Sample Exponents represented in Binary
- Add 127 to actual exponent to produce the biased exponent

Table 12-4 Sample Exponents Represented in Binary.

Exponent (E)	Biased (E + 127)	Binary
+5	132	10000100
0	127	01111111
−10	117	01110101
+127	254	11111110
−126	1	00000001
−1	126	01111110

[9]

Normalizing Binary Floating-Point Numbers

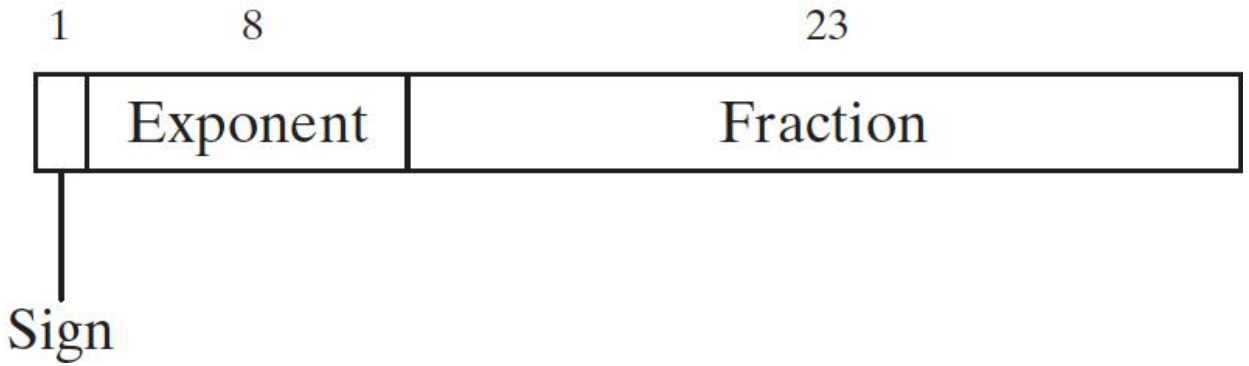
- Mantissa is normalized, when a single 1 appears to the left of the binary point
- Un-normalized: shift binary point until exponent is zero
- Examples

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

Creating IEEE Presentation

Once the Sign Bit, Exponent and Significand fields are Normalized and Encoded, it's easy to generate a complete binary IEEE Short Real

