

## Real Arithmetic

*Computer Organization and Assembly Languages*  
Yung-Yu Chuang  
2006/12/11

## Announcement

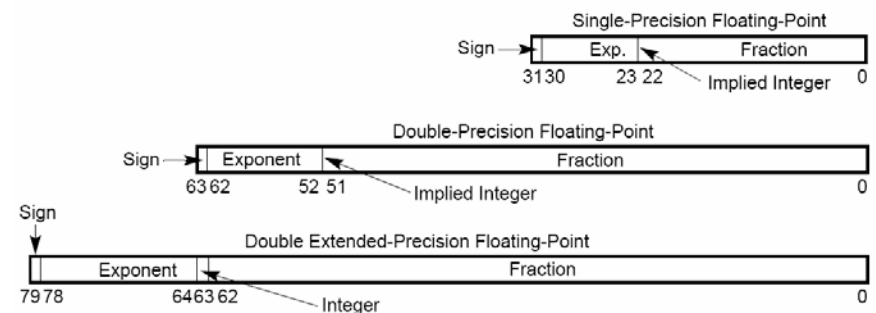
- Homework #4 is extended for two days
- Homework #5 will be announced today, due two weeks later.
- Midterm re-grading by this Thursday.

## IA-32 floating point architecture

- Original 8086 only has integers. It is possible to simulate real arithmetic using software, but it is slow.
- 8087 floating-point processor (and 80287, 80387) was sold separately at early time.
- Since 80486, FPU (floating-point unit) was integrated into CPU.

