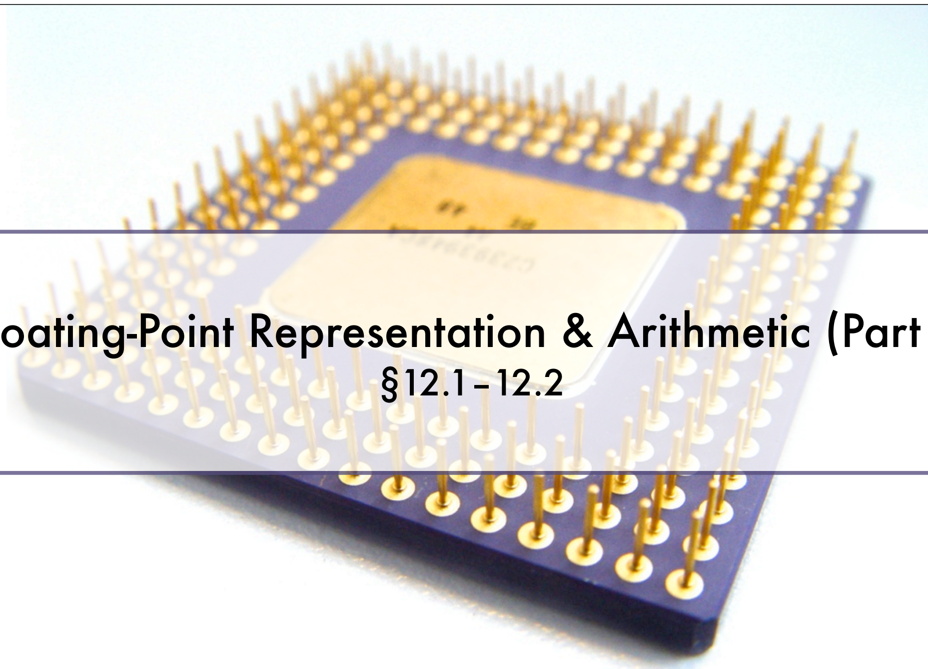
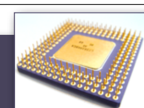


Floating-Point Representation & Arithmetic (Part 2)

§12.1–12.2

Portions based on slides by Kip Irvine for *Assembly Language for x86 Processors*, 6/e. © 2010 Pearson Education. All rights reserved.

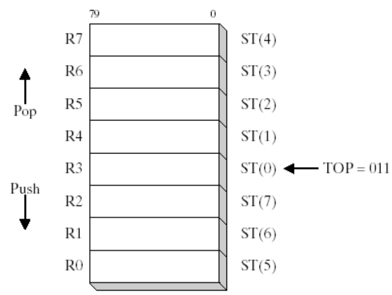
Floating Point Unit



- Originally, the FPU was a separate, optional coprocessor ("math coprocessor"), separate from the CPU
  - 8086 CPU + 8087 FPU
  - 80386 CPU + 80387 FPU
- FPU was integrated in the 486DX processor

FPU Register Stack

- FPU has eight 80-bit data registers
- Treated as a stack (the "FPU stack")
- ST(0) is the value on the top of the stack, ST(1) below that on the stack, etc.



FPU Instruction Set

- Zero, one, or two operands, like other instructions
- No immediate operands
- No general-purpose registers (EAX, EBX, etc.)
  - Load values from memory
- Integers must be loaded from memory onto the stack and converted to floating-point

MASM Data Types

- REAL4 – IEEE 754 single-precision (32 bits/4 bytes)
- REAL8 – IEEE 754 double-precision (64 bits/8 bytes)

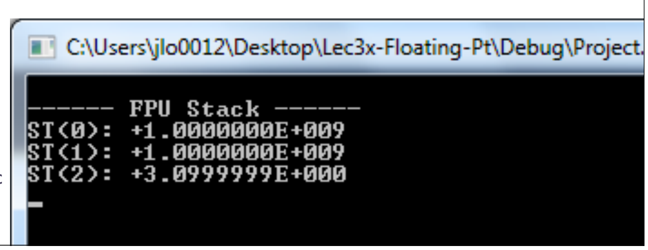
```
.data
three_point_one REAL4 3.1
one_billion     REAL8 1.0e9   ; 1.0 ×10^9
```

