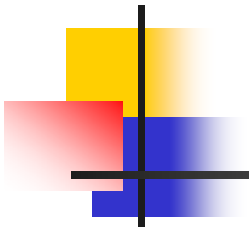
Computer Organization & Assembly Languages

Floating-Point Processing

Pu-Jen Cheng

Adapted from the slides prepared by Kip Irvine for the book, Assembly Language for Intel-Based Computers, 5th Ed.
And prepared by Behrooz Parhami for the book, Computer Architecture, 2005



Outline

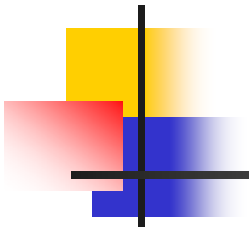
- **Floating-Point Binary Representation**
- Floating-Point Unit

Decimal Fractions vs Binary Floating-Point

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
0.000000000000000000000001	1/8388608

TABLE 17-3 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

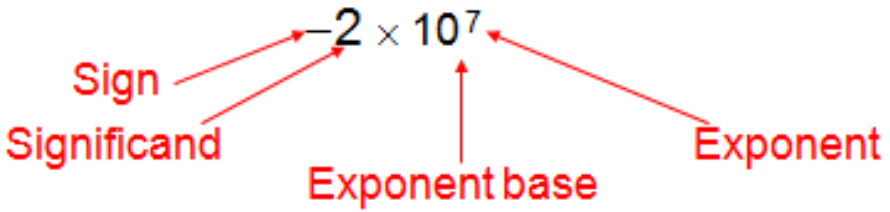


Floating-Point Binary Representation

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

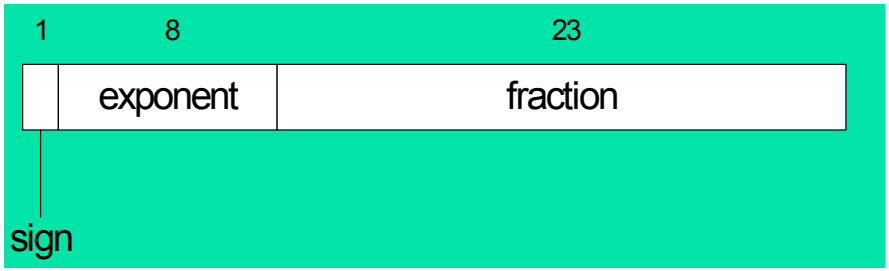
IEEE Floating-Point Binary Reals

- 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 Format (short 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)

    approximate normalized range: 2^{-126} to 2^{127}

Single-Precision Format (cont.)

- Biased Exponent = Actual Exponent + 127

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

- Normalized: a single 1 appears to the left of the binary point
- Unnormalized: exponent is zero

