Shifting Multiple DoubleWords

==========

Shifting Multiple Doublewords

==========

In the realm of assembly programming, you can manipulate extended-precision integers that are organized into arrays of bytes, words, or doublewords.

However, it's imperative to understand how these array elements are stored.

A prevalent method of storing these integers is referred to as "little-endian order."

In little-endian order, the low-order byte is placed at the array's starting address, and then, as you progress from this byte to the high-order byte, each is consecutively stored in the next memory location.

This ordering holds true regardless of whether you're working with bytes, words, or doublewords because x86 machines consistently use little-endian order for all these data formats.

Now, let's delve into the specific steps for shifting an array of bytes one bit to the right:

Step 1:

To accomplish this operation, you start by shifting the highest byte located at [ESI+2] to the right.

During this shift, the lowest bit of this byte is automatically copied into the Carry flag.

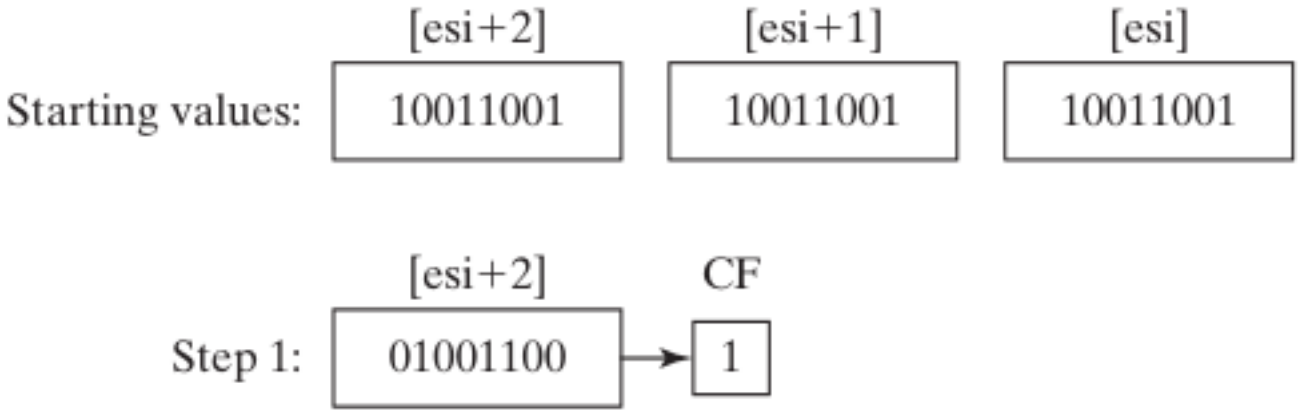
This behavior is a standard operation performed by instructions like SHR (Shift Right) in many assembly languages.

To visually demonstrate this, here's how you might express it in x86

assembly code, assuming that the ESI register holds the base address of the array:

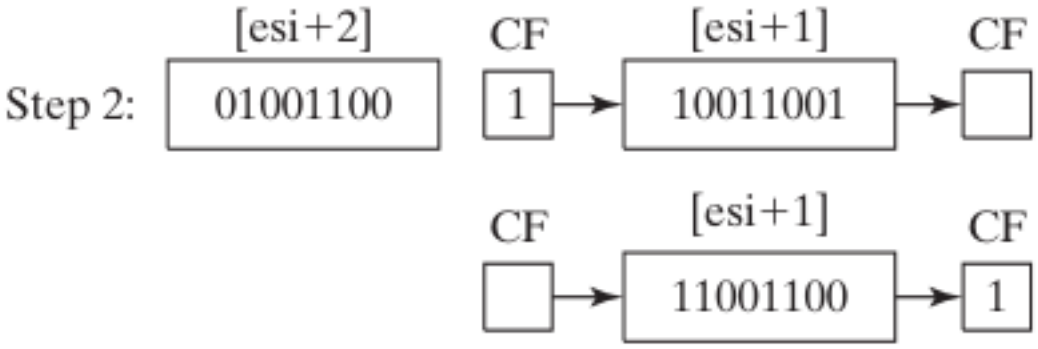
```
SHR byte ptr [ESI+2], 1
```

This instruction effectively shifts the byte at [ESI+2] one bit to the right, with the least significant bit being transferred to the Carry flag.