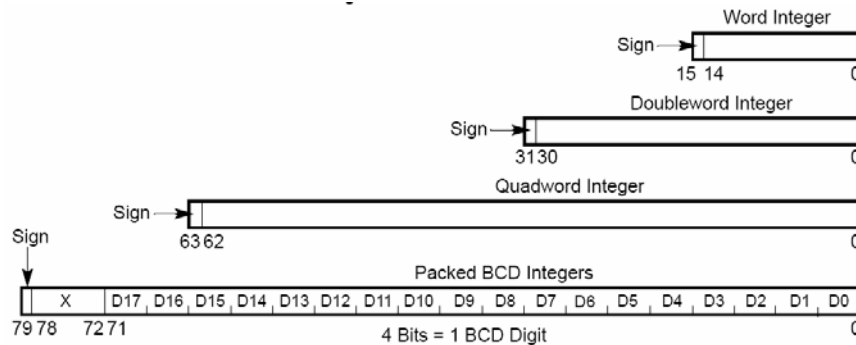
## FPU data types

- Three floating-point types



## FPU data types

- Four integer types

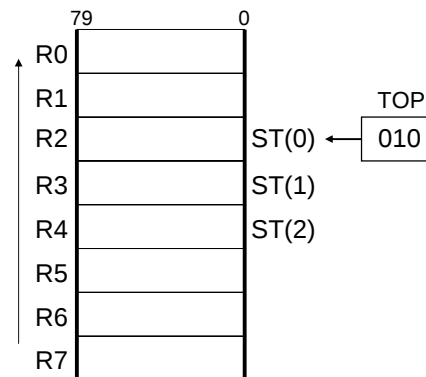


## FPU registers

- Data register
- Control register
- Status register
- Tag register

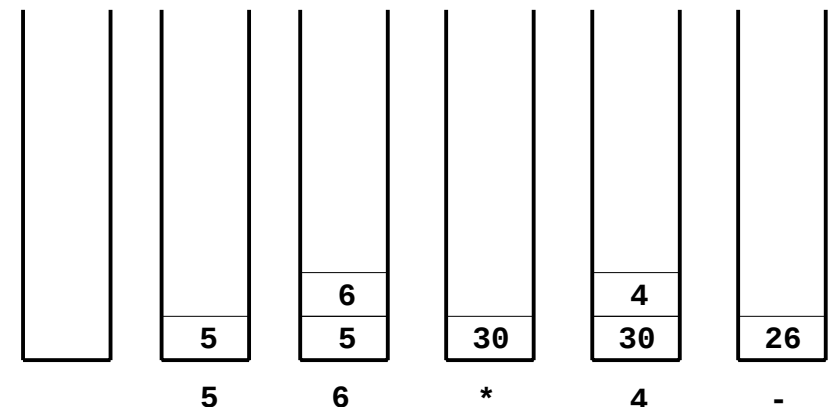
## Data registers

- Load: push, TOP--
- Store: pop, TOP++
- Instructions access the stack using **ST(i)** relative to TOP
- If TOP=0 and push, TOP wraps to R7
- If TOP=7 and pop, TOP wraps to R0
- When overwriting occurs, generate an exception
- Real values are transferred to and from memory and stored in 10-byte temporary format. When storing, convert back to integer, long, real, long real.

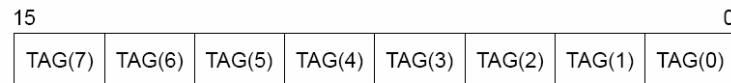
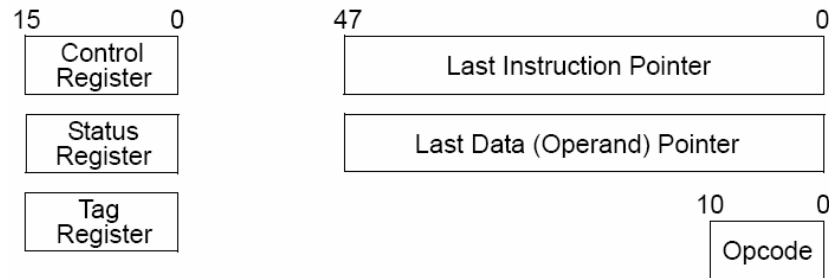


## Postfix expression

- $(5 * 6) - 4 \rightarrow 5 \ 6 \ * \ 4 \ -$



## Special-purpose registers

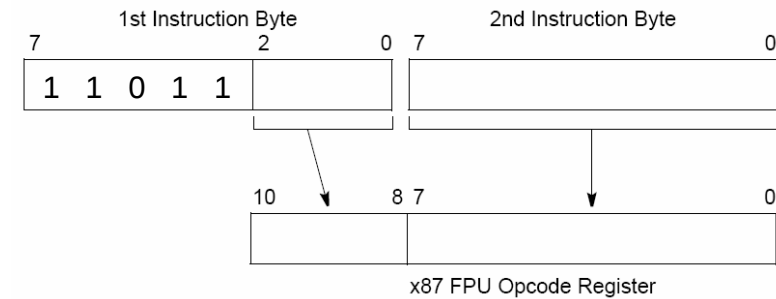


### TAG Values

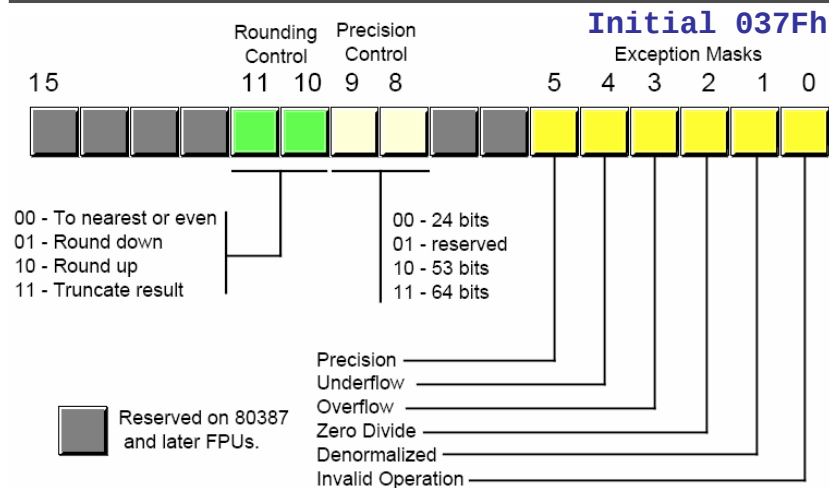
- 00 — Valid
- 01 — Zero
- 10 — Special: invalid (NaN, unsupported), infinity, or denormal
- 11 — Empty

## Special-purpose registers

- Last data pointer stores the memory address of the operand for the last non-control instruction. Last instruction pointer stored the address of the last non-control instruction. Both are 48 bits, 32 for offset, 16 for segment selector.