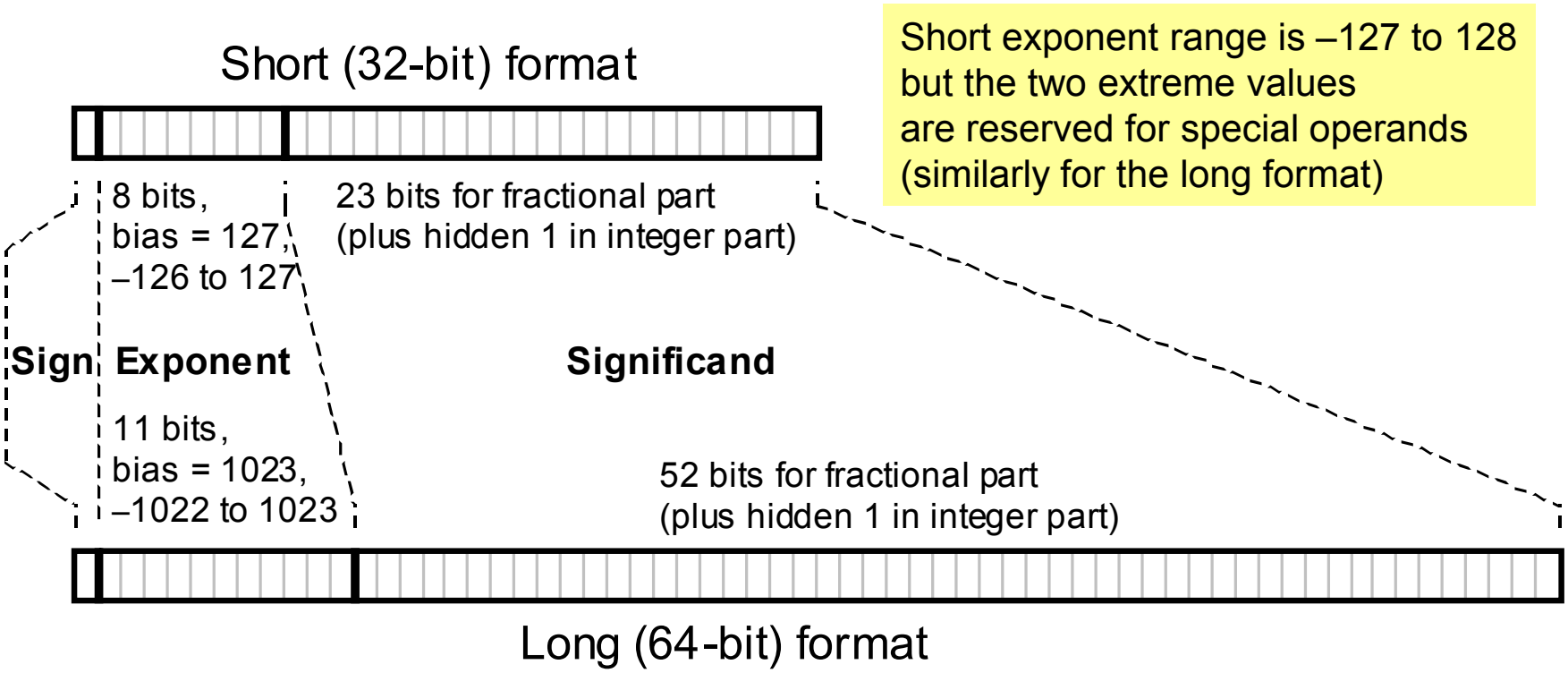
Unnormalized	Normalized
1110.1	1.1101×2^{3}
.000101	1.01×2^{-4}
1010001.	1.010001×2^{6}

Examples (Single Precision)

- Order: sign bit, exponent bits, and fractional part

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

ANSI/IEEE Standard Floating-Point Format



Short exponent range is –127 to 128
but the two extreme values
are reserved for special operands
(similarly for the long format)

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, signaling

Specific encodings (single precision)

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]

