CS220 – Logic Design
AS07-FPU Programming

• Outline
  – The Numeric Coprocessor
  – FPU Instructions
  – FPU Calculation Examples
The Numeric Coprocessor

- All Intel processors since the 80486 have a built-in numeric coprocessor or FPU (floating point unit). On earlier processors the FPU was a separate (optional) chip.

- The coprocessor or FPU (floating point unit) has eight floating point registers named ST0, ST1, ..., ST7. The registers are 80 bits (10 bytes) wide. Arithmetic is always performed using an 80 bit extended precision number format.
Floating point numbers can be stored in memory in single (4 byte), double (8 byte), or extended (10 byte) precision formats. (You can declare extended precision variables in C/C++ by using the `long double` type.)

The 8 registers are arranged in a stack or LIFO. ST0 refers to the top of the stack.

There is also a status register that is useful when comparing floating point numbers.
The following instructions are used to push (load) data onto the FPU stack (Note: We are now working with two stacks, the CPU stack and the FPU stack):

- `fld src` # Push from mem or an FPU reg
- `fild isrc` # Push integer from memory
- `fld1` # Push 1.0
- `fldz` # Push 0.0
- `fldpi` # Push \( \pi \) (an approximation to it)
AS07-FPU Programming
FPU Instructions

• These instructions copy (store) data from the stack (some also pop the FPU stack):
  
  - `fst  dest`  # Copy ST0 to mem/FPU reg
  - `fstp  dest` # Pop ST0 to mem/FPU reg
  - `fist  idest` # Copy ST0 to mem as int
  - `fistp idest` # Pop ST0 to mem as integer
  - `fxch %st(n)` # Exchange ST0 and FPU reg
  - `ffree %st(n)` # Mark FPU reg as empty
Instructions that use memory references to floating point numbers need to be qualified with suffixes s (32 bit), l (64 bit), or t (80 bit) and references to integers need suffixes s (16 bit), l (32 bit), or q (quad, 8 byte/80 bit).

fld and fstp are the only instructions that reference floating point numbers that allow the t suffix. fild and fistp are the only instructions that reference integer numbers that allow the q suffix.
AS07-FPU Programming
FPU Instructions

• For addition:

  \[
  \begin{align*}
  &\text{fadd } \text{src} & \quad \# \text{ ST0 } += \text{ src} \\
  &\text{fadd } \%\text{st}(0),\%\text{st}(n) & \quad \# \text{ STn } += \text{ ST0} \\
  &\text{faddp } \%\text{st}(0),\%\text{st}(n) & \quad \# \text{ STn } += \text{ ST0}, \text{ pop} \\
  &\text{fiadd } \text{isrc} & \quad \# \text{ ST0 } += \text{ isrc}
  \end{align*}
  \]

• There are analogous instructions for multiplication (\text{fmul}, \text{fmulp}, \text{fimul}).
And for subtraction:

- `fsub src` \# ST0 = ST0 – src
- `fsub %st(0),%st(n)` \# STn = ST0 – STn
- `fsubp %st(0),%st(n)` \# STn = ST0 – STn, pop
- `fsubr src` \# ST0 = src – ST0
- `fsubr %st(0),%st(n)` \# STn = STn – ST0
- `fsubrp %st(0),%st(n)` \# STn = STn – ST0, pop
- `fisub isrc` \# ST0 = ST0 – isrc
- `fisubr isrc` \# ST0 = isrc – ST0

There are similar instructions for division.
AS07-FPU Programming

FPU Instructions

- An example, sum an element array \((xarr)\):

```assembly
fldz                  # sum = 0.
movl $0, %eax        # n = 0
begfor:
cmpl size, %eax      # sum+=xarr[n]
faddl xarr(,%eax,8)   # sum+=xarr[n]
incl %eax
jmp begfor
endfor:
call print_double    # sum+=xarr[n]
ffree %st(0)
```
AS07-FPU Programming
FPU Instructions

• The **fcom** instructions are used to compare two floating point numbers:

  - `fcom src` # ST0 – src
  - `fcomp src` # ST0 – src, pop
  - `fcompp` # ST0 – ST1, pop, pop
  - `ficom isrc` # ST0 – isrc
  - `ficomp isrc` # ST0 – isrc, pop
  - `ftst` # ST0 – 0
The `fcom` instructions set bits in the FPU status word. This word must be transferred to the CPU FLAGS register in order to use the conditional jump instructions:

- `fstsw %ax`  # Copy FPU status to AX
- `fstsw dest`  # Copy FPU stat to mem
- `sahf`  # Copy AH to FLAGS
- `lahf`  # Copy FLAGS to AH

The status bits are set as if the comparison is between two unsigned integers.
Here is code for an “if (x > y)” structure:

```plaintext
fldl x   # ST0 = x
fcompl y   # compare STO and y
fstsw %ax # move status to FLAGS
sahf
jbe else_part   # Note jbe, not jle

then_part:
# CODE FOR THEN PART GOES HERE
jmp end_if

else_part:
# CODE FOR ELSE PART GOES HERE
end_if:
```
Compute $\sqrt{x}$:

```
# Read value from input
call     read_double # x
fsqrt     # x^(0.5)
# Display the result
call     print_double
# Clear the FPU stack
ffree     %st(0)     # empty
call     print_nl
```
• Compute $\sqrt{x}$ where $x$ is an integer:

```plaintext
call read_int
# Push value onto CPU stack
pushl %eax
# Load value onto FPU stack
fld (%esp)          # x
# Restore CPU stack
addl $4, %esp
fqrt               # x^(0.5)
call print_double
ffree %st(0)        # empty
call print_nl
```
There is no way to move a value directly from (to) a CPU register to (from) an FPU register. The CPU stack may be used to temporarily store values when transferring between the CPU and FPU.

As shown in the previous examples, I highly recommend keeping track of what FPU registers are in use with comments.
- Compute \( y = \tan(\pi/8) \) where \( y \) labels a double variable in the data or bss sections:

```assembly
fldpi                           # pi
pushl $8
fildl (%esp)                   # 8, pi
addl $4, %esp
fdivrp %st(0),%st(1)           # pi/8
fptan                          # 1, \tan(\pi/8)
fstp %st(0)                    # \tan(\pi/8)
fstpl y                        # empty
```
• Compute cosine of angles between 0 and 90 in 10 degree increments:

```assembly
.section .data
CON90: .double 90
CON10: .double 10
.section .text
pushl $180
fld (%esp)    # 180
addl $4,%esp
fldpi        # pi, 180
fdivp %st(1)  # pi/180
# Create floating point 0
fldz          # deg=0, pi/180
```

begwhile:
  fcoml CON90
  fstsw %ax
  sahf
  ja    endwhile
  call  print_double
  fld   %st(0)   # deg, deg, pi/180
  fmul  %st(2)   # rad, deg, pi/180
  fcos           # cos, deg, pi/180
  call  print_double
  fstp  %st(0)   # deg, pi/180
  faddl CON10    # deg=deg+10, pi/180
  jmp   begwhile
endwhile:
Here are some examples of copying values from the FPU stack to the CPU stack:

```
subl $4, %esp
fsts (%esp)

subl $8, %esp
fstl (%esp)

# Use 12 instead of 10 to keep stack
# 4-byte aligned
subl $12, %esp
fstpt (%esp)
```