## Control register



The instruction **FINIT** will initialize it to 037Fh.

## Rounding

- FPU attempts to **round** an infinitely accurate result from a floating-point calculation
  - Round to nearest even: round toward to the closest one; if both are equally close, round to the even one
  - Round down: round toward to  $-\infty$
  - Round up: round toward to  $+\infty$
  - Truncate: round toward to zero
- 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



| method                | original value | rounded value |
|-----------------------|----------------|---------------|
| Round to nearest even | <b>1.0111</b>  | <b>1.100</b>  |
| Round down            | <b>1.0111</b>  | <b>1.011</b>  |
| Round up              | <b>1.0111</b>  | <b>1.100</b>  |
| Truncate              | <b>1.0111</b>  | <b>1.011</b>  |

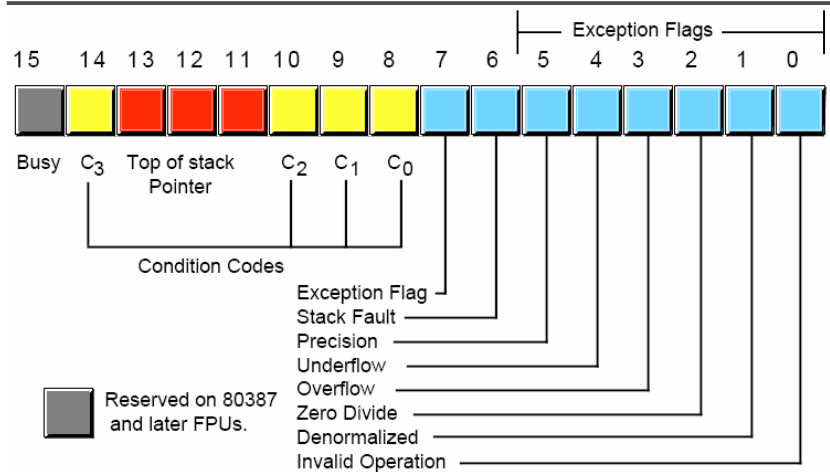
| method                | original value | rounded value |
|-----------------------|----------------|---------------|
| Round to nearest even | <b>-1.0111</b> | <b>-1.100</b> |
| Round down            | <b>-1.0111</b> | <b>-1.100</b> |
| Round up              | <b>-1.0111</b> | <b>-1.011</b> |
| Truncate              | <b>-1.0111</b> | <b>-1.011</b> |

## Floating-Point Exceptions



- Six types of exception conditions
  - #I: Invalid operation
  - #Z: Divide by zero
  - #D: Denormalized operand
  - #O: Numeric overflow
  - #U: Numeric underflow
  - #P: 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

## Status register



C<sub>3</sub>-C<sub>0</sub>: condition bits after comparisons

## FPU data types



.data

bigVal REAL10 1.212342342234234243E+864

.code

Fld bigVal

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 |

## 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
  - FLBD 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, ...) (FSTSW is the only exception which stores FPU status word to AX)
  - if an instruction has two operands, one must be a FPU register
  - integers must be loaded from memory onto the stack and converted to floating-point before being used in calculations

## Instruction format



**Table 17-9** Basic FPU Instruction Formats.

| Instruction Format         | Mnemonic Format | Operands (Dest, Source) | Example                        |
|----------------------------|-----------------|-------------------------|--------------------------------|
| Classical Stack            | <i>Fop</i>      | {ST(1),ST}              | FADD                           |
| Classical Stack, extra pop | <i>FopP</i>     | {ST(1),ST}              | FSUBP                          |
| Register                   | <i>Fop</i>      | ST(n),ST<br>ST, ST(n)   | FMUL ST(1),ST<br>FDIV ST,ST(3) |
| Register, pop              | <i>FopP</i>     | ST(n),ST                | FADDP ST(2),ST                 |
| Real Memory                | <i>Fop</i>      | {ST},memReal            | FDIVR                          |
| Integer Memory             | <i>Flop</i>     | {ST},memInt             | FSUBR hours                    |