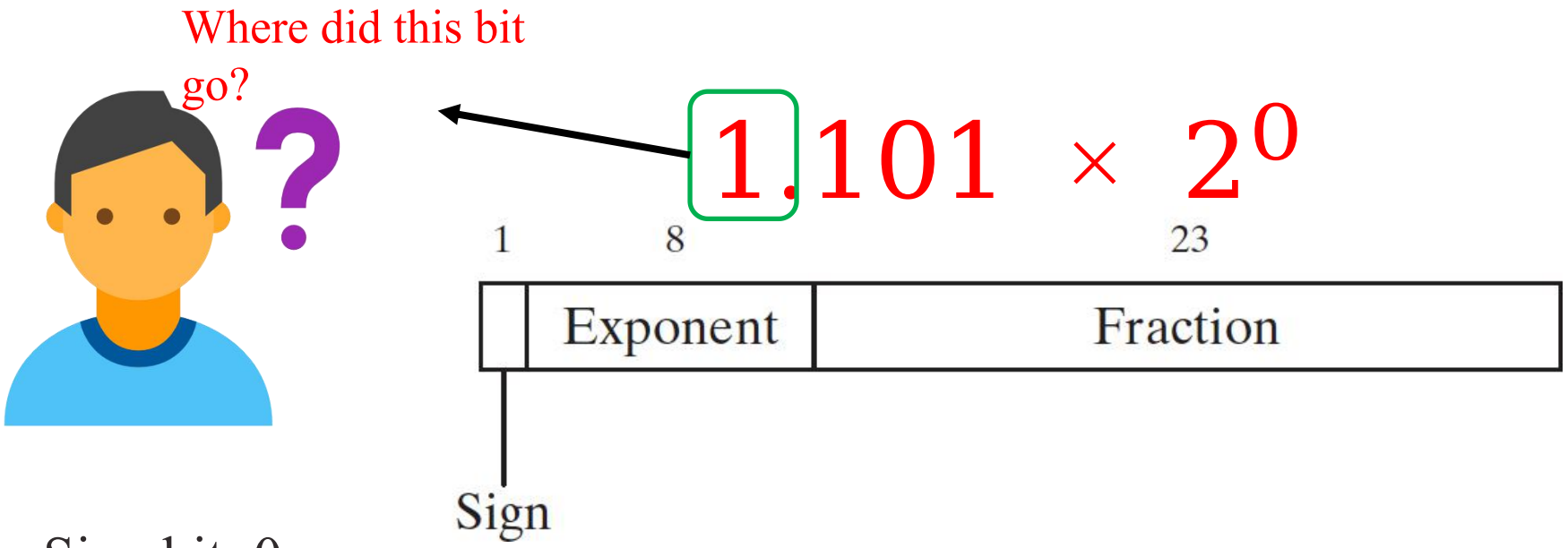
$$1.101 \times 2^{0}$$



- Sign bit: 0
- Exponent: 01111111 (biased- exponent)
- Fraction: 10100000000000000000000

Creating IEEE Presentation

Once the Sign Bit, Exponent and Significand fields are Normalized and Encoded, it's easy to generate a complete binary IEEE Short Real



- Sign bit: 0
- Exponent: 01111111 (biased- exponent)
- Fraction: 10100000000000000000000

Examples

Table 12-5 Examples of Single Precision Bit Encodings.

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

Real-Number Encodings

- **Normalized Finite Numbers**
  - all the nonzero finite values that can be encoded in a normalized real number between zero and infinity
- Positive and Negative Infinity

- NaN (Not a Number)
  - Bit Pattern that is not a valid FP value

  - Two Types of NaN:
    - Quiet NaN
    - Signaling NaN

Real-Number Encodings

- **Normalized Finite Numbers**
  - All the nonzero finite values that can be encoded in a normalized real number between zero and infinity
- **Positive and Negative Infinity**

- **NaN (not a number)**
  - Bit Pattern that is not a valid FP value

- Two Types:
  - Quiet Nan
  - Signaling Nan

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 Fractions to Binary Reals

- Express as a sum of fractions having denominators that are powers of 2
- Examples: Decimal 0.5 as fraction 5/10
- Decimal 5 is binary 0101, and decimal 10 is binary 1010

Binary long division

```
          .1
      ____________
1010 | 0101.0
     | −1010
     |-------
          0
```

Table 12-7 Examples of Decimal Fractions and 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

Converting Single-Precision to Decimal

1. If the **MSB is 1, the Number is Negative**; otherwise, it is Positive

2. The next **8 Bits represent the Exponent**
   - Subtract Binary 01111111 (Decimal 127), producing the **Unbiased exponent**
   - Convert the Unbiased Exponent to Decimal.

3. The next **23 Bits represent the Significand**
   - Notate a "**1.**", followed by the significand bits
   - Trailing Zeros can be ignored
   - Create a Floating-point Binary Number, using the Significand, the Sign determined in step 1, and the Exponent calculated in step 2

4. Denormalize the binary number produced in step 3, i.e. **Shift the binary point to the number of places equal to the value of the exponent**
   - Shift right if the exponent is positive, or left if the exponent is negative

5. From left to right, **use Weighted Positional Notation to form the decimal sum of the powers of 2 represented** by the floating-point binary number

Example:

Convert IEEE *0 10000010 01011000000000000000000* to Decimal.

$$1.xxxxx \times 2^{+x}$$

1. The number is Positive

Example:

Convert IEEE *0 10000010 01011000000000000000000* to Decimal.

$$1.xxxxx \times 2^{+3}$$

$$10000010 - 01111111 = 00000011$$

1. The number is Positive
2. The unbiased exponent is Binary 00000011, or Decimal 3

Example:

Convert IEEE *0 10000010 01011000000000000000000* to Decimal.

$$1.01011 \times 2^{+3}$$

1. The number is Positive
2. The unbiased exponent is Binary 00000011, or Decimal 3
3. Combining the Sign, Exponent and Significand, the Binary Number is $+1.01011 \text{ X } 2^3$

Example:

Convert IEEE 0 10000010 01011000000000000000000 to Decimal

$$1.01011 \times 2^{+3}$$

1. The number is positive
2. The unbiased exponent is binary 00000011, or decimal 3
3. Combining the Sign, Exponent and Significand, the Binary Number is $+1.01011 \text{ X } 2^3$
4. The un-normalized Binary Number is +1010.11

Example:

Convert IEEE 0 10000010 01011000000000000000000 to Decimal

$$1.01011 \times 2^{+3}$$

1. The number is positive
2. The unbiased exponent is binary 00000011, or decimal 3
3. Combining the Sign, Exponent and Significand, the Binary Number is $+1.01011 \text{ X } 2^3$
4. The un-normalized Binary Number is +1010.11
5. The Decimal value is +10 3/4, or +10.75

Contents

- Floating-Point Binary Representation
- Floating-Point Unit
- x86 Instruction Encoding

Floating Point Unit

- FPU Register Stack
- Rounding
- Floating-Point Exceptions
- Floating-Point Instruction Set
- Arithmetic Instructions
- Comparing Floating-Point Values
- Reading and Writing Floating-Point Values
- Exception Synchronization
- Mixed-Mode Arithmetic
- Masking and Unmasking Exceptions

FPU Register Stack

- The FPU does not use the general-purpose registers (EAX, EBX, etc.)
- Instead, it has its own set of registers called a Register Stack

- It loads values from Memory into the Register Stack, performs calculations, and stores stack values into Memory

FPU instructions evaluate Mathematical Expressions in Postfix Format

FPU Register Stack

FPU instructions evaluate Mathematical Expressions in Postfix Format

Infix Format

Postfix Format

(A + B) × C

FPU Register Stack

FPU instructions evaluate Mathematical Expressions in Postfix Format

Infix Format

$(A + B) \times C$

Postfix Format

$(A + B) \times C$

FPU Register Stack

FPU instructions evaluate Mathematical Expressions in Postfix Format

Infix Format

(A + B) × C

Postfix Format

A B + C ×

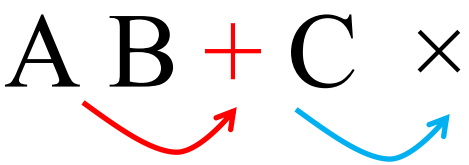
FPU Register Stack

FPU instructions evaluate Mathematical Expressions in Postfix Format

Infix Format

$(A + B) \times C$

Postfix Format



FPU Register Stack

FPU instructions evaluate mathematical expressions in postfix format

Infix Format	Postfix Format
$(A + B) \times C$	$A\ B\ +\ C\ \times$
$(5 \times 6) + 4$	? ? ?

FPU Register Stack

FPU instructions evaluate mathematical expressions in postfix format

Infix Format	Postfix Format
$(A + B) \times C$	$A\ B\ +\ C\ \times$
$(5 \times 6) + 4$	$5\ 6\ \times\ 4\ +$

FPU Register Stack

FIGURE 12–2 Evaluating the postfix expression 5 6 * 4 – .

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

FPU Register Stack

Table 12-8 Infix to Postfix Examples.

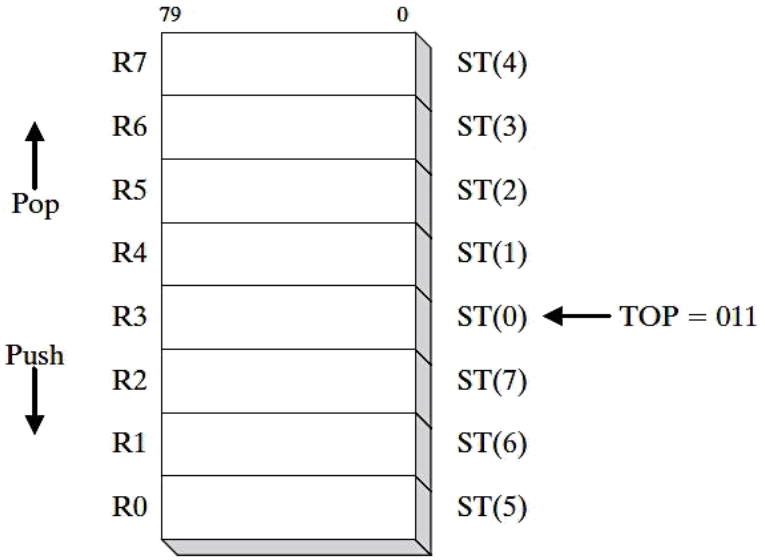
Infix	Postfix
A + B	A B +
(A - B) / D	
(A + B) * (C + D)	
((A + B) / C) * (E - F)	

FPU Register Stack

- Eight individually addressable 80-bit data registers named R0 through R7

- Three-bit field named TOP in the FPU status word identifies the register number that is currently the top of stack

Figure 12–3 Floating-point data register stack.

