Assignment Number # 02

IA32 Architecture Floating point unit

The FPU (floating-point unit) on systems based on the IA-32 architecture contains eight floating-point registers the system uses for numeric calculations, status and control words, and error pointers. You normally need to consider only the status and control words, and then only when customizing your floating-point environment.

The FPU status and control words correspond to 16-bit registers whose bits hold the value of a state of the FPU or control its operation. Intel Fortran defines a set of symbolic constants to set and reset the proper bits in the status and control words.

Note:

> The symbolic constants and the library routines used to read and write the control and status registers only affect the x87 control and status registers. They do not affect the MXCSR register (the control and status register for the Intel(R) SSE and Intel(R) SSE2 instructions).

They do not affect the MXCSR (the control and status register for the Intel(R) SSE and Intel(R) SSE2 instructions). For example: shown in figure:

```
USE IFPORT
INTEGER(2) status, control, controlo, mask_all_traps
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
!     Save old control word
controlo = control
!     Clear the rounding control flags
control = IAND(control,NOT(FPCW$MCW_RC))
!     Set new control to round up
control = IOR(control,FPCW$UP)
CALL SETCONTROLFPQQ(control)
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
! Demonstrate setting and clearing exception mask flags
mask_all_traps = FPCW$INVALID + FPCW$DENORMAL + &
    FPCW$ZERODIVIDE + FPCW$OVERFLOW + &
FPCW$UNDERFLOW + FPCW$INEXACT
!     Clear the exception mask flags
control = IAND(control,NOT(FPCW$MCW_EM))
!     Set new exception mask to disallow overflow
!     (i.e., enable overflow traps)
! but allow (i.e., mask) all other exception conditions.
control = IOR(control,IEOR(mask_all_traps,FPCW$OVERFLOW))
CALL SETCONTROLFPQQ(control)
CALL GETCONTROLFPQQ(control)
WRITE (*, 9000) 'Control word: ', control
9000 FORMAT (1X, A, Z4)
END
```

The status and control symbolic constants (such as FPCW$OVERFLOW and FPCW$CHOP in the preceding example) are defined as INTEGER (2) parameters in the module IFORT.F90 in the *...\INCLUDE* folder. The status and control words are logical combinations (such as with .AND.) of different parameters for different FPU options.

The name of a symbolic constant takes the general form *name*$*opton*

Prefixes and parameter Flag

name	Meaning
FPSW	Floating-point status word
FPCW	Floating-point control word
SIG	Signal
FPE	Floating-point exception
MTH	Math function

The suffix *option* is one of the options available for that *name*. The parameter *name*$*option* corresponds either to a status or control option (for example, FPSW$ZERODIVIDE, a status word parameter that shows whether a zero-divide exception has occurred or not) or *name*$*option* corresponds to a mask, which sets all symbolic constants to 1 for all the options of *name*. You can use the masks in logical functions (such as IAND, IOR, and NOT) to set or to clear all options for the specified *name*. The following sections define the *option*s and illustrate their use with examples.

You can control the floating-point processor options (on systems based on the IA-32 architecture) and find out its status with the run-time library routines GETSTATUSFPQQ (IA-32 architecture only), GETCONTROLFPQQ (IA-32 architecture only), and SETCONTROLFPQQ (IA-32 architecture only).

Understanding Floating-point Status Word

On systems based on the IA-32 architecture, the FPU status word includes bits that show the floating-point exception state of the processor. The status word parameters describe six exceptions: invalid result, demoralized operand, zero divide, overflow, underflow and inexact precision. These are described in the section, loss of precision Error section. When one of the bits is set to 1, it means a past floating-point operation produced that exception type. (Intel Fortran initially clears all status bits. It does not reset the status bits before performing additional floating-point operations after an exception occurs. The status bits accumulate.)

The following table shows the floating-point exception status parameters.

