Exception Synchronization

The passage below describes the potential problem of floating-point exceptions in concurrent systems and how to use the WAIT and FWAIT instructions to solve it.

In a concurrent system, two or more tasks can execute at the same time. This can be a problem for floating-point exceptions because the **FPU and the CPU are separate units.**

If an unmasked **floating-point exception occurs while the CPU is executing an integer or system instruction**, the exception will not be handled until the next floating-point instruction or the FWAIT instruction is executed.

This can lead to problems if the floating-point instruction that caused the exception is followed by an integer or system instruction that modifies the same memory operand as the floating-point instruction.

For example, the following code could cause a problem:

```
445 .data
446     intVal DWORD 25   ; Define an integer value
447
448 .code
449     fild intVal        ; Load the integer value into ST(0)
450     inc intVal         ; Increment the integer value
```

This code loads the integer value stored in intVal into the FPU stack register ST(0) and then increments the integer value by 1.

Please note that in this context, the inc instruction increments the integer value stored in memory at the address specified by intVal.

If an unmasked floating-point exception occurs while the FILD instruction is executing, the exception will not be handled until the INC instruction is executed.

If the INC instruction modifies the same memory operand as the FILD instruction, the exception handler will not be able to access the

correct value of the memory operand.

The WAIT and FWAIT instructions can be used to solve this problem.

Both instructions force the processor to check for pending, unmasked floating-point exceptions before proceeding to the next instruction.

This ensures that the exception handler will have a chance to execute before any integer or system instructions that modify the same memory operand as the floating-point instruction that caused the exception.

To solve the problem in the example code above, you could add a WAIT or FWAIT instruction after the FILD instruction. For example:

```
.data
    intVal DWORD 25

.code
    ; Load the integer into ST(0)
    fild intVal

    ; Wait for pending exceptions
    fwait

    ; Increment the integer
    inc intVal
```

In this version, the comments are placed next to the relevant instructions.

The FILD instruction loads the integer into the FPU stack, followed by the FWAIT instruction to ensure any pending exceptions are handled.

Then, the INC instruction increments the integer value in memory. This sequence of instructions allows for proper exception handling before modifying the shared memory operand.

This would ensure that the exception handler would have a chance to execute before the INC instruction is executed.

EXAMPLE CODE FOR FPU OPERATIONS:

Here's the code for the expression valD = -valA + (valB * valC)

```
.data
    valA REAL8 1.5
    valB REAL8 2.5
    valC REAL8 3.0
    valD REAL8 ? ; Initialize valD as a placeholder for the result

.code
    ; Load valA on the FPU stack and negate it
    fld valA        ; ST(0) = valA
    fchs            ; Negate the value in ST(0)

    ; Load valB into ST(0) and multiply by valC
    fld valB        ; Load valB into ST(0)
    fmul valC       ; Multiply ST(0) by valC, leaving the product in ST(0)

    ; Add the two values on the stack (ST(0) and ST(1))
    fadd            ; Add ST(0) and ST(1), result in ST(0)

    ; Store the result in valD
    fstp valD       ; Store the result in valD
```

In this code, we calculate valD as the result of the expression -valA + (valB * valC).

We start by loading valA onto the FPU stack and negating it using fchs.

Then, we load valB into ST(0) and multiply it by valC, resulting in the product in ST(0).

Finally, we add the two values on the stack and store the result in valD.

The fstp instruction pops the result from the stack and stores it in valD.

SUM OF ARRAY OF DOUBLE PRECISION NUMBERS

In the code below, you are calculating the sum of an array of double-

precision real numbers.

```
; Define the size of the array
ARRAY_SIZE = 20

.data
    sngArray REAL8 ARRAY_SIZE DUP(?)

.code
    mov esi, 0          ; Initialize array index
    fldz               ; Push 0.0 onto the FPU stack
    mov ecx, ARRAY_SIZE ; Set the loop counter to ARRAY_SIZE

    L1:
    fld sngArray[esi]  ; Load the current element into ST(0)
    fadd               ; Add ST(0) to the accumulator (ST(0)), and pop
    add esi, TYPE REAL8 ; Move to the next element
    loop L1            ; Continue the loop until all elements are processed

    call WriteFloat    ; Display the sum in ST(0)
```

In this code, we are calculating and displaying the sum of an array of double-precision real numbers.