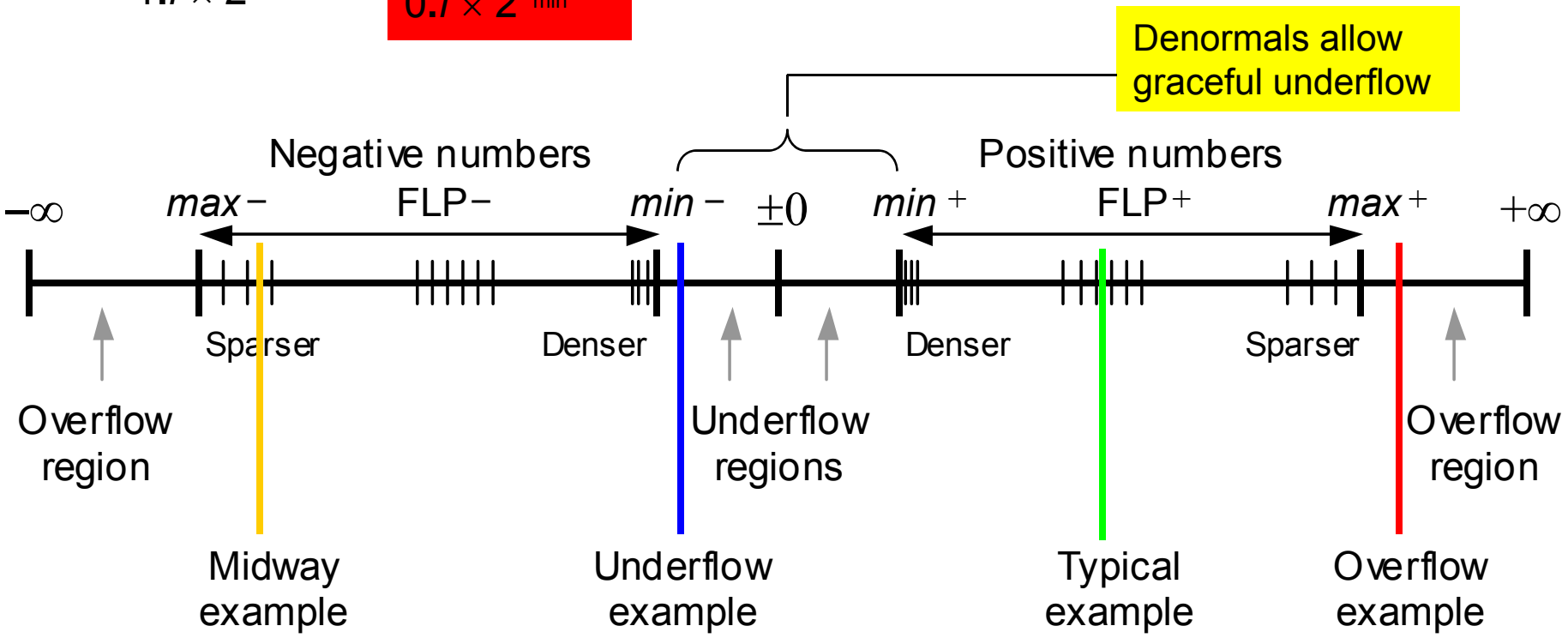
Distribution of Floating-point Numbers on the Real Line

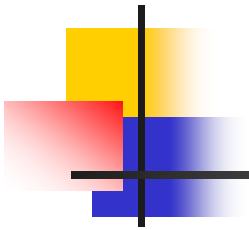
± 0, $\pm \infty$, NaN

$1.f \times 2^e$

Denormals:

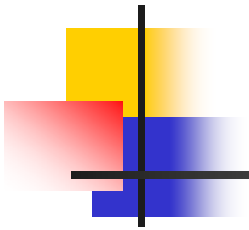
$0.f \times 2^{e_{min}}$





Converting Single-Precision to Decimal

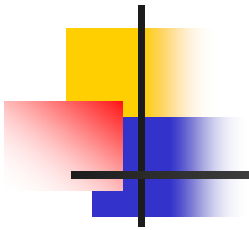
1. If the MSB is 1, the number is negative; otherwise, it is positive.
2. The next 8 bits represent the exponent. Subtract binary 01111111 (decimal 127), producing the unbiased exponent. Convert the unbiased exponent to decimal.
3. The next 23 bits represent the significand. Notate a "1.", followed by 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. Unnormalize the binary number produced in step 3. (Shift the binary point 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

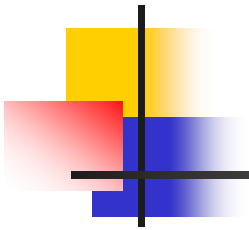
Convert 0 10000010 10110000000000000000 to Decimal

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 $+1.01011 \times 2^3$.
4. The unnormalized binary number is +1010.11.
5. The decimal value is +10 3/4, or +10.75.