Parameter Name	Value in Hex	Description
FPSW$MSW_EM	#003F	Status Mask (set all bits to 1)
FPSW$INVALID	#0001	An invalid result occurred
FPSW$DENORMAL	#0002	A denormal operand occurred
FPSW$ZERODIVIDE	#0004	A divide by zero occurred
FPSW$OVERFLOW	#0008	An overflow occurred
FPSW$UNDERFLOW	#0010	>An underflow occurred
FPSW$INEXACT	#0020	Inexact precision occurred

You can find out which exceptions have occurred by retrieving the status word and comparing it to the exception parameters.

Loss of Precision Errors

If a real number is not exactly one of the representable floating-point numbers, then the nearest floating-point number must represent it. The rounding error is the difference between the exact real number and it's nearest floating-point representation. If the rounding error is non-zero, the rounded floating-point number is called *inexact*.

Normally, calculations proceed when an inexact value results. Almost any floating-point operation can produce an inexact result. The rounding mode (round up, round down, round nearest, truncate) is determined by the floating-point control word.

If an arithmetic operation results in a floating-point number that cannot be represented in a specific data type, the operation may produce a special value: signed zero, signed

infinity, NaN, or a DE normal. Numbers that have been rounded to an exactly representable floating-point number also result in a special value. Special-value results are a limiting case of the arithmetic operation involved. Special values can propagate through your arithmetic operations without causing your program to fail, and often provide usable results.

If an arithmetic operation results in an exception, the operation can cause an underflow or overflow:

- Underflow occurs when an arithmetic result is too small for the math processor to handle. Depending on the setting of the `/fpe` compiler option, underflows are set to zero (they are usually harmless) or they are left as is (demoralized).
- Overflow occurs when an arithmetic result is too large for the math processor to handle. Overflows are more serious than underflows, and may indicate an error in the formulation of a problem (for example, unintended exponentiation of a large number by a large number). Overflows generally produce an appropriately signed infinity value. (This depends on the rounding mode as per the IEEE standard.)

An arithmetic operation can also throw the following exceptions: divide-by-zero exception, an invalid exception, and an inexact exception.

You can select how exceptions are handled by setting the floating-point control word.

On Windows* systems, you can select how exceptions are handled by setting the floating-point control word.

Floating-point Control Word Overview

On systems based on the IA-32 architecture, the FPU control word includes bits that control the FPU's precision, rounding mode, and whether exceptions generate signals if they occur. You can read the control word value with `GETCONTROLFPQQ` (IA-32 architecture only) to find out the current control settings, and you can change the control word with `SETCONTROLFPQQ` (IA-32 architecture only).

The `GETCONTROLFPQQ` and `SETCONTROLFPQQ` routines only affect the x87 status register. They do not affect the `MXCSR` register (the control and status register for the SSE and SSE2 instructions).

Each bit in the floating-point control word corresponds to a mode of the floating-point math processor. The *`IFORT.F90`* module file in the *`...\INCLUDE`* folder contains the INTEGER (2) parameters defined for the control word, as shown in the following table.

Parameter Name	Value in Hex	Description
FPCW$MCW_PC	#0300	Precision control mask
FPCW$64	#0300	64-bit precision
FPCW$53	#0200	53-bit precision
FPCW$24	#0000	24-bit precision
FPCW$MCW_RC	#0C00	Rounding control mask
FPCW$CHOP	#0C00	Truncate
FPCW$UP	#0800	Round up
FPCW$DOWN	#0400	Round down
FPCW$NEAR	#0000	Round to nearest
FPCW$MCW_EM	#003F	Exception mask
FPCW$INVALID	#0001	Allow invalid numbers
>FPCW$DENORMAL	#0002	Allow denormals (very small numbers)
FPCW$ZERODIVIDE	#0004	Allow divide by zero
FPCW$OVERFLOW	#0008	Allow overflow
FPCW$UNDERFLOW	#0010	Allow underflow
FPCW$INEXACT	#0020	Allow inexact precision