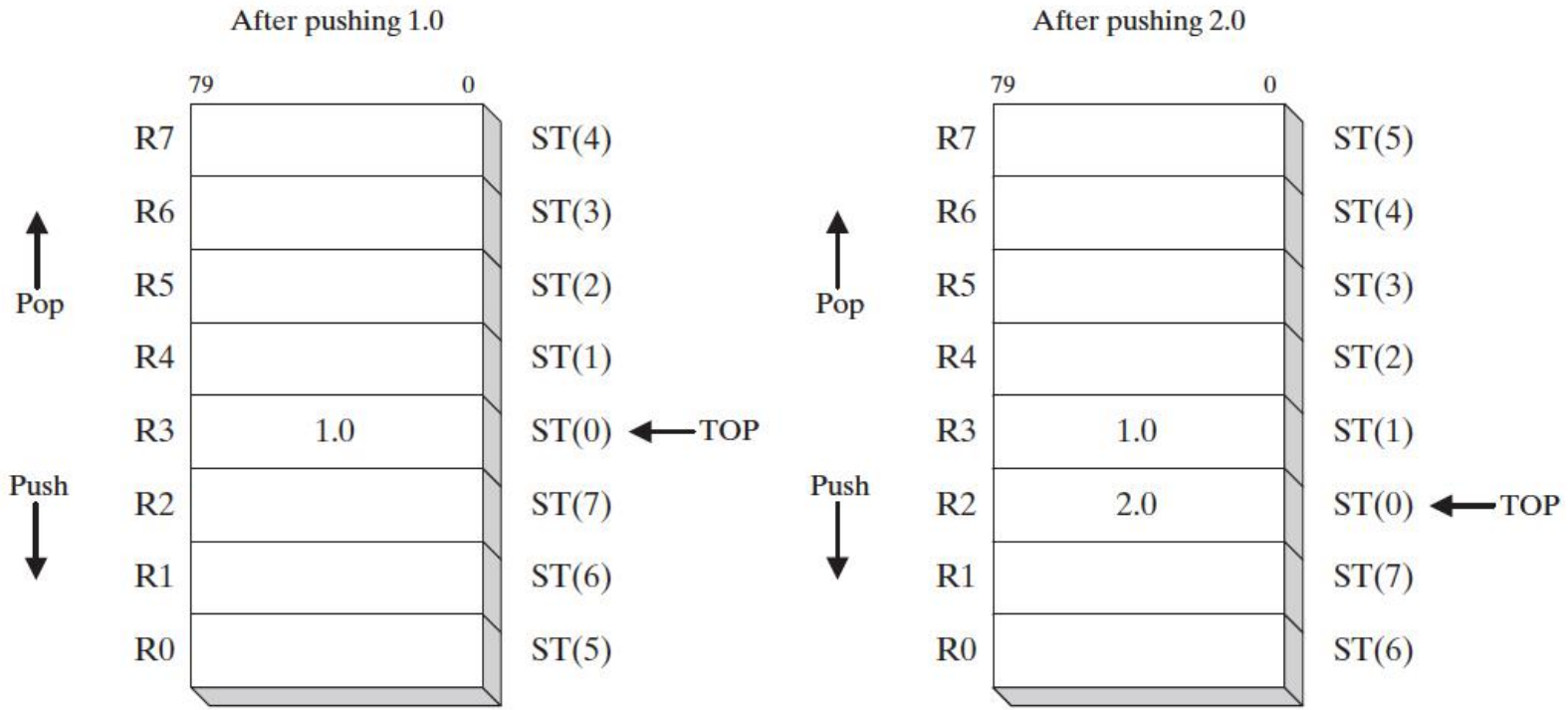


FPU Register Stack

- A push operation (also called load) decrements TOP by 1 and copies an operand into the register identified as ST(0)

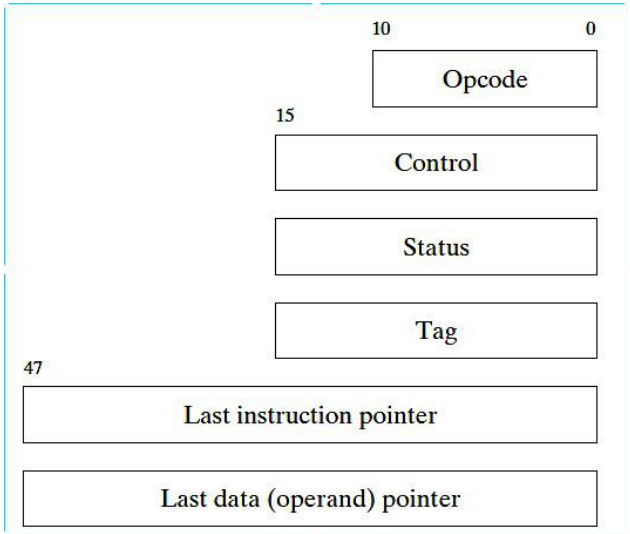
- Overwriting existing data in the register stack, a floating-point exception is generated

FIGURE 12–4 FPU stack after pushing 1.0 and 2.0.



