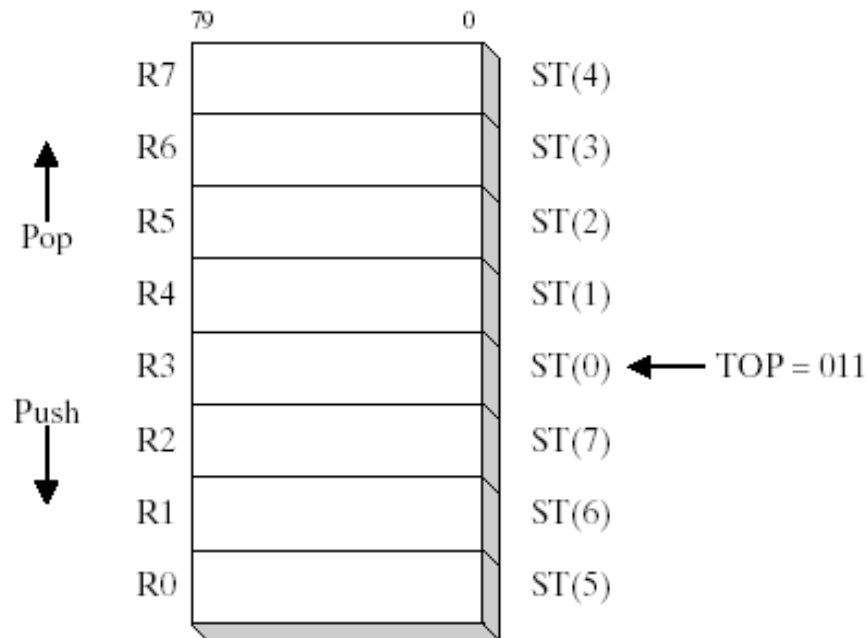


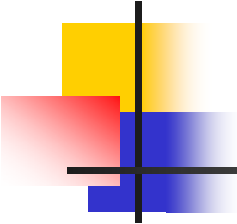
Floating Point Unit

- 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

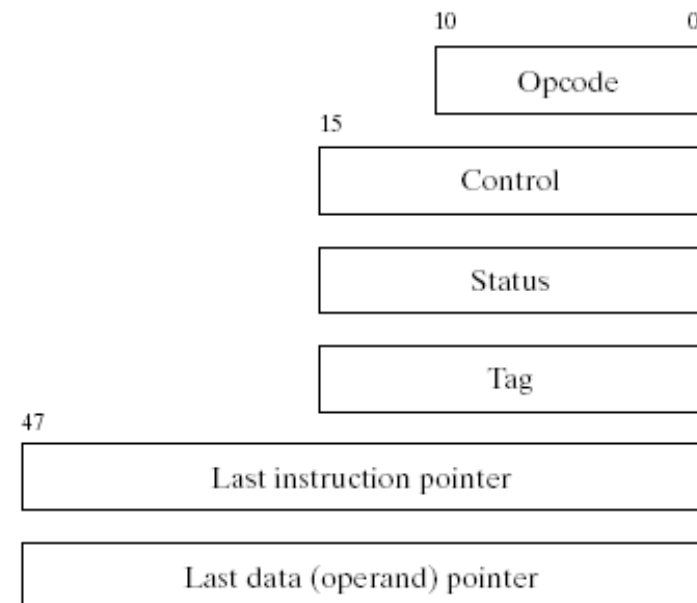
- 8 individually addressable 80-bit data registers
- Register stack: R0~R7
- Three-bit field named TOP in the FPU status word identifies the register number that is currently the top of stack.

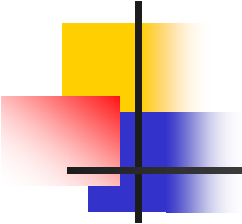




Special-Purpose Registers

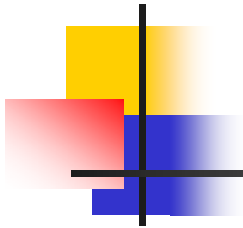
- Opcode register: stores opcode of last noncontrol instruction executed
- Control register: controls precision and rounding method for calculations
- Status register: top-of-stack pointer, condition codes, exception warnings
- Tag register: indicates content type of each register in the register stack
- 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