{...}: implied operands

## Classic stack



- ST(0) as source, ST(1) as destination. Result is stored at ST(1) and ST(0) is popped, leaving the result on the top.

```
fld op1           ; op1 = 20.0
fld op2           ; op2 = 100.0
fadd
```

|       | Before |       | After |
|-------|--------|-------|-------|
| ST(0) | 100.0  | ST(0) | 120.0 |
| ST(1) | 20.0   | ST(1) |       |

## Real memory and integer memory



- ST(0) as the implied destination. The second operand is from memory.

```
FADD mySingle      ; ST(0) = ST(0) + mySingle
FSUB mySingle      ; ST(0) = ST(0) - mySingle
FSUBR mySingle     ; ST(0) = mySingle - ST(0)
```

```
FIADD myInteger    ; ST(0) = ST(0) + myInteger
FISUB myInteger    ; ST(0) = ST(0) - myInteger
FISUBR myInteger   ; ST(0) = myInteger - ST(0)
```

## Register and register pop



- Register: operands are FP data registers, one must be ST.

```
FADD    st,st(1)      ; ST(0) = ST(0) + ST(1)
FDIVR   st,st(3)      ; ST(0) = ST(3) / ST(0)
FMUL    st(2),st      ; ST(2) = ST(2) * ST(0)
```

- Register pop: the same as register with a ST pop afterwards.

```
FADDP st(1),st
```

|       | Before | Intermediate | After       |
|-------|--------|--------------|-------------|
| ST(0) | 200.0  | ST(0) 200.0  | ST(0) 232.0 |
| ST(1) | 32.0   | ST(1) 232.0  |             |

## Example: evaluating an expression

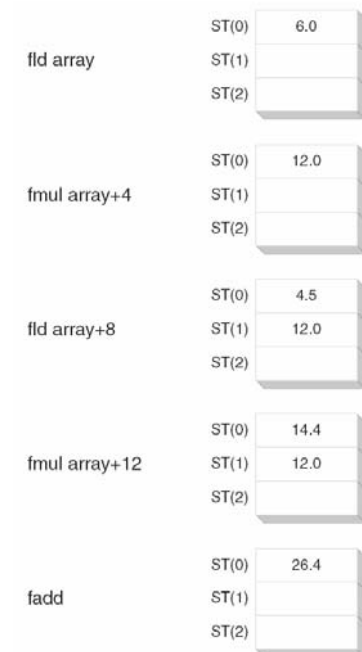


```
INCLUDE Irvine32.inc      (6.0 * 2.0) + (4.5 * 3.2)
.data
array      REAL4 6.0, 2.0, 4.5, 3.2
dotProduct REAL4 ?

.code
main PROC
    finit
    fld array              ; push 6.0 onto the stack
    fmul array+4           ; ST(0) = 6.0 * 2.0
    fld array+8            ; push 4.5 onto the stack
    fmul array+12          ; ST(0) = 4.5 * 3.2
    fadd                   ; ST(0) = ST(0) + ST(1)
    fstp dotProduct        ; pop stack into memory operand
    exit
main ENDP
END main
```

(6.0 \* 2.0) + (4.5 \* 3.2)

```
fld array
fmul array+4
fld array+8
fmul array+12
fadd
fstp dotProduct
```



## Load



|                           |                                                                                                                                                                                 |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>FLD <i>source</i></b>  | loads a floating point number from memory onto the top of the stack. The <i>source</i> may be a single, double or extended precision number or a coprocessor register.          |
| <b>FILD <i>source</i></b> | reads an <i>integer</i> from memory, converts it to floating point and stores the result on top of the stack. The <i>source</i> may be either a word, double word or quad word. |
| <b>FLD1</b>               | stores a one on the top of the stack.                                                                                                                                           |
| <b>FLDZ</b>               | stores a zero on the top of the stack.                                                                                                                                          |
| <b>FLDPI</b>              | stores $\pi$                                                                                                                                                                    |
| <b>FLDL2T</b>             | stores $\log_2(10)$                                                                                                                                                             |
| <b>FLDL2E</b>             | stores $\log_2(e)$                                                                                                                                                              |
| <b>FLDLG2</b>             | stores $\log_{10}(2)$                                                                                                                                                           |
| <b>FLDLN2</b>             | stores $\ln(2)$                                                                                                                                                                 |