Special Purpose Registers

- **Opcode register:** stores opcode of last non-control instruction executed
- **Control register:** controls precision and rounding method for calculations
- **Status register:** top-of-stack pointer, condition codes, exception warnings
- **Tag register:** indicates content type of each register in the register stack
- **Last instruction pointer register:** pointer to last non-control executed instruction
- **Last data (operand) pointer register:** points to data operand used by last executed instruction



Rounding

- FPU attempts to round an infinitely accurate result from a floating-point calculation
  - May be impossible because of storage limitations

- **Example**
  - suppose 3 fractional bits can be stored, and a calculated value equals +1.0111.
  - rounding up by adding .0001 produces 1.100
  - rounding down by subtracting .0001 produces 1.011

Rounding

The FPU lets you select one of four rounding methods:

- **Round to nearest even:** The rounded result is the closest to the infinitely precise result. If two values are equally close, the result is an even value (LSB = 0).

- **Round down toward -∞:** The rounded result is less than or equal to the infinitely precise result.

- **Round up toward +∞:** The rounded result is greater than or equal to the infinitely precise result.

- **Round toward zero:** (also known as truncation): The absolute value of the rounded result is less than or equal to the infinitely precise result.

FPU Control word

The FPU control word contains two bits named the RC field that specify which rounding method to use.

The field values are 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)

TABLE 12-9 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

FPU Exception

- **Six types of exception conditions**
  - Invalid operation (#I)
  - Divide by zero (#Z)
  - De-normalized operand (#D)
  - Numeric overflow (#O)
  - Numeric underflow (#U)
  - Inexact precision (#P)

For 'exceptions' details.
http://www.umiacs.umd.edu/~resnik/ling645_sp2002/cmu_manual/node19.html

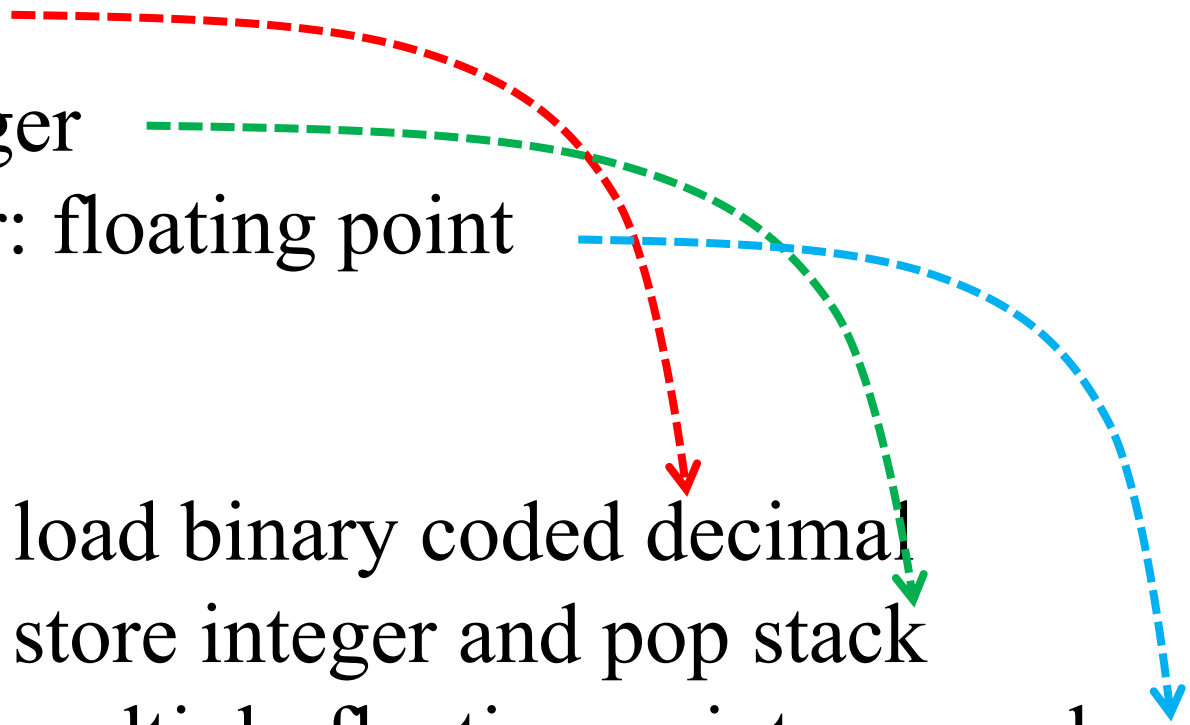
- **Each has a corresponding *mask* bit**
  - if set when an exception occurs, the exception is handled automatically by FPU
  - if clear when an exception occurs, a software exception handler is invoked