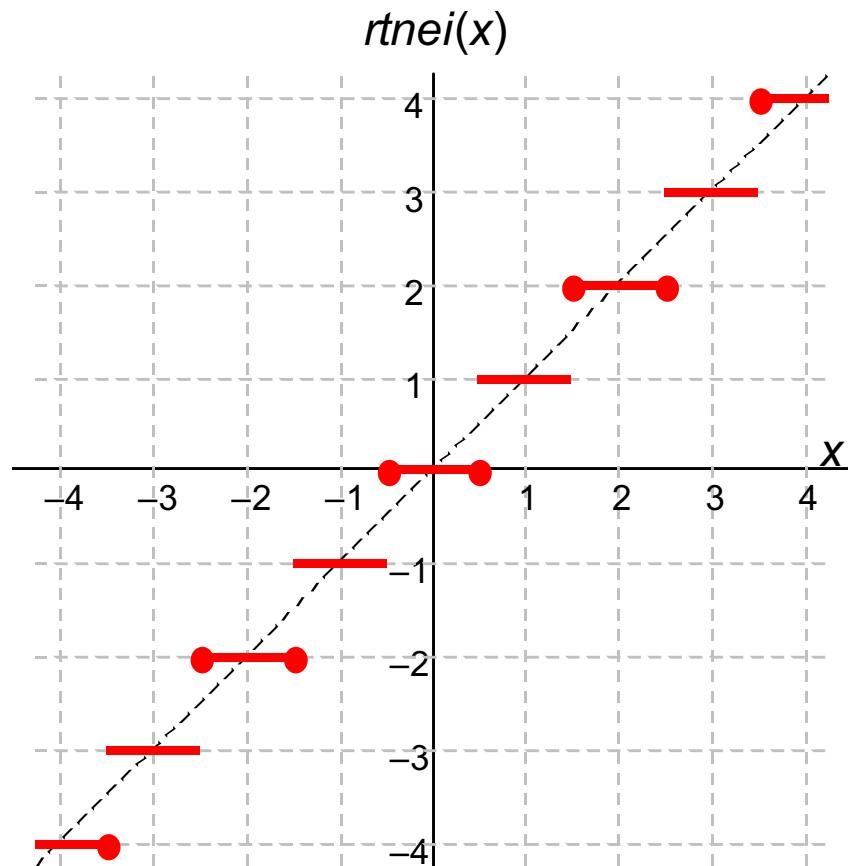


Rounding

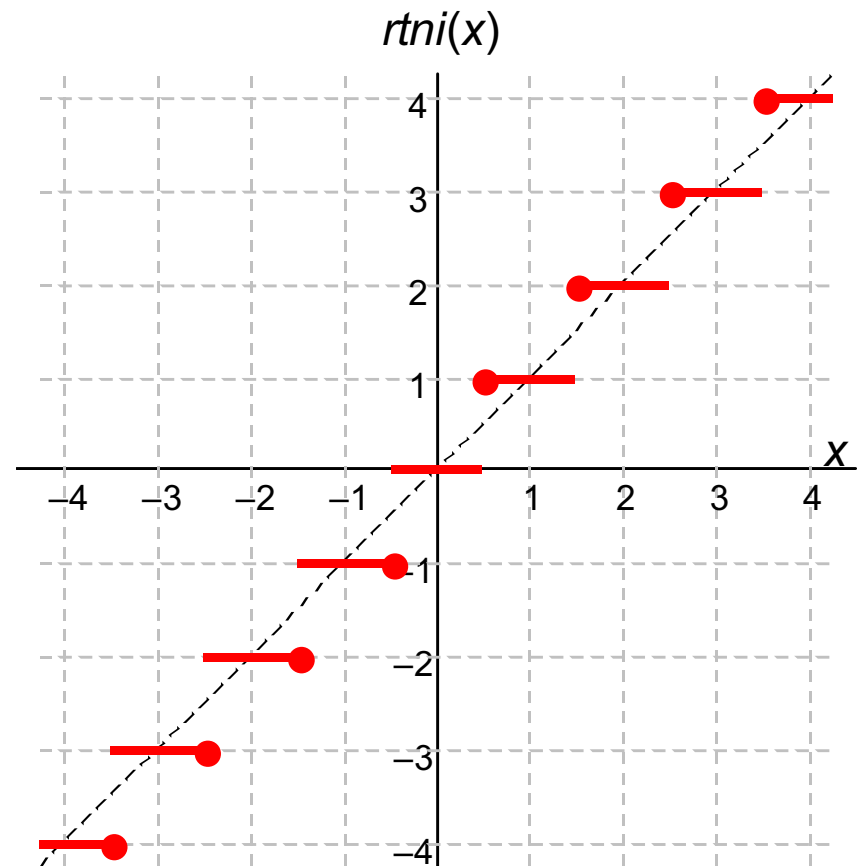
- FPU attempts to round an infinitely accurate result from a floating-point calculation
 - may be impossible because of storage limitations
- Example
 - Store +1.0111 in 3 fractional bits
 - rounding up by adding .0001 produces 1.100
 - rounding down by subtracting .0001 produces 1.011
- Round to nearest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