## load



```
.data
array REAL8 10 DUP(?)
.code
fld array                ; direct
fld [array+16]           ; direct-offset
fld REAL8 PTR[esi]       ; indirect
fld array[esi]           ; indexed
fld array[esi*8]         ; indexed, scaled
fld REAL8 PTR[ebx+esi]; base-index
fld array[ebx+esi]       ; base-index-displacement
```

## Store



|                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>FST <i>dest</i></b>   | stores the top of the stack (ST0) into memory. The <i>destination</i> may either be a single or double precision number or a coprocessor register.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>FSTP <i>dest</i></b>  | stores the top of the stack into memory just as <b>FST</b> ; however, after the number is stored, its value is popped from the stack. The <i>destination</i> may either a single, double or extended precision number or a coprocessor register.                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>FIST <i>dest</i></b>  | stores the value of the top of the stack converted to an integer into memory. The <i>destination</i> may either a word or a double word. The stack itself is unchanged. How the floating point number is converted to an integer depends on some bits in the coprocessor's <i>control word</i> . This is a special (non-floating point) word register that controls how the coprocessor works. By default, the control word is initialized so that it rounds to the nearest integer when it converts to integer. However, the <b>FSTCW</b> (Store Control Word) and <b>FLDCW</b> (Load Control Word) instructions can be used to change this behavior. |
| <b>FISTP <i>dest</i></b> | Same as <b>FIST</b> except for two things. The top of the stack is popped and the <i>destination</i> may also be a quad word.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

## Store



```
fst  dblOne
fst  dblTwo
fstp dblThree
fstp dblFour
```

|       |       |
|-------|-------|
| ST(0) | 200.0 |
| ST(1) | 32.0  |

## Register



- FXCH ST $n$**  exchanges the values in ST0 and ST $n$  on the stack (where  $n$  is register number from 1 to 7).
- FFREE ST $n$**  frees up a register on the stack by marking the register as unused or empty.

## Arithmetic instructions



**Table 17-12** Basic Floating-Point Arithmetic Instructions.

|              |                                  |
|--------------|----------------------------------|
| <b>FCHS</b>  | Change sign                      |
| <b>FADD</b>  | Add source to destination        |
| <b>FSUB</b>  | Subtract source from destination |
| <b>FSUBR</b> | Subtract destination from source |
| <b>FMUL</b>  | Multiply source by destination   |
| <b>FDIV</b>  | Divide destination by source     |
| <b>FDIVR</b> | Divide source by destination     |

- FCHS** ; change sign of ST
- FABS** ; ST=|ST|

## Floating-Point add



- **FADD**
  - adds source to destination
  - No-operand version pops the FPU stack after addition
- Examples:

|                   |         |       |        |
|-------------------|---------|-------|--------|
| fadd st(1), st(0) | Before: | ST(1) | 234.56 |
|                   |         | ST(0) | 10.1   |
|                   | After:  | ST(1) | 244.66 |
|                   |         | ST(0) | 10.1   |

FADD<sup>4</sup>  
FADD *m32fp*  
FADD *m64fp*

## Floating-Point subtract



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

FSUB<sup>5</sup>  
FSUB *m32fp*  
FSUB *m64fp*  
FSUB ST(0), ST( $i$ )  
FSUB ST( $i$ ), ST(0)

**fsub mySingle** ; ST -= mySingle  
**fsub array[edi\*8]** ; ST -= array[edi\*8]

## Floating-point multiply/divide



- FMUL

FMUL<sup>6</sup>

– Multiplies source by destination, stores product in destination

FMUL *m32fp*  
FMUL *m64fp*  
FMUL ST(0), ST(*i*)  
FMUL ST(*i*), ST(0)
- FDIV

FDIV<sup>7</sup>

– Divides destination by source, then pops the stack

FDIV *m32fp*  
FDIV *m64fp*  
FDIV ST(0), ST(*i*)  
FDIV ST(*i*), ST(0)