Outline

- Floating-Point Binary Representation
- **Floating-Point Unit**

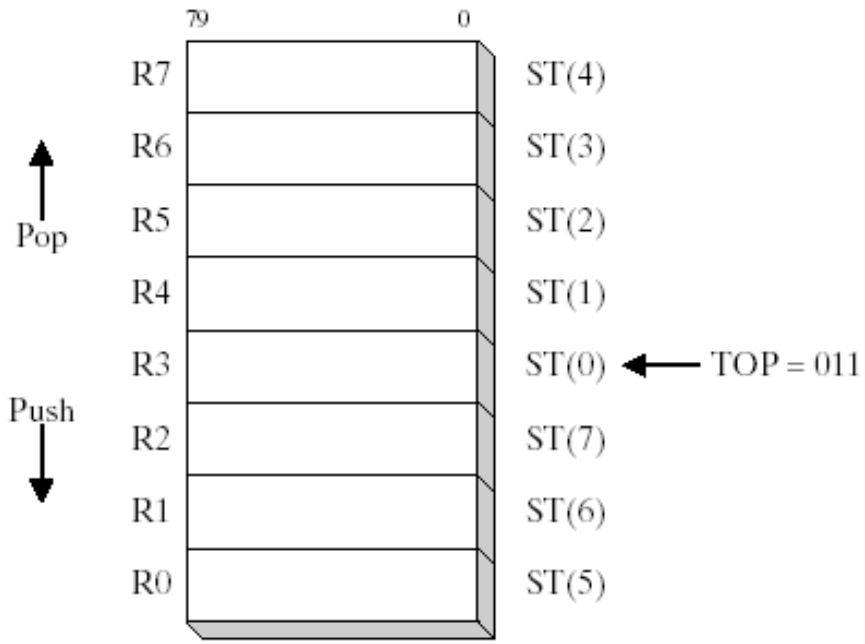


Floating Point Unit

- 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

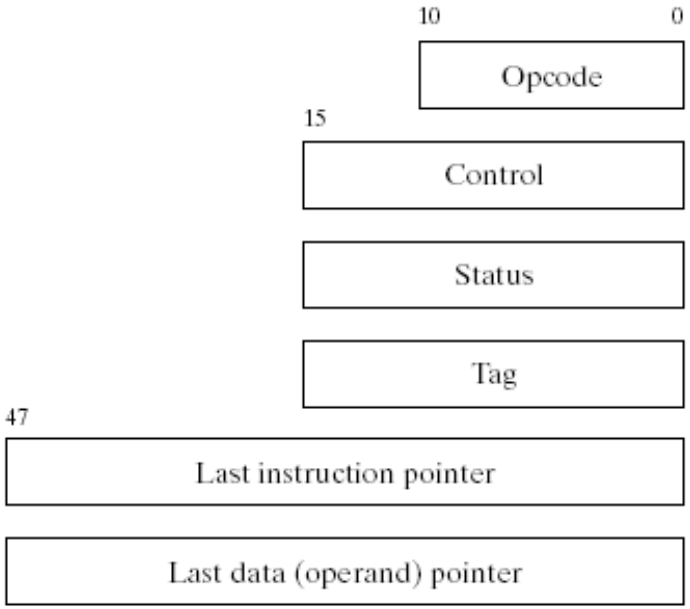
FPU Register Stack

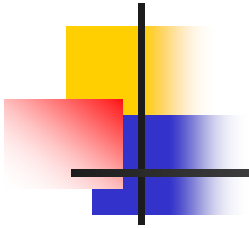
- 8 individually addressable 80-bit data registers
- Register stack: R0~R7
- Three-bit field named TOP in the FPU status word identifies the register number that is currently the top of stack.



Special-Purpose Registers

- Opcode register: stores opcode of last noncontrol 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

