the MMX registers are aliased to the x87 FPU data registers. Therefore, when writing code that mixed x87 FPU and MMX instructions, the programmer must explicitly manage the x87 FPU and MMX state (see Section 9.5., “Compatibility with x87 FPU Architecture”).

8.1.1. x87 FPU Data Registers

The x87 FPU data registers (shown in Figure 8-1) consist of eight 80-bit registers. Values are stored in these registers in the double extended-precision floating-point format shown in Figure 4-3. When floating-point, integer, or packed BCD integer values are loaded from memory into any of the x87 FPU data registers, the values are automatically converted into double extended-precision floating-point format (if they are not already in that format). When computation results are subsequently transferred back into memory from any of the x87 FPU registers, the results can be left in the double extended-precision floating-point format or converted back into a shorter floating-point format, an integer format, or the packed BCD integer format. (See Section 8.2., “x87 FPU Data Types” for a description of the data types operated on by the x87 FPU.)

The x87 FPU instructions treat the eight x87 FPU data registers as a register stack (see Figure 8-2). All addressing of the data registers is relative to the register on the top of the stack. The register number of the current top-of-stack register is stored in the TOP (stack TOP) field in the x87 FPU status word. Load operations decrement TOP by one and load a value into the new top-of-stack register, and store operations store the value from the current TOP register in memory and then increment TOP by one. (For the x87 FPU, a load operation is equivalent to a push and
a store operation is equivalent to a pop.) Note that load and store operations are also available that do not push and pop the stack.

If a load operation is performed when TOP is at 0, register wraparound occurs and the new value of TOP is set to 7. The floating-point stack-overflow exception indicates when wraparound might cause an unsaved value to be overwritten (see Section 8.5.1.1., “Stack Overflow or Underflow Exception (#IS)").

Many floating-point instructions have several addressing modes that permit the programmer to implicitly operate on the top of the stack, or to explicitly operate on specific registers relative to the TOP. Assemblers supports these register addressing modes, using the expression ST(0), or simply ST, to represent the current stack top and ST(i) to specify the i\textsuperscript{th} register from TOP in the stack (0 \leq i \leq 7). For example, if TOP contains 011B (register 3 is the top of the stack), the following instruction would add the contents of two registers in the stack (registers 3 and 5):

\[
\text{FADD ST, ST(2)};
\]

Figure 8-3 shows an example of how the stack structure of the x87 FPU registers and instructions are typically used to perform a series of computations. Here, a two-dimensional dot product is computed, as follows:

1. The first instruction (FLD value1) decrements the stack register pointer (TOP) and loads the value 5.6 from memory into ST(0). The result of this operation is shown in snap-shot (a).
2. The second instruction multiplies the value in ST(0) by the value 2.4 from memory and stores the result in ST(0), shown in snap-shot (b).
3. The third instruction decrements TOP and loads the value 3.8 in ST(0).  
4. The fourth instruction multiplies the value in ST(0) by the value 10.3 from memory and stores the result in ST(0), shown in snap-shot (c).
5. The fifth instruction adds the value and the value in ST(1) and stores the result in ST(0), shown in snap-shot (d).
The style of programming demonstrated in this example is supported by the floating-point instruction set. In cases where the stack structure causes computation bottlenecks, the FXCH (exchange x87 FPU register contents) instruction can be used to streamline a computation.

### 8.1.1.1. **PARAMETER PASSING WITH THE X87 FPU REGISTER STACK**

Like the general-purpose registers, the contents of the x87 FPU data registers are unaffected by procedure calls, or in other words, the values are maintained across procedure boundaries. A calling procedure can thus use the x87 FPU data registers (as well as the procedure stack) for passing parameter between procedures. The called procedure can reference parameters passed through the register stack using the current stack register pointer (TOP) and the ST(0) and ST(i) nomenclature. It is also common practice for a called procedure to leave a return value or result in register ST(0) when returning execution to the calling procedure or program.

When mixing MMX and x87 FPU instructions in the procedures or code sequences, the programmer is responsible for maintaining the integrity of parameters being passed in the x87 FPU data registers. If an MMX instruction is executed before the parameters in the x87 FPU data registers have been passed to another procedure, the parameters may be lost (see Section 9.5., “Compatibility with x87 FPU Architecture”).

![Figure 8-3. Example x87 FPU Dot Product Computation](image-url)
Like the CMOVcc instructions, the FCMOVcc instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

Software can check if the FCMOVcc instructions are supported by checking the processor’s feature information with the CPUID instruction (see “CPUID—CPU Identification” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2).

### 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 \(\log_e{2}\)
- **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

### 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:

\[ \text{FADD ST}(0), \text{ST}(0) ; \]

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 operands are infinities of the same sign, the result is \(\infty\) of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated.
FADD/FADDP/FIADD—Add (Continued)

### Operation

IF instruction is FIADD

THEN

\[
\text{DEST} \leftarrow \text{DEST} + \text{ConvertToDoubleExtendedPrecisionFP}(\text{SRC});
\]

ELSE (* source operand is floating-point value *)

\[
\text{DEST} \leftarrow \text{DEST} + \text{SRC};
\]

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.