## Example: compute distance



```
; compute D=sqrt(x^2+y^2)
fld  x           ; load x
fld  st(0)       ; duplicate x
fmul                ; x*x

fld  y           ; load y
fld  st(0)       ; duplicate y
fmul                ; y*y

fadd                ; x*x+y*y
fsqrt
fst  D
```

## Example: expression



```
; 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
```

## Example: array sum



```
.data
N = 20
array REAL8 N DUP(1.0)
sum REAL8 0.0
.code
mov ecx, N
mov esi, OFFSET array
fldz                 ; ST0 = 0
lp: fadd REAL8 PTR [esi]; ST0 += *(esi)
add esi, 8           ; move to next double
loop lp
fstp sum             ; store result
```

## Comparisons



|                          |                                                                                                                             |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <b>FCOM <i>src</i></b>   | compares ST0 and <i>src</i> . The <i>src</i> can be a coprocessor register or a float or double in memory.                  |
| <b>FCOMP <i>src</i></b>  | compares ST0 and <i>src</i> , then pops stack. The <i>src</i> can be a coprocessor register or a float or double in memory. |
| <b>FCOMPP</b>            | compares ST0 and ST1, then pops stack twice.                                                                                |
| <b>FICOM <i>src</i></b>  | compares ST0 and (float) <i>src</i> . The <i>src</i> can be a word or dword integer in memory.                              |
| <b>FICOMP <i>src</i></b> | compares ST0 and (float) <i>src</i> , then pops stack. The <i>src</i> can be a word or dword integer in memory.             |
| <b>FTST</b>              | compares ST0 and 0.                                                                                                         |

| Instruction                         | Condition Code Bits |    |    |    | Condition              |
|-------------------------------------|---------------------|----|----|----|------------------------|
|                                     | C3                  | C2 | C1 | C0 |                        |
| fcom, fcomp, fcompp, ficom, ficompp | 0                   | 0  | X  | 0  | ST > source            |
|                                     | 0                   | 0  | X  | 1  | ST < source            |
|                                     | 1                   | 0  | X  | 0  | ST = source            |
|                                     | 1                   | 1  | X  | 1  | ST or source undefined |
|                                     | X = Don't care      |    |    |    |                        |

## Comparisons



- The above instructions change FPU's status register of FPU and the following instructions are used to transfer them to CPU.

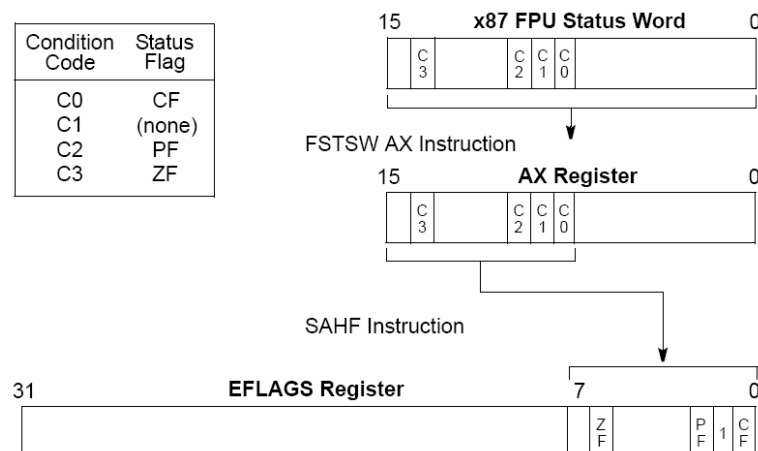
|                          |                                                                                     |
|--------------------------|-------------------------------------------------------------------------------------|
| <b>FSTSW <i>dest</i></b> | Stores the coprocessor status word into either a word in memory or the AX register. |
| <b>SAHF</b>              | Stores the AH register into the FLAGS register.                                     |
| <b>LAHF</b>              | Loads the AH register with the bits of the FLAGS register.                          |

- SAHF** copies C<sub>0</sub> into carry, C<sub>2</sub> into parity and C<sub>3</sub> to zero. Since the sign and overflow flags are not set, use conditional jumps for unsigned integers (**ja**, **jae**, **jb**, **jbe**, **je**, **jz**).