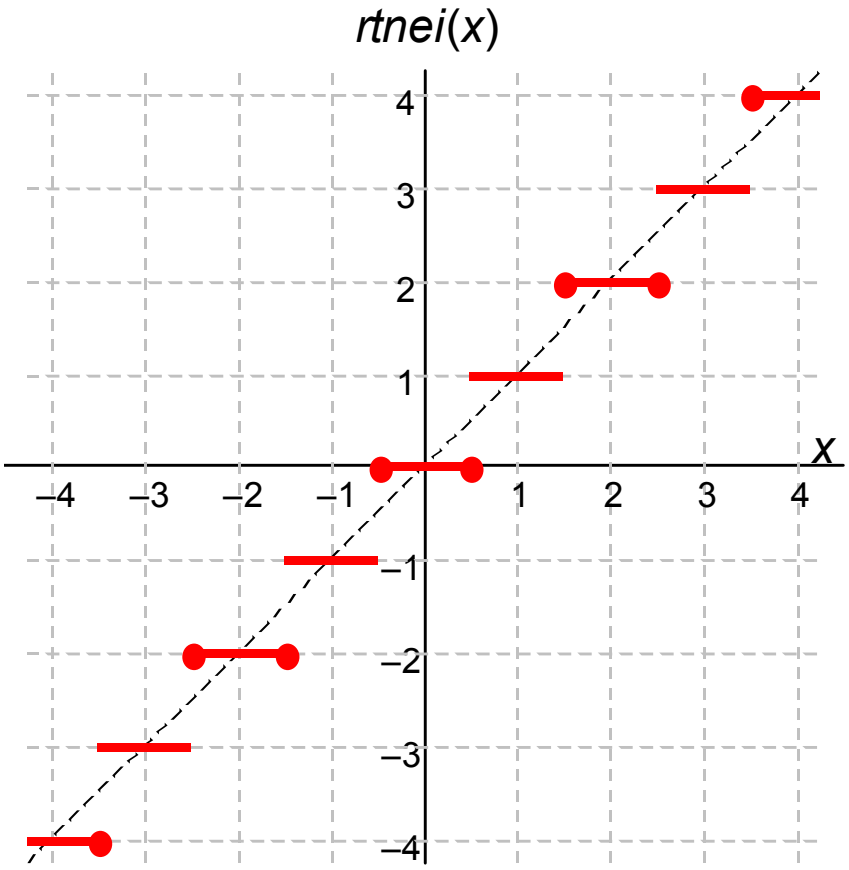




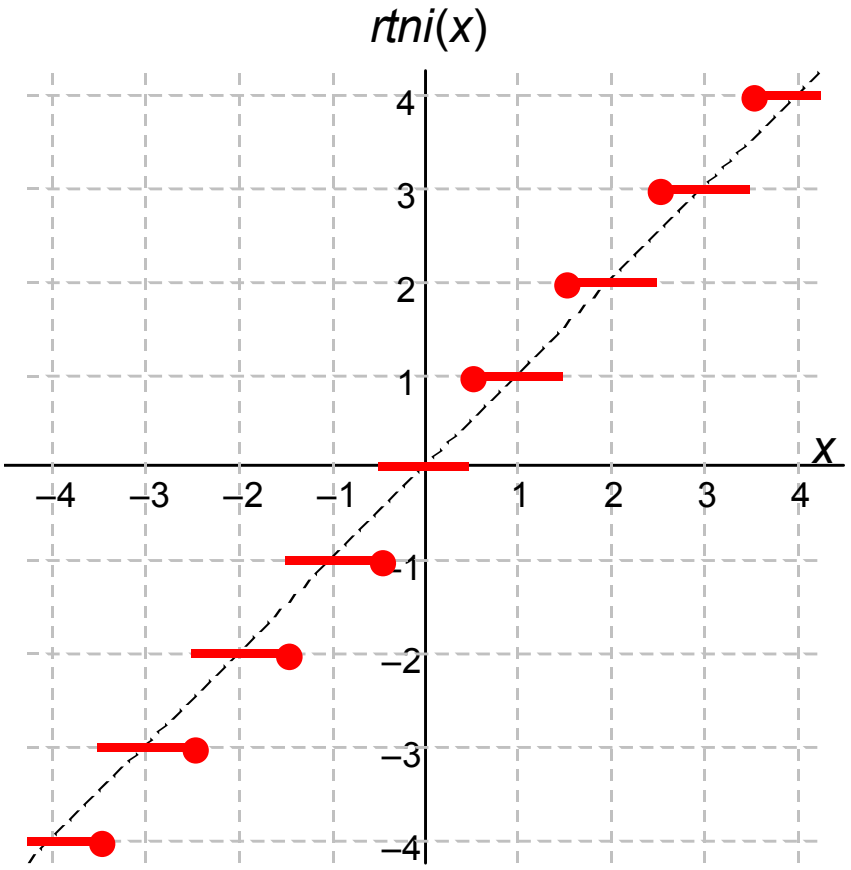
Rounding

- FPU attempts to round an infinitely accurate result from a floating-point calculation
  - may be impossible because of storage limitations
- Example
  - Store +1.0111 in 3 fractional bits
  - rounding up by adding .0001 produces 1.100
  - rounding down by subtracting .0001 produces 1.011
- Round to nearest even
- Round down toward $-\infty$
- Round up toward ∞
- Round toward zero