The size of the array is defined by the symbolic constant ARRAY_SIZE, which has a value of 20.

This constant makes the code more maintainable, allowing us to change the array size in a single place if needed.

We initialize a loop counter esi to 0 and push 0.0 onto the FPU stack using fldz. This will be our accumulator for the sum of the array elements.

The loop counter ecx is set to the value of ARRAY_SIZE, which determines the number of elements in the array.

We enter a loop labeled as L1, where we perform the following operations for each element of the array:

Load the current array element into the FPU stack using fld sngArray[esi].

Add the value in the FPU stack to the accumulator (ST(0)) using fadd.

This operation also pops the top of the stack.

Move to the next element by adding the size of a REAL8 (8 bytes) to esi using add esi, TYPE REAL8.

The loop continues until all elements of the array have been processed.

After the loop, we call WriteFloat to display the sum of the array elements in ST(0).

CALCULATING THE SUM OF SQUARES OF TWO NUMBERS

In this code, we are calculating the sum of the square roots of two numbers, valA and valB. Here are the steps involved:

We define two real numbers, valA and valB, which have the values 25.0 and 36.0, respectively.

```
.data
    valA REAL8 25.0
    valB REAL8 36.0

.code
    fld valA    ; Load valA onto the FPU stack
    fsqrt       ; Replace ST(0) with the square root of valA
    fld valB    ; Load valB onto the FPU stack
    fsqrt       ; Replace ST(0) with the square root of valB
    fadd        ; Add the two square roots and leave the result in ST(0)
```

We use the FPU instructions to perform the calculation:

fld valA: This instruction loads the value of valA onto the FPU stack, making it the top element (ST(0)) on the stack.

fsqrt: This instruction replaces the value in ST(0) with its square root, so now ST(0) contains the square root of valA.

fld valB: We load the value of valB onto the FPU stack, replacing the previous value in ST(0).

fsqrt: This instruction calculates the square root of the new value

in ST(0), which is the square root of valB.

fadd: Finally, we add the two square roots together, and the result is left in ST(0). This means ST(0) now contains the sum of the square roots of valA and valB.

This code efficiently calculates the sum of the square roots of valA and valB, utilizing the FPU instructions to load, compute square roots, and add the results. The final sum is stored in ST(0).

CALCULATING THE DOT PRODUCT OF TWO PAIRS OF NUMBERS

Here, you have a code snippet that calculates the dot product of two pairs of numbers from an array. The input data contains two pairs of numbers stored in the "array" variable.

```
.data
    array REAL4 6.0, 2.0, 4.5, 3.2

.code
    fld dword ptr [array]     ; Load the first number of the first pair
    fmul dword ptr [array+4]  ; Multiply with the second number of the first pair
    fld dword ptr [array+8]   ; Load the first number of the second pair
    fmul dword ptr [array+12]  ; Multiply with the second number of the second pair
    fadd                      ; Add the two products in ST(0)
```

fld array: This instruction loads the first number from the "array" into ST(0), which is the top of the FPU stack.

fmul [array+4]: Here, we multiply the value at the memory location array+4 with the value in ST(0). This corresponds to multiplying the first pair (6.0 * 2.0) and leaving the result in ST(0).

fld [array+8]: We load the second number from the second pair (4.5) into ST(0).

fmul [array+12]: This instruction multiplies the value at memory location array+12 (which is 3.2) with the value in ST(0). This is equivalent to multiplying the second pair (4.5 * 3.2) and leaves the result in ST(0).

fadd: Finally, we add the two results in ST(0), which represents the dot product (6.0 * 2.0) + (4.5 * 3.2).