## Comparisons



| Condition Code | Status Flag |
|----------------|-------------|
| C0             | CF          |
| C1             | (none)      |
| C2             | PF          |
| C3             | ZF          |



## Branching after FCOM



- Required steps:
  - Use the **FSTSW** 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.
- Pentium Pro supports two new comparison instructions that directly modify CPU's FLAGS.
 

```
FCOMI ST(0), src ; src=STn
FCOMIP ST(0), src
```

Example

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

## Example: comparison



```
.data
x REAL8    1.0
y REAL8    2.0
.code
    ; if (x>y) return 1 else return 0
    fld  x          ; ST0 = x
    fcomp y         ; compare ST0 and y
    fstsw ax        ; move C bits into FLAGS
    sahf
    jna  else_part  ; if x not above y, ...
then_part:
    mov  eax, 1
    jmp  end_if
else_part:
    mov  eax, 0
end_if:
```

## Example: comparison



```
.data
x REAL8    1.0
y REAL8    2.0
.code
    ; if (x>y) return 1 else return 0
    fld  y          ; ST0 = y
    fld  x          ; ST0 = x ST1 = y
    fcomi ST(0), ST(1)

    jna  else_part  ; if x not above y, ...
then_part:
    mov  eax, 1
    jmp  end_if
else_part:
    mov  eax, 0
end_if:
```

## Comparing for Equality



- Not to compare floating-point values directly because of precision limit. For example,  
 $\text{sqrt}(2.0) * \text{sqrt}(2.0) \neq 2.0$

| instruction              | FPU stack                     |
|--------------------------|-------------------------------|
| <b>fld two</b>           | <b>ST(0): +2.0000000E+000</b> |
| <b>fsqrt</b>             | <b>ST(0): +1.4142135+000</b>  |
| <b>fmul ST(0), ST(1)</b> | <b>ST(0): +2.0000000E+000</b> |
| <b>fsub two</b>          | <b>ST(0): +4.4408921E-016</b> |

## 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:
```

## Miscellaneous instructions



FCHS     ST0 = - ST0 Changes the sign of ST0  
FABS     ST0 = |ST0| Takes the absolute value of ST0  
FSQRT    ST0 =  $\sqrt{\text{ST0}}$  Takes the square root of ST0  
FSCALE   ST0 = ST0  $\times 2^{\text{ST1}}$  multiplies ST0 by a power of 2 quickly. ST1 is not removed from the coprocessor stack.

### .data

x     REAL4     2.75

five REAL4     5.2

### .code

```
fld      five      ; ST0=5.2
fld      x          ; ST0=2.75, ST1=5.2
fscale                   ; ST0=2.75*32=88
                        ; ST1=5.2
```

## Example: quadratic formula



$$ax^2 + bx + c = 0 \quad x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
fild      MinusFour      ; stack -4
fld       a               ; stack: a, -4
fld       c               ; stack: c, a, -4
fmulp     st1,st0         ; stack: a*c, -4
fmulp     st1,st0         ; stack: -4*a*c
fld       b               ; stack: b, -4*a*c
fld       b               ; stack: b, b, -4*a*c
fmulp     st1,st0         ; stack: b*b, -4*a*c
faddp     st1,st0         ; stack: b*b - 4*a*c
```

## Example: quadratic formula



```
ftst                      ; test with 0
fstsw     ax
sahf
jb        no_real_solutions ; if disc < 0, no solutions
fsqrt                      ; stack: sqrt(b*b - 4*a*c)
fstp      disc             ; store and pop stack
fldl      1.0              ; stack: 1.0
fld       a                ; stack: a, 1.0
fscale    1                ; stack: a * 2^(1.0) = 2*a, 1
fdivp     st1,st0          ; stack: 1/(2*a)
fst       one_over_2a      ; stack: 1/(2*a)
```