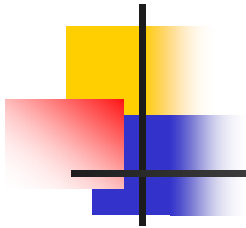
Round-to-Nearest (Even)



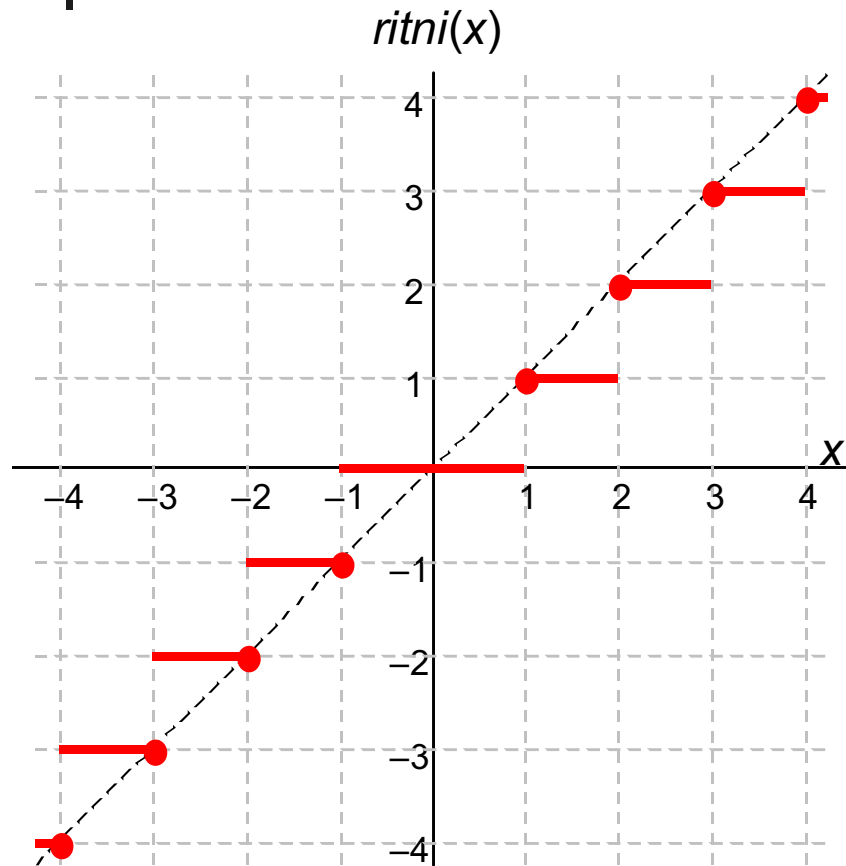
(a) Round to nearest even integer



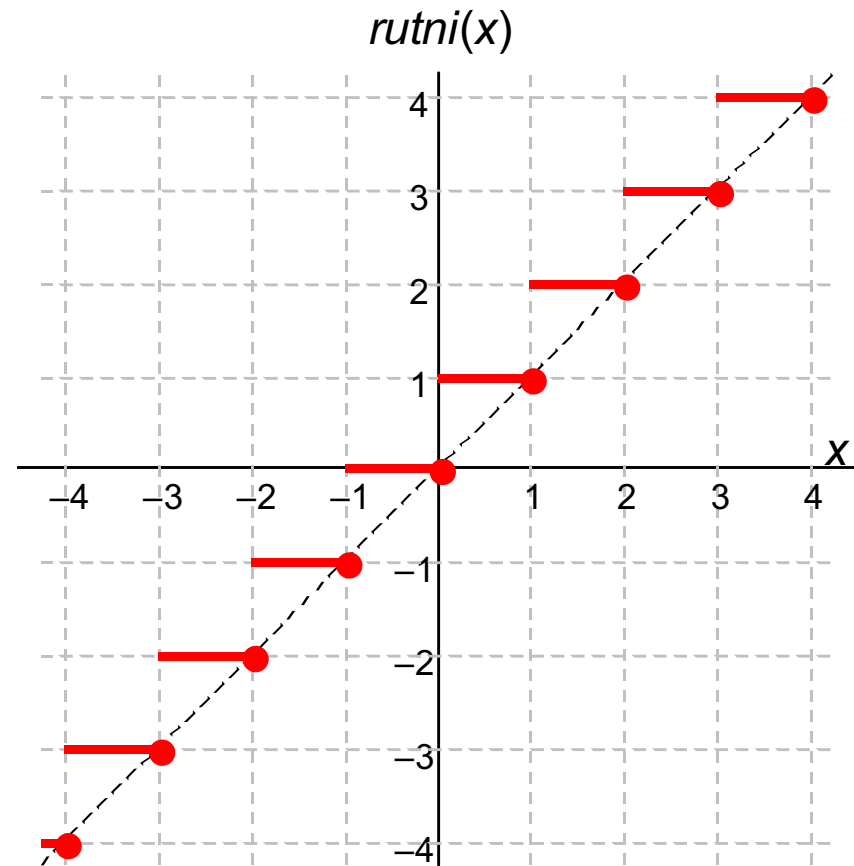
(b) Round to nearest integer



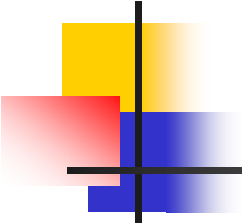
Directed Rounding



(a) Round inward to nearest integer

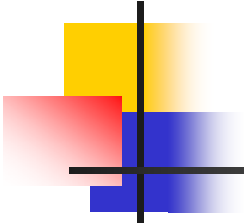


(b) Round upward to nearest integer



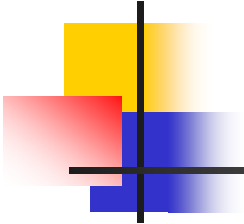
Floating-Point Exceptions

- 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

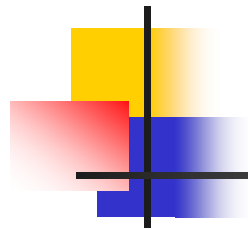
- Instruction mnemonics begin with letter F
- Second letter identifies data type of memory operand
 - B = bcd
 - I = integer
 - no letter: floating point
- Examples
 - FBLD load binary coded decimal