Round-to-Nearest (Even)

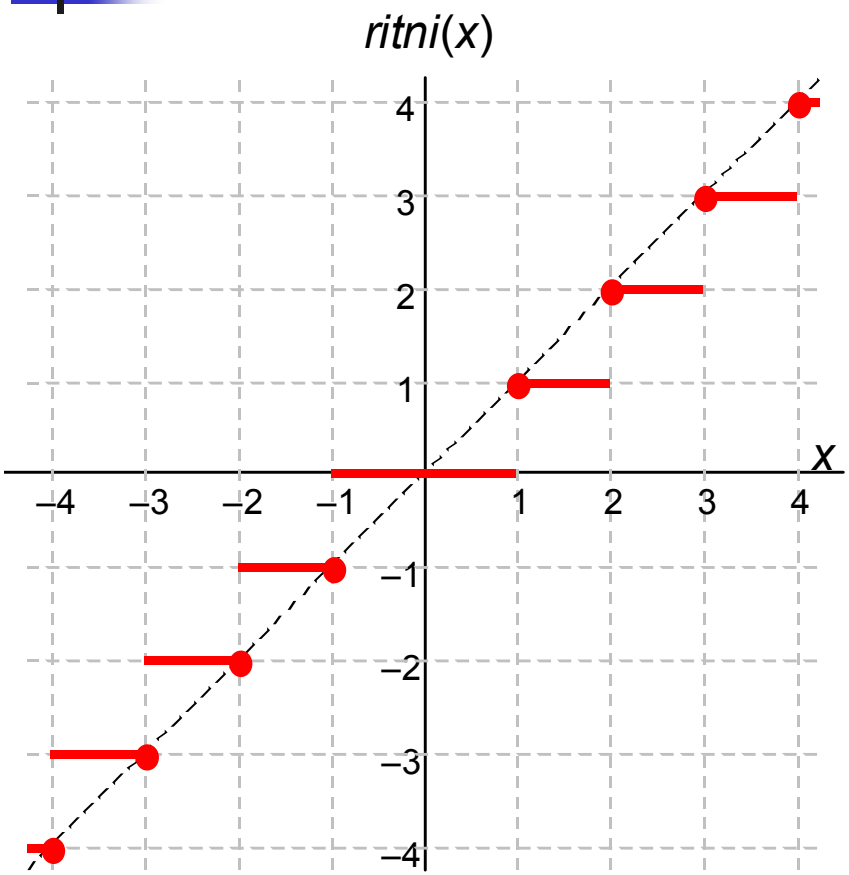


(a) Round to nearest even integer

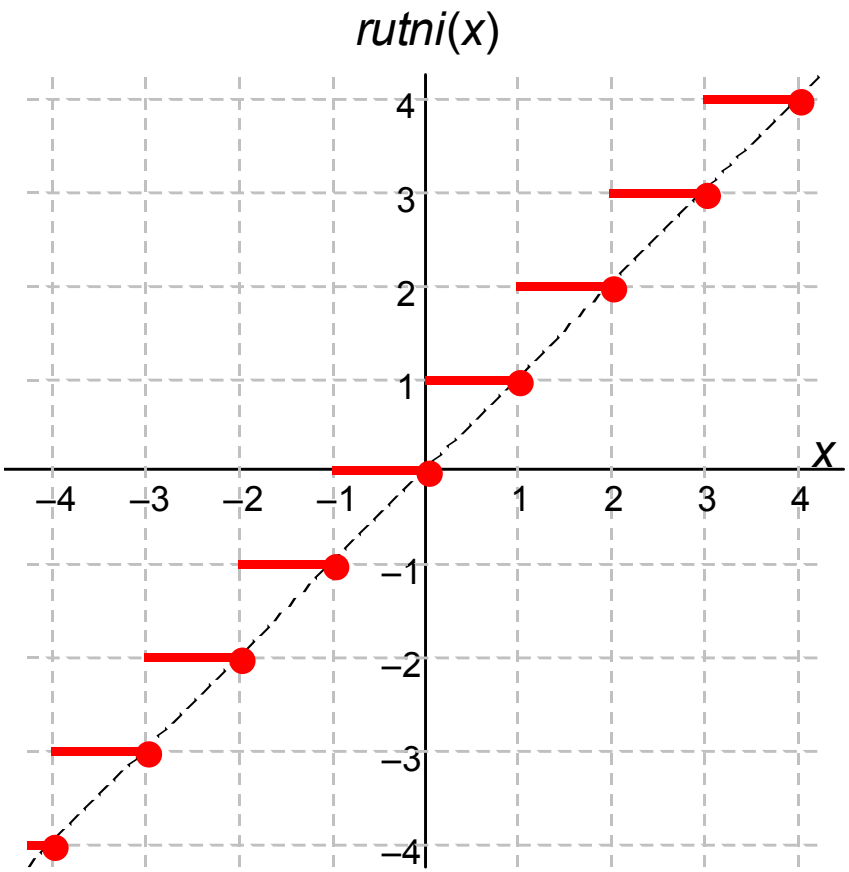


(b) Round to nearest integer

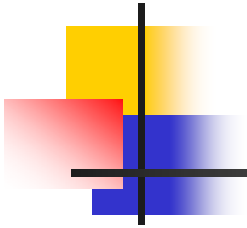
Directed Rounding



(a) Round inward to nearest integer

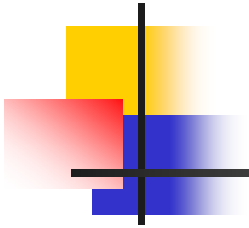


(b) Round upward to nearest integer



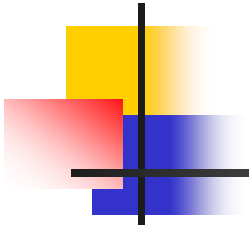
Floating-Point Exceptions

- Six types of exception conditions
  - Invalid operation
  - Divide by zero
  - Denormalized operand
  - Numeric overflow (or underflow)
  - Inexact precision
- 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



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



FPU 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

FP Instruction Set

- Data Types

Table 17-11 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

- FLD
  - Copies floating point operand from memory into the top of the FPU stack, ST(0)

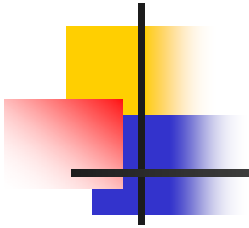
```
FLD m32fp
FLD m64fp
FLD m80fp
FLD ST(i)
```

- Example

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

- Loading constants: FLD1 (1), FLDZ (0), FLDL2T ($\log_2 10$), FLDPI, FLDLG2 ($\log_{10} 2$), FLDLN2 ($\log_e 2$)

Store Floating-Point Value



- FST
  - Copies floating point operand from the top of the FPU stack into memory

```
FST m32fp
FST m64fp
FST ST(i)
```

- FSTP
  - pops the stack after copying

Arithmetic Instructions

- Same operand types as FLD and FST