Initialization, Load, Store Instructions

- FINIT – initialize the FPU (call this at the beginning of main)
- FLD – load a value from memory, pushing it onto the stack at ST(0)
- FST – copy (store) the value from ST(0) into memory
- FSTP – store, then pop the value off the FPU stack

```
.data
three_point_one REAL4 3.1
one_billion     REAL8 1.0e9
.code
finit
fld   three_point_one
fld   one_billion
fld   one_billion
call  ShowFPUStack
```

Irvine32 library proc (*Note*: do not use if stack is empty)



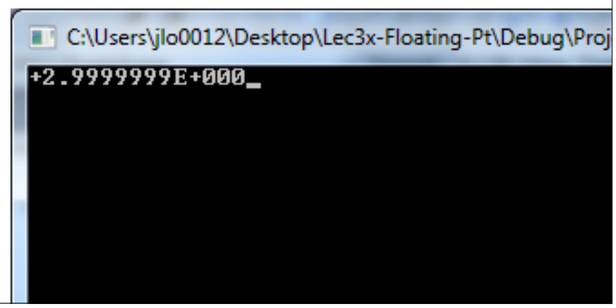
+ – × ÷

- FADD, FSUB, FMUL, FDIV with no operands:
  - Compute ST(1) *op* ST(0)
  - Pop both of those values
  - Push the result

```
.data
f_3_1 REAL4 3.1
f_0_1 REAL4 0.1

.code
finit
fld  f_3_1
fld  f_0_1
fsub
call WriteFloat
```

Irvine32 library routine (display ST(0))



Arithmetic Using FPU Stack

- To evaluate a complex formula, convert it to Reverse Polish Notation (RPN), then translate into assembly
  - Formula: $a + \sqrt{}(b \times c) - d$
  - RPN: $a\ b\ c \times \sqrt{}\ + d\ -$
  - Assembly:

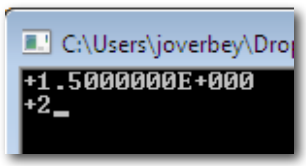
```
fld a
fld b
fld c
fmul
fsqrt
fadd
fld d
fsub
```

Integer Load/Store Instructions

- FILD – convert an SDWORD to floating point and push onto the stack at ST(0)
- FIST – round ST(0) to an integer and copy (store) into memory
- FISTP – store, then pop the value off the FPU stack

```
.data
neg_three     SDWORD  -3
four_pt_five  REAL4   4.5
int_result    SDWORD  ?
float_result  REAL4   ?
```

```
.code
finit
fild neg_three
fld  four_pt_five
fadd
call WriteFloat
call Crlf
fst float_result
fistp int_result
mov eax, int_result
call WriteInt
```



More Operations

- Load mathematical constants (e.g., π)
- Sine, cosine, square root, etc.
- Compare floating point values
- Load/store control word
  (to unmask exceptions, change rounding, etc.)

- *See textbook: §§12.1–1.2 and Appendix B.3*
- *You will only be responsible for the material discussed in lecture,* ***not*** *the rest of the material in §12.2*

Comparisons Instructions

- FCOMI ST(0), ST(*i*) – compare ST(0) to ST(*i*), copying the flags to the CPU
  - Floating-point numbers are signed, but this sets the zero and carry flags (and parity flag)
  - So, follow with an **unsigned** conditional jump (!!!) – `jb`, `ja`, etc.
- FCOMIP pops afterward. (Use `ffree st(0)` then `fincstp` to manually pop.)

```
.data
f0p1 REAL4 0.1
f0p2 REAL4 0.2
```

```
.code
    finit
    fld f0p2        ; Note the order we push!
    fld f0p1
    fcomi ST(0), ST(1)
    jae above_or_eq

    ; If we're here, ST(0) < ST(1): 0.1 < 0.2
    mov al, "<"
    call WriteChar

above_or_eq:
```

Caveats

- Do not compare floating-point values for equality
  - E.g (REAL4), $3.1 - 0.1 \neq 3.0$
  - Instead, test whether $|n_2 - n_1| < \varepsilon$, for some small value of ε
- In general, floating-point addition, multiplication:
  - Are **not** associative $(a + b) + c \neq a + (b + c)$
  - Multiplication does **not** distribute over addition $a(b + c) \neq ab + ac$