 - FISTP store integer and pop stack
 - FMUL multiply floating-point operands



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

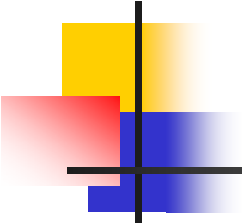


FP Instruction Set

- 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

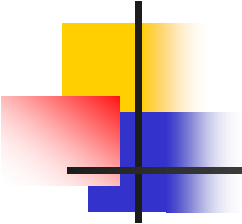
```
.data
dblOne    REAL8 234.56
dblTwo    REAL8 10.1
```

```
.code
fld  dblOne
fld  dblTwo
```

; ST(0) = dblOne

; 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

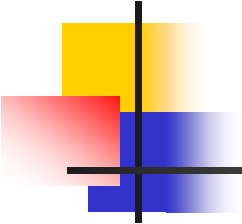
FST m32fp

FST m64fp

FST ST(i)

- FSTP

- pops the stack after copying

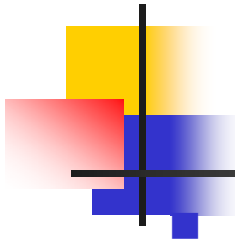


Arithmetic Instructions

- 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 stack adding ($ST(1) = ST(0) + ST(1)$; pop $ST(0)$)

