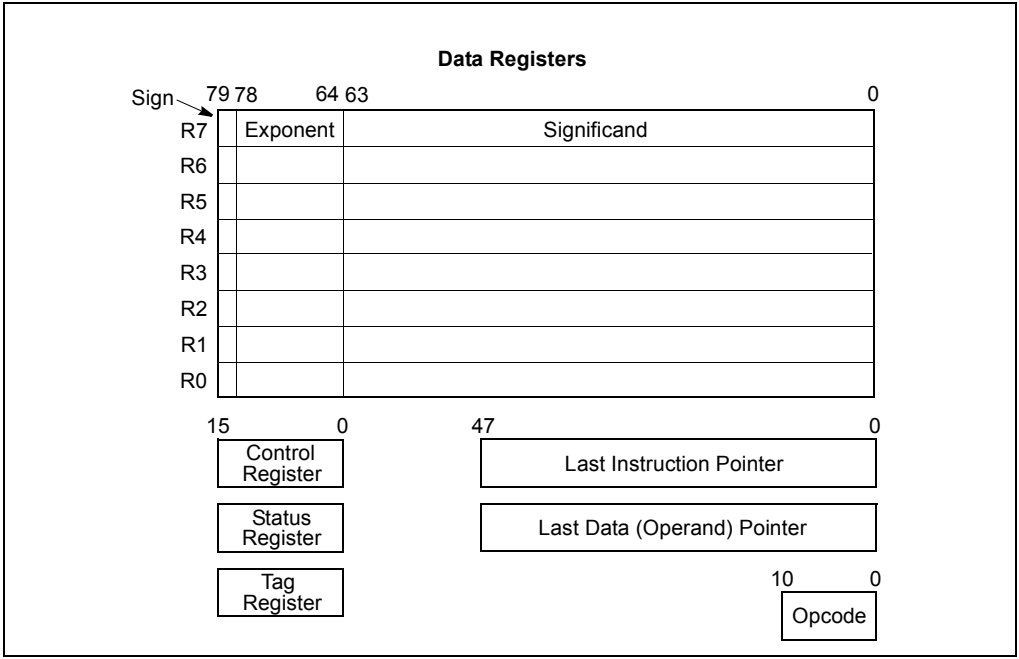

Figure 8-1. x87 FPU Execution Environment

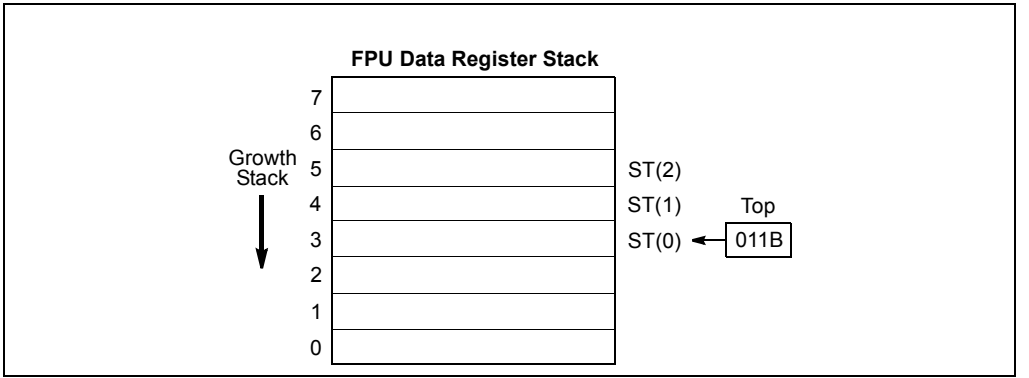

Figure 8-2. x87 FPU Data Register Stack

Computation
Dot Product = (5.6 x 2.4) + (3.8 x 10.3)

Code:

```
FLD  value1 ;(a) value1=5.6
FMUL value2 ;(b) value2=2.4
FLD  value3 ; value3=3.8
FMUL value4 ;(c)value4=10.3
FADD ST(1)  ;(d)
```