Table 17-12 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

Floating-Point Add

- FADD
  - Adds source to destination
  - No-operand version pops the FPU sta
    adding (ST(1) = ST(0)+ST(1); pop ST(0))

FADD[4]
FADD *m32fp*
FADD *m64fp*

- Examples:

```
fadd
```

Before:

ST(1)	234.56
ST(0)	10.1

fadd=faddp st(1), st(0)

After:

ST(0)	244.66

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

Before:

ST(1)	234.56
ST(0)	10.1

After:

ST(1)	244.66
ST(0)	10.1

Floating-Point Subtract

- FSUB
  - subtracts source from destination.
  - No-operand version pops the FPU stack after subtracting

```
FSUB[5]
FSUB m32fp
FSUB m64fp
FSUB ST(0), ST(i)
FSUB ST(i), ST(0)
```

- Example:

```
fsub mySingle                ; ST(0) -= mySingle
fsub array[edi*8]            ; ST(0) -= array[edi*8]
```

Floating-Point Multiply

- FMUL
  - Multiplies source by destination, stores product in destination

```
FMUL[6]
FMUL m32fp
FMUL m64fp
FMUL ST(0), ST(i)
FMUL ST(i), ST(0)
```

- FDIV
  - Divides destination by source, then pops the stack

```
FDIV[7]
FDIV m32fp
FDIV m64fp
FDIV ST(0), ST(i)
FDIV ST(i), ST(0)
```

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

Comparing FP 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 set by FPU
  - Codes similar to CPU flags

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*)

[a] If an invalid arithmetic operand exception is raised (because of invalid operands) and the exception is masked, C3, C2, and C0 are set according to the row marked *Unordered*.