Floating-Point Instruction Set

Instruction mnemonics begin with letter F

Second letter identifies data type of memory operand;

- B = bcd
- I = integer
- no letter: floating point



Examples

- FLBD: load binary coded decimal
- FISTP: store integer and pop stack
- FMUL: multiply floating-point operands

Floating-Point Instruction Set

Operands

- zero, one, or two
- no immediate operands
- no general-purpose registers (EAX, EBX, ...)
- integers must be loaded from memory onto the stack and converted to floating-point before being used in calculations
- if an instruction has two operands, one must be a FPU register

Floating Point Data Types

Table 12-11 Intrinsic Data Types.

Type	Usage
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

Load Floating-Point Value

```
.data
bigVal REAL10 1.21234234234234243E+864
.code
fld bigVal ; load variable into stack
```

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

Table 12-11 Intrinsic Data Types.

Type	Usage
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

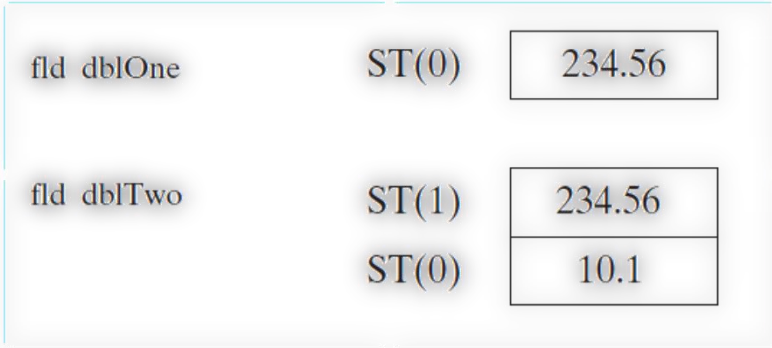
Load Floating-Point Value

```
.data
bigVal REAL10 1.21234234234234243E+864
.code
fld bigVal ; load variable into stack
```

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

Example The following example loads two direct operands on the FPU stack:

```
.data
dblOne REAL8 234.56
dblTwo REAL8 10.1
.code
fld dblOne ; ST(0) = dblOne
fld dblTwo ; ST(0) = dblTwo, ST(1) = dblOne
```



Store Floating-Point Value

FST

copies floating point operand from the top of the FPU stack into memory

FSTP

pops the stack after copying

- FST m32fp
- FST m80fp
- FST m64fp
- FST ST(i)

Arithmetic Instructions

Same operand types as FLD and FST

Table 12-12 Basic Floating-Point Arithmetic Instructions.

FCHS	Change sign
FADD	Add source to destination
FSUB	Subtract source from destination
FSUBR	Subtract destination from source
FMUL	Multiply source by destination
FDIV	Divide destination by source
FDIVR	Divide source by destination

Arithmetic Instructions: FCHS and FABS

The FCHS (change sign) instruction reverses the sign of the floating-point value in ST(0)

The FABS (absolute value) instruction clears the sign of the number in ST(0) to create its absolute value

Neither instruction has operands:

- FCHS
- FABS

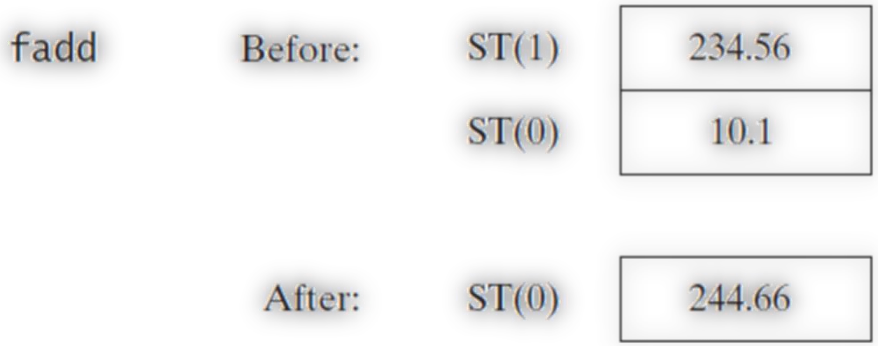
Arithmetic Instructions: FADD

The FADD (add) instruction has the following formats:

- FADD
- FADD m32fp
- FADD m64fp
- FADD ST(0), ST(i)
- FADD ST(i), ST(0)

If no operands are used with FADD, ST(0) is added to ST(1).

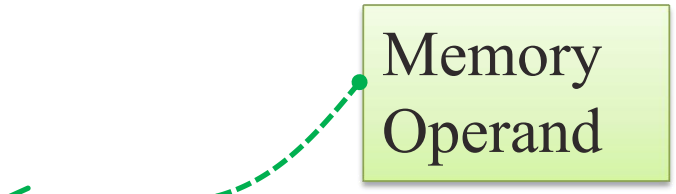
The result is temporarily stored in ST(1). ST(0) is then popped from the stack, leaving the result on the top of the stack.



Arithmetic Instructions: FADD

The FADD (add) instruction has the following formats:

- FADD
- FADD m32fp
- FADD m64fp
- FADD ST(0), ST(i)
- FADD ST(i), ST(0)



In memory operand, ST(0) will act as second operands. And memory location is added with ST(0).

Arithmetic Instructions: FADD

The FADD (add) instruction has the following formats:

- FADD
- FADD m32fp
- FADD m64fp
- FADD ST(0), ST(i)
- FADD ST(i), ST(0)

Register Operand

```
fadd st(1), st(0)
```

Before:	ST(1)	234.56
	ST(0)	10.1
After:	ST(1)	244.66
	ST(0)	10.1

Arithmetic Instructions: FSUB

The FSUB instruction subtracts a source operand from a destination operand, storing the difference in the destination operand

- FSUB
- FSUB m32fp
- FSUB m64fp
- FSUB ST(0), ST(i)
- FSUB ST(i), ST(0)

No Operand

Memory Operand

Register Operand

Arithmetic Instructions: FMUL

FMUL:

Multiplies source by destination, stores product in destination

```
fmul mySingle ; ST(0) *= mySingle
```

FDIV:

Divides destination by source, then pops the stack

The no-operand form of FDIV divides ST(1) by ST(0)

The no-operand versions of FMUL and FDIV pop the stack after multiplying or dividing.

Arithmetic Instructions: FMUL

FDIV:

Divides destination by source, then pops the stack

The no-operand form of FDIV divides ST(1) by ST(0)

ST(0) is popped from the stack, leaving the dividend on the top of the stack

```
.data
dblOne REAL8 1234.56
dblTwo REAL8 10.0
dblQuot REAL8 ?
.code
fld dblOne ; load into ST(0)
fdiv dblTwo ; divide ST(0) by dblTwo
fstp dblQuot ; store ST(0) to dblQuot
```

If the source operand is zero, a divide-by-zero exception is generated.

Comparing Floating Point Values

FCOM instruction

Operands:

Instruction	Description
FCOM	Compare ST(0) to ST(1)
FCOM *m32fp*	Compare ST(0) to *m32fp*
FCOM *m64fp*	Compare ST(0) to *m64fp*
FCOM ST(*i*)	Compare ST(0) to ST(*i*)

Comparing Floating Point Values

FCOM instruction

Operands:

Instruction	Description
FCOM	Compare ST(0) to ST(1)
FCOM *m32fp*	Compare ST(0) to *m32fp*
FCOM *m64fp*	Compare ST(0) to *m64fp*
FCOM ST(*i*)	Compare ST(0) to ST(*i*)

Condition Codes: Three FPU condition code flags, C3, C2, and C0, indicate the results of comparing floating-point values

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*)

Comparing Floating Point Values

FCOM instruction

Operands:

Instruction	Description
FCOM	Compare ST(0) to ST(1)
FCOM *m32fp*	Compare ST(0) to *m32fp*
FCOM *m64fp*	Compare ST(0) to *m64fp*
FCOM ST(*i*)	Compare ST(0) to ST(*i*)