FADD⁴

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

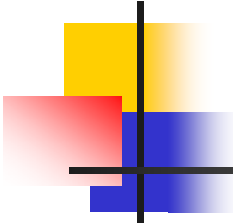
After:

ST(1)

244.66

ST(0)

10.1



Floating-Point Subtract

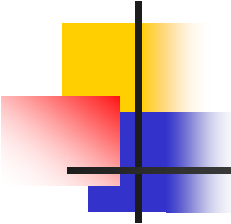
■ FSUB

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

FSUB⁵
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

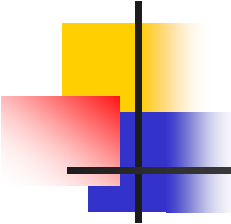
FMUL⁶
FMUL *m32fp*
FMUL *m64fp*
FMUL ST(0), ST(*i*)
FMUL ST(*i*), ST(0)

■ FDIV

- Divides destination by source, then pops the stack

FDIV⁷
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

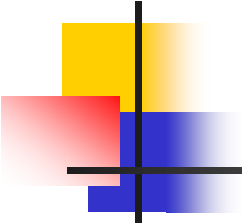
- FCOM instruction
- Operands:

Instruction	Description
FCOM	Compare ST(0) to ST(1)
FCOM <i>m32fp</i>	Compare ST(0) to <i>m32fp</i>
FCOM <i>m64fp</i>	Compare ST(0) to <i>m64fp</i>
FCOM ST(<i>i</i>)	Compare ST(0) to ST(<i>i</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)

^aIf 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*.



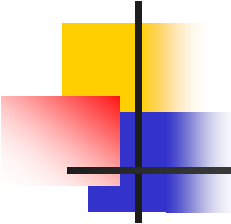
Branching after FCOM

- 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
```

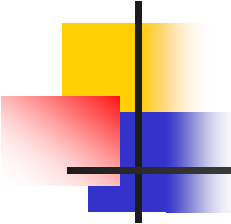


Comparing for Equality

- 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

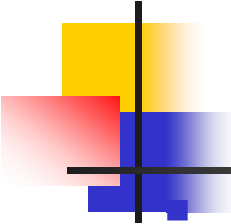
.code
; if( val2 == val3 ), display "Values are equal".
    fld epsilon
    fld val2
    fsub val3
    fabs
    fcomi ST(0),ST(1)
    ja skip
    mWrite <"Values are equal",0dh,0ah>
skip:
```



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
fld intVal          ; load integer into ST(0)
inc intVal           ; increment the integer
```

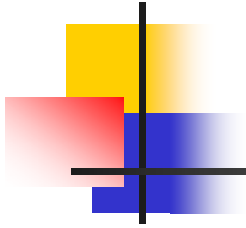


Exception Synchronization

(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

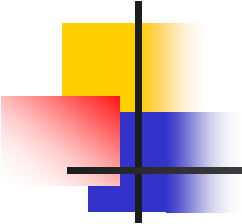
fchs ; change sign of ST(0)

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 ?
```

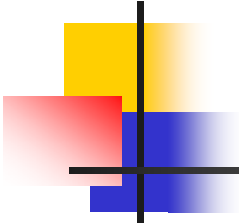
```
.code
```

```
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
```

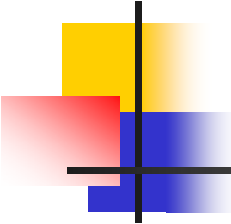
```
fstp Z ; store ST(0) to mem real8 Z = N + X
```



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
```

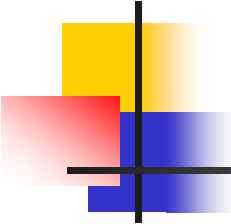


Masking and Unmasking Exceptions

- 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

$$(\pm 2^{e_1} s_1) + (\pm 2^{e_1} (s_2 / 2^{e_1 - e_2})) = \pm 2^{e_1} (s_1 \pm s_2 / 2^{e_1 - e_2})$$

$(\pm 2^{e_2} s_2)$ ————— ↑

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:

$$s = 2^5 \times 1.0100101111101$$

$$s = 2^5 \times 1.01001100$$

Extra bits to be
rounded off

Rounded sum

Hardware for Floating-Point Addition