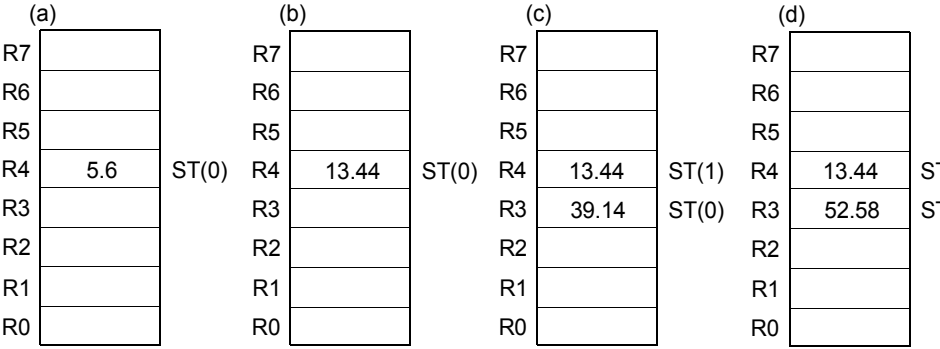

Figure 8-3. Example x87 FPU Dot Product Computation

8.3.4. Load Constant Instructions

The following instructions push commonly used constants onto the top [ST(0)] of the x87 FPU register stack:

FLDZ	Load +0.0
FLD1	Load +1.0
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

The constant values have full double extended-precision floating-point precision (64 bits) and are accurate to approximately 19 decimal digits. They are stored internally in a format more precise than double extended-precision floating point. When loading the constant, the x87 FPU rounds the more precise internal constant according to the RC (rounding control) field of the x87 FPU control word. See Section 8.3.8., "Pi", for information on the π constant.

8.3.5. Basic Arithmetic Instructions

The following floating-point instructions perform basic arithmetic operations on floating-point numbers. Where applicable, these instructions match IEEE Standard 754:

FADD/FADDP	Add floating point
FIADD	Add integer to floating point
FSUB/FSUBP	Subtract floating point
FISUB	Subtract integer from floating point
FSUBR/FSUBRP	Reverse subtract floating point
FISUBR	Reverse subtract floating point from integer
FMUL/FMULP	Multiply floating point
FIMUL	Multiply integer by floating point
FDIV/FDIVP	Divide floating point
FIDIV	Divide floating point by integer
FDIVR/FDIVRP	Reverse divide
FIDIVR	Reverse divide integer by floating point
FABS	Absolute value
FCHS	Change sign
FSQRT	Square root

FADD/FADDP/FIADD—Add

Opcode	Instruction	Description
D8 /0	FADD *m32fp*	Add *m32fp* to ST(0) and store result in ST(0)
DC /0	FADD *m64fp*	Add *m64fp* to ST(0) and store result in ST(0)
D8 C0+i	FADD ST(0), ST(i)	Add ST(0) to ST(i) and store result in ST(0)
DC C0+i	FADD ST(i), ST(0)	Add ST(i) to ST(0) and store result in ST(i)
DE C0+i	FADDP ST(i), ST(0)	Add ST(0) to ST(i), store result in ST(i), and pop the register stack
DE C1	FADDP	Add ST(0) to ST(1), store result in ST(1), and pop the register stack
DA /0	FIADD *m32int*	Add *m32int* to ST(0) and store result in ST(0)
DE /0	FIADD *m16int*	Add *m16int* to ST(0) and store result in ST(0)

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is –0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated.

FADD/FADDP/FIADD—Add (Continued)

		DEST						
		−∞	−F	−0	+0	+F	+∞	NaN
SRC	-∞	-∞	-∞	-∞	-∞	-∞	*	NaN
	−F or −I	-∞	−F	SRC	SRC	±F or ±0	+∞	NaN
	−0	-∞	DEST	−0	±0	DEST	+∞	NaN
	+0	-∞	DEST	±0	+0	DEST	+∞	NaN
	+F or +I	-∞	±F or ±0	SRC	SRC	+F	+∞	NaN
	+∞	*	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

```
IF instruction is FIADD
    THEN
        DEST ← DEST + ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* source operand is floating-point value *)
        DEST ← DEST + SRC;
FI;
IF instruction ← FADDP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of unlike sign.

```
; File: double1.asm
;
; Using C printf function to print double values
;

; Declare some external functions
;
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section

msg:    db "Answer: %f", 10, 0          ; The string to print.
pi:     dq 3.14159265


        SECTION .text                   ; Code section.

        global main
main:
        push    ebp                     ; set up stack frame
        mov     ebp,esp

        mov     eax, [pi+4]
        push    eax
        mov     eax, [pi]
        push    eax

        ; Answer should be at the top of the stack
        push    DWORD msg               ; address of ctrl string
        call    printf                  ; Call C function
        add     esp, 12                 ; pop 2 args from stack

        sub     esp, 8
        fld     QWORD [pi]
        fstp    QWORD [esp]

        push    DWORD msg               ; address of ctrl string
        call    printf                  ; Call C function
        add     esp, 12                 ; pop 2 args from stack