Program :

```
INCLUDE irvine.inc
.DATA
      result
     array
      REAL8 0.0
      REAL4 -1.0, 1.2, 2.3, 3.4, 4.567, 0.0
      SDWORD -1
       ; End of array -> NaN
.CODE
main PROC
     xor ebx, ebx
     mov eax, DWORD PTR array[ebx]
     cmp eax, -1
                    je @F ; "DWORD PTR" = "REAL4 PTR"
; NaN = end of array?
; Yes -> Jump to the next @@
      fld DWORD PTR array[ebx]
     fstp QWORD PTR result
    printf("%f\n",result) ; Load a single into FPU ...
    add ebx, 4
    jmp @B ; REAL4 has 4 bytes
   ; Jump to the previous @@
@@:
exit 0
  main ENDP
  END main
```

Program 2:

```
INCLUDE irvine.inc
.DATA
         result REAL8 0.0
        lpstring DWORD 0
         array   REAL4 16 DUP (0.0)
         SDWORD -1

.CODE
 main PROC
xor ebx, ebx
@@:
   mov esi, input("Enter number here ",62," ") ; Input string ... STRING!
   cmp BYTE PTR [esi], 0;
je @F
; Yes -> jump forward to the next
@@
   push ebx
 to save
; StrToFloat changes EBX! So it isINVOKE StrToFloat, esi, ADDR result
```

```
pop ebx ; Convert string to double
; Restore the saved EBX
fld REAL8 PTR result
fstp REAL4 PTR array[ebx]
mov eax, -1
mov DWORD PTR array[ebx+4], eax ;
;
;
;

;

;
add ebx, 4
array
jmp @B
Load a double ...
... and save it as single
NaN = end of array
Store the NaN as the next element
; Pointer to the next REAL4 in
; Jump back to the previous @@
@@:
xor ebx, ebx
@@:
mov eax, DWORD PTR array[ebx]
cmp eax, -1
je @F
fld DWORD PTR array[ebx]
fstp QWORD PTR result
printf("%f\n",result)
function
add ebx, 4
jmp @B
@@:
exit 0
main ENDP
Example 3:
.386
.MODEL FLAT
PUBLIC atofproc
false EQU 0
true EQU 1
.DATA
ten REAL4 10.0
point BYTE ?
minus BYTE ?
digit WORD ?
```

```
; "DWORD PTR" = "REAL4 PTR"
; NaN = end of array?
; Yes -> Jump to the next @@
; Load a single into FPU ...
; ... and store it as double
; MASM32 macro that acts like the C
; REAL4 has 4 bytes
; Jump to the previous @@.CODE
push ebp ; establish stack frame
mov ebp, esp
push eax ; save registers
push ebx
push esi
fld1 ; divisor := 1.0
fldz ; value := 0.0
mov point, false ; no decimal point found yet
mov minus, false ; no minus sign found yet
mov esi, [ebp+8] ; address of first source character
cmp BYTE PTR [esi], '-' ; leading minus sign?
jne endifMinus ; skip if not
mov minus, true ; minus sign found
inc esi ; point at next source character
endifMinus:
whileOK: mov bl, [esi] ; get next character
cmp bl, '.' ; decimal point?
jne endifPoint ; skip if not
mov point, true ; found decimal point
jmp nextChar
endifPoint:
cmp bl, '0' ; character a digit?
jl endwhileOK ; exit if lower than '0'
cmp bl, '9'
jg endwhileOK ; exit if higher than '9'
and bx, 000fh ; convert ASCII to integer value
mov digit, bx ; put integer in memory
fmul ten ; value := value * 10
fiadd digit ; value := value + digit
cmp point, true ; already found a decimal point?
jne endifDec ; skip if not
fxch ; put divisor in ST and value in ST(1)
fmul ten ; divisor := divisor * 10
fxch ; value back to ST; divisor back to ST(1)
endifDec:nextChar: inc esi ; point at next source character
jmp whileOK
endwhileOK:
fdivr ; value := value / divisor
cmp minus, true ; was there a minus sign?
```

```
jne endifNeg
fchs ; value := -value
endifNeg:
pop esi ; restore registers
pop ebx
pop eax
pop ebp
ret 4
atofproc ENDP
END
```