The primary challenge after comparing two values and setting FPU condition codes is to find a way to branch to a label based on the conditions

Two steps are involved:

- Use the FNSTSW instruction to move the FPU status word into AX
- Use the SAHF instruction to copy AH into the EFLAGS register
- Use JA, JB, etc to do the branching

Comparing Floating Point Values

Calculate the absolute value of the difference between two floating-point values

```
.data
epsilon REAL8 1.0E-12                ; difference value
val2 REAL8 0.0                       ; value to compare
val3 REAL8 1.001E-13                 ; considered equal to val2

.code
; if( val2 == val3 ), display "Values are equal".
        fld epsilon                  ; load epsilon
        fld val2                     ; load val2
        fsub val3                    ; subtract
        fabs
        fcomi ST(0),ST(1)
        ja skip
        mWrite <"Values are equal",0dh,0ah>
skip:
```

Example

```
double X = 1.2;
double Y = 3.0;
int N = 0;
if( X < Y )
N = 1;
```

Write Assembly code for the above program?

Example

```
double X = 1.2;
double Y = 3.0;
int N = 0;
if( X < Y )
N = 1;
```

```
.data
X REAL8 1.2
Y REAL8 3.0
N DWORD 0
.code
; if( X < Y )
; N = 1
fld X           ; ST(0) = X
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:
```

Comparing Floating Point Values

The primary challenge after comparing two values and setting FPU condition codes is to find a way to branch to a label based on the conditions

Two steps are involved:

- Use the FNSTSW instruction to move the FPU status word into AX.
- Use the SAHF (store AH into flag) instruction to copy AH into the EFLAGS register.
- Use JA, JB, etc to do the branching.

Fortunately, the FCOMI instruction does steps 1 and 2 for you.

```
fcomi ST(0), ST(1)    ; Floating Point Compare, move results to EFLAGS
jnb   Label1
```

Floating-Point I/O

Irvine32 library procedures

ReadFloat

Reads FP value from keyboard, pushes it on the FPU stack

WriteFloat

Writes value from ST(0) to the console window in exponential format

ShowFPUStack

Displays contents of FPU stack

Exception Synchronization

Main CPU and FPU can execute instructions concurrently

- if an unmasked exception occurs, the current FPU instruction is interrupted and the FPU signals an exception
- But the main CPU does not check for pending FPU exceptions. It might use a memory value that the interrupted FPU instruction was supposed to set.
- Example:

```
.data
intVal DWORD 25
.code
fild intVal       ; load integer into ST(0)
inc    intVal     ; increment the integer
```

Exception Synchronization

For safety, insert a fwait instruction, which tells the CPU to wait for the FPU's exception handler to finish:

```
.data
intVal DWORD 25
.code
fild intVal       ; load integer into ST(0)
fwait             ; wait for pending exceptions
inc    intVal     ; increment the integer
```

Example

expression: valD = –valA + (valB * valC)

```
.data
valA REAL8 1.5
valB REAL8 2.5
valC REAL8 3.0
valD REAL8 ?              ; will be +6.0

.code
fld valA          ; ST(0) = valA
fchs                      ; change sign of ST(0)
fld  valB                 ; load valB into ST(0)
fmul valC                 ; ST(0) *= valC
fadd                      ; ST(0) += ST(1)
fstp valD                 ; store ST(0) to valD
```

Mixed Mode Arithmetic

Combining integers and reals. Integer arithmetic instructions such as ADD and MUL cannot handle reals FPU has instructions that promote integers to reals and load the values onto the floating point stack

Example: Z = N + X

```
.data
N SDWORD 20
X REAL8 3.5
Z REAL8 ?
.code
fild N                  ; load integer into ST(0)
fwait                   ; wait for exceptions
fadd X                  ; add mem to ST(0)
fstp Z                  ; store ST(0) to mem
```

Masking and Unmasking Exceptions

- Exceptions are masked by default
- Divide by zero just generates infinity, without halting the program
- If you unmask an exception processor executes an appropriate exception handler
- Unmask the divide by zero exception by clearing bit 2:

```
.data
ctrlWord WORD ?
.code
fstcw ctrlWord                          ; get the control word
and ctrlWord,1111111111111011b ; unmask divide by zero
fldcw ctrlWord                          ; load it back into FPU
```

Questions?

THANK YOU!