        ; return from main
        mov     esp, ebp                ; takedown stack frame
        pop     ebp                     ;   same as "leave" op

        ret
```

```
linuxserver2% nasm -f elf double1.asm
linuxserver2% gcc double1.o

linuxserver2% a.out
Answer: 3.141593
Answer: 3.141593
```

```
; File: double2.asm
;
; Using C printf function to print double values
;

; Declare some external functions
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section
msg:    db "Answer: %f", 10, 0          ; The string to print.
dv1:    dq 1.111
dv2:    dq 2.222
dv3:    dq 3.333
dv4:    dq 4.444
dv5:    dq 5.555
dv6:    dq 6.666
dv7:    dq 7.777
dv8:    dq 8.888
dv9:    dq 9.999
dva:    dq 10.101010


        SECTION .text                   ; Code section.

        global main
main:
        push    ebp                     ; set up stack frame
        mov     ebp,esp
        push    ebx

        fld     QWORD [dv1]
        fld     QWORD [dv2]
        fld     QWORD [dv3]

        sub     esp, 8
        push    DWORD msg               ; address of ctrl string
        mov     ebx, 3

loop1:  fstp    QWORD [esp+4]
        call    printf                  ; Call C function
        dec     ebx
        jnz     loop1

        add     esp, 12                 ; pop 2 args from stack

        ; return from main
        pop     ebx
        mov     esp, ebp                ; takedown stack frame
        pop     ebp                     ;   same as "leave" op
        ret
```

```
linuxserver2% nasm -f elf double2.asm
linuxserver2% gcc double2.o

linuxserver2% ./a.out
Answer: 3.333000
Answer: 2.222000
Answer: 1.111000
```

```
; File: double3.asm
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section
msg:    db "Answer: %f", 10, 0          ; The string to print.
dv1:    dq 1.111
dv2:    dq 2.222
dv3:    dq 3.333
dv4:    dq 4.444
dv5:    dq 5.555
dv6:    dq 6.666
dv7:    dq 7.777
dv8:    dq 8.888
dv9:    dq 9.999
dva:    dq 10.101010

        SECTION .text                   ; Code section.
        global main
main:
        push    ebp                     ; set up stack frame
        mov     ebp,esp
        push    ebx

        fld     QWORD [dv1]
        fld     QWORD [dv2]
        fld     QWORD [dv3]
        fld     QWORD [dv4]
        fld     QWORD [dv5]
        fld     QWORD [dv6]
        fld     QWORD [dv7]
        fld     QWORD [dv8]
        fld     QWORD [dv9]
        fld     QWORD [dva]

        sub     esp, 8
        push    DWORD msg               ; address of ctrl string
        mov     ebx, 10

loop1:  fstp    QWORD [esp+4]
        call    printf                  ; Call C function
        dec     ebx
        jnz     loop1

        add     esp, 12                 ; pop 2 args from stack

        ; return from main
        pop     ebx
        mov     esp, ebp                ; takedown stack frame
        pop     ebp                     ;   same as "leave" op
        ret
```

```
linuxserver2% nasm -f elf double3.asm
linuxserver2% gcc double3.o

linuxserver2% ./a.out
Answer: nan
Answer: nan
Answer: 8.888000
Answer: 7.777000
Answer: 6.666000
Answer: 5.555000
Answer: 4.444000
Answer: 3.333000
Answer: nan
Answer: nan
```

```
; File: double4.asm
;
; Using C printf function to print double values
; Checking out floating point arithmetic
;

; Declare some external functions
;
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section

msg:    db "Answer: %f", 10, 0          ; The string to print.
dv1:    dq 1.111
dv2:    dq 2.222
dv3:    dq -3.333
dv4:    dq -4.444
dv5:    dq 5.555
dv6:    dq 6.666
dv7:    dq 7.777


        SECTION .text                   ; Code section.

        global main
main:
        push    ebp                     ; set up stack frame
        mov     ebp,esp

        sub     esp, 8
        push    DWORD msg               ; address of ctrl string

        fld     QWORD [dv1]
        fld     QWORD [dv2]
        fadd    st0, st1                ; floating point add
        fstp    QWORD [esp+4]
        call    printf                  ; Call C function