Branching after FCOM

- Required steps:
  1. Use the FNSTSW instruction to move the FPU status word into AX.
  2. Use the SAHF instruction to copy AH into the EFLAGS register.
  3. Use JA, JB, etc to do the branching.

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

```
fcomi ST(0), ST(1)
jnb   Label1
```

Comparing for Equality

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

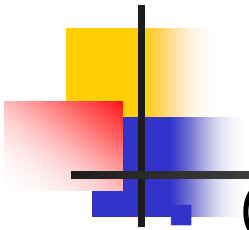
```
.data
epsilon REAL8 1.0E-12          ; difference value
val2 REAL8 0.0                 ; value to compare
val3 REAL8 1.001E-13           ; considered equal to val2

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

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:

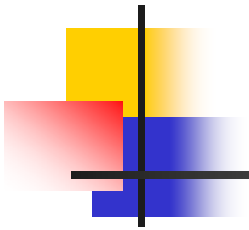
```
.data
intVal DWORD 25
.code
fild intVal          ; load integer into ST(0)
inc  intVal          ; increment the integer
```



Exception Synchronization

- (continued)
- 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
```



FPU Code 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:

```
.data
N SDWORD 20
X REAL8 3.5
Z REAL8 ?
Y SDWORD ?
.code
fild N                  ; load integer into ST(0)
fadd X                  ; add mem to ST(0)
fist Y                   ; store ST(0) to mem int Y = N + X
fstp Z                  ; store ST(0) to mem real8 Z = N + X
```

Changing the Rounding Mode

- The **RC** field (bits 11 and 10) of the FPU Control Word determines how the FPU will round results:
  - 00 = Round to nearest, or to even if equidistant (initialized state)
    01 = Round down (toward -infinity)
    10 = Round up (toward +infinity)
    11 = Truncate (toward 0)

```
.data
N SDWORD 20
X REAL8 3.5
Z REAL8 ?
ctrlWord WORD ?
.code
fstcw ctrlWord                    ; store control word
or    ctrlWord, 110000000000b     ; set RC = truncate
fldcw ctrlWord, ctrlWord          ; load control word
fild N                  ; load integer into ST(0)
fadd X                  ; add mem to ST(0)
fist Y                   ; store ST(0) to mem int Y = 23
```

Masking and Unmasking Exceptions

- Exceptions are masked by default
  - Divide by zero just generates infinity, without halting the program

```
.data
ctrlWord  WORD ?
val1      DWORD 1
val2      REAL8 0.0
.code
fild    val1                  ; load int into ST(0)
fdiv    val2                  ; ST(0) = positive infinity
fstcw ctrlWord                      ; get the control word
and ctrlWord,1111111111111011b      ; unmask divide by zero
fldcw ctrlWord                      ; load it back into FPU
fild    val1                  ; load int into ST(0)
fdiv    val2                  ; MS Windows will show exception
```

Floating-Point Addition

$$(\pm 2^{e1} s1) + (\pm 2^{e1}(s2 / 2^{e1-e2})) = \pm 2^{e1}(s1 \pm s2 / 2^{e1-e2})$$

$(\pm 2^{e2} s2)$

Numbers to be added:

$x = 2^5 \times 1.00101101$

$y = 2^1 \times 1.11101101$ ← Operand with smaller exponent to be preshifted

Operands after alignment shift:

$x = 2^5 \times 1.00101101$

$y = 2^5 \times 0.000111101101$

Result of addition:

Extra bits to be rounded off

$s = 2^5 \times 1.010010111101$

$s = 2^5 \times 1.01001100$ ← Rounded sum

Hardware for Floating-Point Addition