## Example: quadratic formula



```
fld       b                ; stack: b, 1/(2*a)
fld       disc             ; stack: disc, b, 1/(2*a)
fsubrp    st1,st0          ; stack: disc - b, 1/(2*a)
fmulp     st1,st0          ; stack: (-b + disc)/(2*a)
fstp      root1            ; store in *root1
fld       b                ; stack: b
fld       disc             ; stack: disc, b
fchs      1                ; stack: -disc, b
fsubrp    st1,st0          ; stack: -disc - b
fmul      one_over_2a      ; stack: (-b - disc)/(2*a)
fstp      root2            ; store in *root2
```

## Other instructions



- **F2XM1** ;  $ST = 2^{ST(0)} - 1$ ; ST in  $[-1, 1]$
- **FYL2X** ;  $ST = ST(1) * \log_2(ST(0))$
- **FYL2XP1** ;  $ST = ST(1) * \log_2(ST(0) + 1)$
  
- **FPTAN** ;  $ST(0) = 1; ST(1) = \tan(ST)$
- **FPATAN** ;  $ST = \arctan(ST(1)/ST(0))$
- **FSIN** ; ST=sin(ST) in radius
- **FCOS** ; ST=sin(ST) in radius
- **FSINCOS** ;  $ST(0) = \cos(ST); ST(1) = \sin(ST)$

## 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



- Exception is issued and pended; the next floating-point instruction checks exceptions before it executes.
- 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
```

## 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
```

## Mixed-mode arithmetic



```
int    N=20;
double X=3.5;
int    Z=(int)(N+X);
```

```
fild N
fadd X
fist Z          ; Z=24
```

```
fstcw ctrlWord
or     ctrlWord, 110000000000b ; round mode=truncate
fldcw ctrlWord
fild N
fadd X
fist Z          ; Z=23
```

## 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

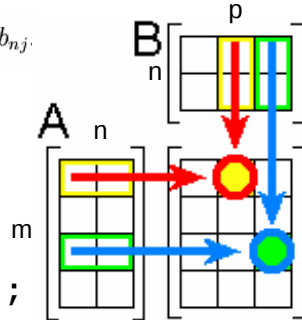
```
.data
ctrlWord WORD ?
.code
fstcw ctrlWord          ; get the control word
and  ctrlWord,111111111111011b ; unmask #Z
fldcw ctrlWord          ; load it back into FPU
```

## Homework #5



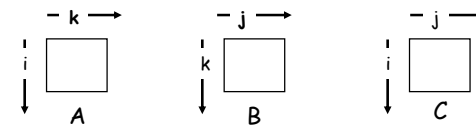
$$(AB)_{ij} = \sum_{r=1}^n a_{ir}b_{rj} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj}.$$

```
for (i=0; i<m; i++)
  for (j=0; j<p; j++) {
    C[i,j]=0;
    for (r=0; r<n; r++)
      C[i,j]+=A[i,r]*B[r,j];
  }
```



- Strassen's algorithm?  $O(n^{\log_2 7}) \approx O(n^{2.807})$
- Coppersmith & Winograd  $O(n^{2.376})$
- Memory coherence

## Homework #5



```
/* ijk */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

## Homework #5



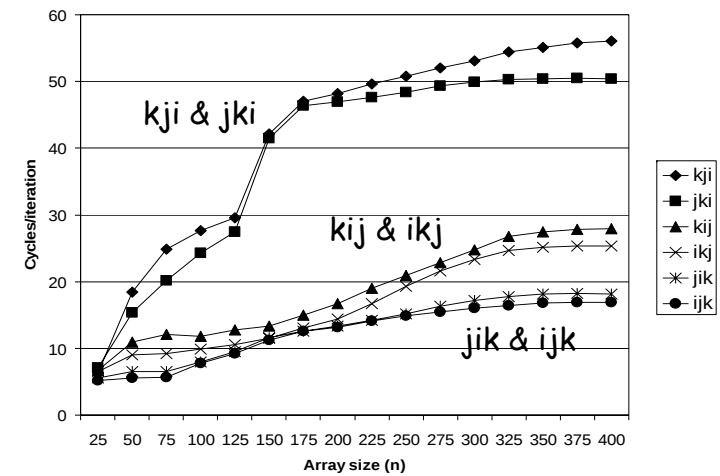
```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

## Homework #5



## Blocked array