        ; note that 1.111 is still on the FPU stack

        fld     QWORD [dv3]
        fsubp   st1, st0                ; st1 := st1 - st0, pop
        fstp    QWORD [esp+4]
        call    printf                  ; Call C function

        ; note that FPU stack is at bottom

        fld     QWORD [dv3]
        fld     QWORD [dv4]
        fmulp   st1, st0                ; f.p. multiply + pop
        fstp    QWORD [esp+4]
        call    printf
```

```
fld     QWORD [dv6]
fld     QWORD [dv3]
fdivp   st1, st0                ; f.p. divide + pop
fstp    QWORD [esp+4]
call    printf

fld     QWORD [dv7]
fsqrt                           ; Compute the square root
fstp    QWORD [esp+4]
call    printf                  ; Call C function

add     esp, 12                 ; pop 2 args from stack

; return from main
mov     esp, ebp                ; takedown stack frame
pop     ebp                     ;   same as "leave" op

ret
```

```
linuxserver2% nasm -f elf double4.asm
linuxserver2% gcc double4.o

linuxserver2% ./a.out
Answer: 3.333000
Answer: 4.444000
Answer: 14.811852
Answer: -2.000000
Answer: 2.788727
```

FCOM/FCOMP/FCOMPP—Compare Floating Point Values

Opcode	Instruction	Description
D8 /2	FCOM *m32fp*	Compare ST(0) with *m32fp*.
DC /2	FCOM *m64fp*	Compare ST(0) with *m64fp*.
D8 D0+i	FCOM ST(i)	Compare ST(0) with ST(i).
D8 D1	FCOM	Compare ST(0) with ST(1).
D8 /3	FCOMP *m32fp*	Compare ST(0) with *m32fp* and pop register stack.
DC /3	FCOMP *m64fp*	Compare ST(0) with m64fp and pop register stack.
D8 D8+i	FCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Compare ST(0) with ST(1) and pop register stack twice.

Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that –0.0 ← +0.0.

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) ← SRC	1	0	0
Unordered*	1	1	1

NOTE:

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared (see “FXAM—Examine” in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to “unordered.” If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

8.1.2. x87 FPU Status Register

The 16-bit x87 FPU status register (see Figure 8-4) indicates the current state of the x87 FPU. The flags in the x87 FPU status register include the FPU busy flag, top-of-stack (TOP) pointer, condition code flags, error summary status flag, stack fault flag, and exception flags. The x87 FPU sets the flags in this register to show the results of operations.

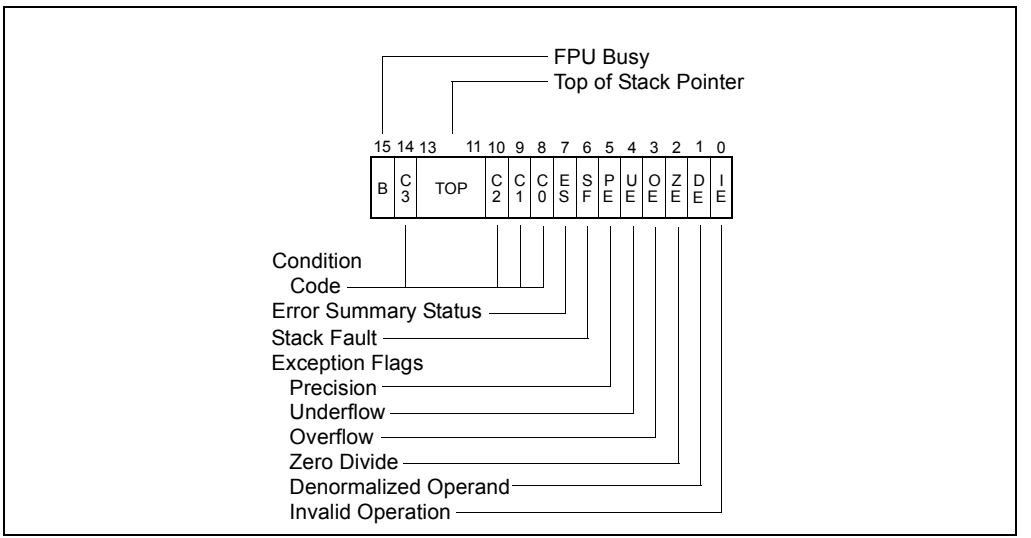

Figure 8-4. x87 FPU Status Word

The contents of the x87 FPU status register (referred to as the x87 FPU status word) can be stored in memory using the FSTSW/FNSTSW, FSTENV/FNSTENV, FSAVE/FNSAVE, and FXSAVE instructions. It can also be stored in the AX register of the integer unit, using the FSTSW/FNSTSW instructions.

TEST—Logical Compare

Opcode	Instruction	Description
A8 *ib*	TEST AL,*imm8*	AND *imm8* with AL; set SF, ZF, PF according to result
A9 *iw*	TEST AX,*imm16*	AND *imm16* with AX; set SF, ZF, PF according to result
A9 *id*	TEST EAX,*imm32*	AND *imm32* with EAX; set SF, ZF, PF according to result
F6 /0 *ib*	TEST *r/m8,imm8*	AND *imm8* with *r/m8*; set SF, ZF, PF according to result
F7 /0 *iw*	TEST *r/m16,imm16*	AND *imm16* with *r/m16*; set SF, ZF, PF according to result
F7 /0 *id*	TEST *r/m32,imm32*	AND *imm32* with *r/m32*; set SF, ZF, PF according to result
84 /*r*	TEST *r/m8,r8*	AND *r8* with *r/m8*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m16,r16*	AND *r16* with *r/m16*; set SF, ZF, PF according to result
85 /*r*	TEST *r/m32,r32*	AND *r32* with *r/m32*; set SF, ZF, PF according to result

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

Operation