### Notes:

- F Means finite floating-point value.
- I Means integer.
- * Indicates floating-point invalid-arithmetic-operand (#IA) exception.
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
; 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 ; address of ctrl string
push    DWORD msg
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]
fsinp 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D8 /2</td>
<td>FCOM m32fp</td>
<td>Compare ST(0) with m32fp.</td>
</tr>
<tr>
<td>DC /2</td>
<td>FCOM m64fp</td>
<td>Compare ST(0) with m64fp.</td>
</tr>
<tr>
<td>D8 D0+i</td>
<td>FCOM ST(i)</td>
<td>Compare ST(0) with ST(i).</td>
</tr>
<tr>
<td>D8 D1</td>
<td>FCOM</td>
<td>Compare ST(0) with ST(1).</td>
</tr>
<tr>
<td>D8 /3</td>
<td>FCOMP m32fp</td>
<td>Compare ST(0) with m32fp and pop register stack.</td>
</tr>
<tr>
<td>DC /3</td>
<td>FCOMP m64fp</td>
<td>Compare ST(0) with m64fp and pop register stack.</td>
</tr>
<tr>
<td>D8 D8+i</td>
<td>FCOMP ST(i)</td>
<td>Compare ST(0) with ST(i) and pop register stack.</td>
</tr>
<tr>
<td>D8 D9</td>
<td>FCOMP</td>
<td>Compare ST(0) with ST(1) and pop register stack.</td>
</tr>
<tr>
<td>DE D9</td>
<td>FCOMPP</td>
<td>Compare ST(0) with ST(1) and pop register stack twice.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Condition</th>
<th>C3</th>
<th>C2</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) &gt; SRC</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ST(0) &lt; SRC</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ST(0) == SRC</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unordered*</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**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](image)

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.

8.1.2.1. TOP OF STACK (TOP) POINTER

A pointer to the x87 FPU data register that is currently at the top of the x87 FPU register stack is contained in bits 11 through 13 of the x87 FPU status word. This pointer, which is commonly referred to as TOP (for top-of-stack), is a binary value from 0 to 7. See Section 8.1.1., “x87 FPU Data Registers”, for more information about the TOP pointer.

8.1.2.2. CONDITION CODE FLAGS

The four condition code flags (C0 through C3) indicate the results of floating-point comparison and arithmetic operations. Table 8-1 summarizes the manner in which the floating-point instructions set the condition code flags. These condition code bits are used principally for conditional branching and for storage of information used in exception handling (see Section 8.1.3., “Branching and Conditional Moves on Condition Codes”).
TEST—Logical Compare

### 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

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

### 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.
in the EFLAGS register if the condition code flags indicate an unordered result; otherwise, the ZF flag will be set. The JNZ instruction can then be used to transfer control (if necessary) to a procedure for handling unordered operands.

<table>
<thead>
<tr>
<th>Order</th>
<th>Constant</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST(0) &gt; Source Operand</td>
<td>4500H</td>
<td>JZ</td>
</tr>
<tr>
<td>ST(0) &lt; Source Operand</td>
<td>0100H</td>
<td>JNZ</td>
</tr>
<tr>
<td>ST(0) = Source Operand</td>
<td>4000H</td>
<td>JNZ</td>
</tr>
<tr>
<td>Unordered</td>
<td>0400H</td>
<td>JNZ</td>
</tr>
</tbody>
</table>

2. Check ordered comparison result. Use the constants given in Table 8-8 in the TEST instruction to test for a less than, equal to, or greater than result, then use the corresponding conditional branch instruction to transfer program control to the appropriate procedure or section of code.

If a program or procedure has been thoroughly tested and it incorporates periodic checks for QNaN results, then it is not necessary to check for the unordered result every time a comparison is made.

See Section 8.1.3., “Branching and Conditional Moves on Condition Codes”, for another technique for branching on x87 FPU condition codes.

Some non-comparison x87 FPU instructions update the condition code flags in the x87 FPU status word. To ensure that the status word is not altered inadvertently, store it immediately following a comparison operation.

### 8.3.7. Trigonometric Instructions

The following instructions perform four common trigonometric functions:

- `FSIN` Sine
- `FCOS` Cosine
- `FSINCOS` Sine and cosine
- `FPTAN` Tangent
- `FPATAN` Arctangent

These instructions operate on the top one or two registers of the x87 FPU register stack and they return their results to the stack. The source operands for the FSIN, FCOS, FSINCOS, and FPTAN instructions must be given in radians; the source operand for the FPATAN instruction is given in rectangular coordinate units.

The FSINCOS instruction returns both the sine and the cosine of a source operand value. It operates faster than executing the FSIN and FCOS instructions in succession.

The FPATAN instruction computes the arctangent of ST(1) divided by ST(0), returning a result in radians. It is useful for converting rectangular coordinates to polar coordinates.
; 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
mov    ebp,esp

fld    QWORD [dv1]
fld    QWORD [dv2]
fcompp
fstsw   ax

    test   ax, 4500H
    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  ; logical 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  ax
fstsw   ax  ; 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