```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);
IF TEMP ← 0
    THEN ZF ← 1;
    ELSE ZF ← 0;
FI:
PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(*AF is Undefined*)
```

Flags Affected

The OF and CF flags are cleared to 0. The SF, ZF, and PF flags are set according to the result (see the “Operation” section above). The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Table 8-8. TEST Instruction Constants for Conditional Branching

Order	Constant	Branch
ST(0) > Source Operand	4500H	JZ
ST(0) < Source Operand	0100H	JNZ
ST(0) = Source Operand	4000H	JNZ
Unordered	0400H	JNZ

```
; File: double5.asm
;
; Using C printf function to print double values
; Checking out comparisons
;
; Declare some external functions
;
        extern printf                   ; the C function, we'll call

        SECTION .data                   ; Data section
; Strings to print
msg1:   db "dv2 > dv1", 10, 0
msg2:   db "dv2 <= dv1", 10, 0

msg3:   db "dv3 < dv2", 10, 0
msg4:   db "dv3 >= dv2", 10, 0

msg5:   db "dv5 == dv2 + dv4", 10, 0
msg6:   db "dv5 != dv2 + dv4", 10, 0

dv1:    dq 1.111
dv2:    dq 2.222
dv3:    dq -3.333
dv4:    dq 4.444
dv5:    dq 5.555
dv6:    dq 6.666
dv7:    dq 7.777


        SECTION .text                   ; Code section.

        global main
main:
        push    ebp                     ; set up stack frame
        mov     ebp,esp

        fld     QWORD [dv1]
        fld     QWORD [dv2]
        fcompp                          ; compare then 2x pop
        fstsw   ax                      ; store status of comp

        test    ax, 4500H               ; logial AND
        jz      st0_gt_st1
        push    DWORD msg2
        call    printf
        add     esp, 4
        jmp     done1
st0_gt_st1:
        push    DWORD msg1
        call    printf
        add     esp, 4
done1:
```

```
        fld     QWORD [dv2]
        fld     QWORD [dv3]
        fcompp                          ; compare then 2x pop
        fstsw   ax                      ; store status of comp

        test    ax, 0100H               ; logial AND
        jnz     st0_lt_st1              ; note the 'n' in jnz
        push    DWORD msg4
        call    printf
        add     esp, 4
        jmp     done2
st0_lt_st1:
        push    DWORD msg3
        call    printf
        add     esp, 4
done2:

        fld     QWORD [dv2]
        fld     QWORD [dv4]
        faddp   st1, st0
        fld     QWORD [dv5]
        fcompp
        fstsw   ax

        test    ax, 4000H               ; logial AND
        jnz     st0_eq_st1              ; note the 'n' in jnz
        push    DWORD msg6
        call    printf
        add     esp, 4
        jmp     done3
st0_eq_st1:
        push    DWORD msg5
        call    printf
        add     esp, 4
done3:


        ; return from main
        mov     esp, ebp                ; takedown stack frame
        pop     ebp                     ;   same as "leave" op

        ret
```

```
linuxserver2% nasm -f elf double5.asm
linuxserver2% gcc double5.o

linuxserver2% ./a.out
dv2 > dv1
dv3 < dv2
dv5 != dv2 + dv4
```