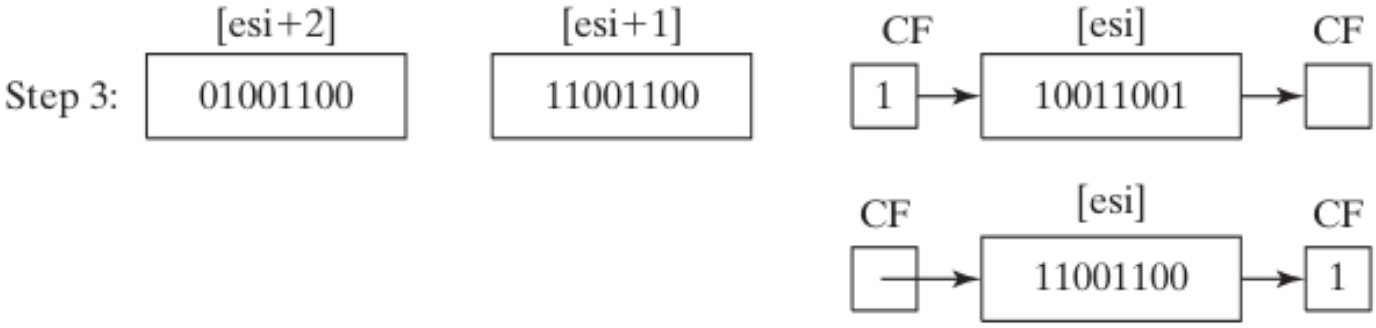
Step 2:

Rotate the value at [ESI+1] to the right, filling the highest bit with the value of the Carry flag, and shifting the lowest bit into the Carry flag:

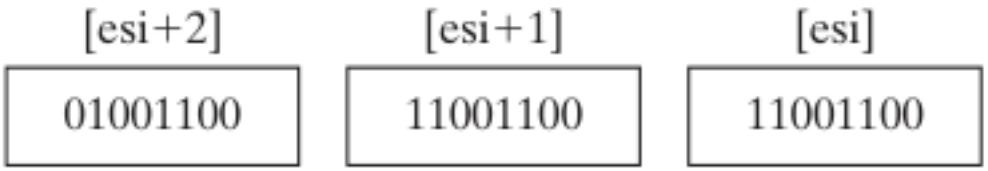


Step 3:

Rotate the value at [ESI] to the right, filling the highest bit with the value of the Carry flag, and shifting the lowest bit into the Carry flag:



After Step 3 is complete, all bits have been shifted 1 position to the right:




```
.data
ArraySize = 3
array BYTE ArraySize DUP(99h) ; Initialize an array of 3 bytes with the value 99h (1001 1001 in binary).

.code
main PROC
mov esi, 0        ; Initialize the ESI register to 0, pointing to the beginning of the array.
shr array[esi+2], 1  ; Shift the high byte at array[2] one bit to the right.
rcr array[esi+1], 1  ; Rotate the middle byte at array[1] one bit to the right, including the Carry flag.
rcr array[esi], 1    ; Rotate the low byte at array[0] one bit to the right, including the Carry flag.
```

Here's a breakdown of what's happening in this code:

The **.data section** defines the ArraySize as 3 and initializes an array called array with three bytes, each containing the value 99h. This means the binary representation of each byte is "1001 1001."

In the **.code section**, the main procedure begins.

mov esi, 0 initializes the ESI register to 0, pointing to the first byte in the array.

shr array[esi+2], 1 performs a right shift on the high byte at array[2]. This operation moves the most significant bit one position to the right, effectively shifting the entire byte to the right by one bit.

rcr array[esi+1], 1 rotates the middle byte at array[1] one bit to the right. A rotate operation combines shifting with a rotation of the Carry flag. It effectively moves the least significant bit of the high byte into the least significant bit of the middle byte, and the Carry flag into the most significant bit of the high byte.

Similarly, **rcr array[esi], 1** rotates the low byte at array[0], including the Carry flag. This rotates the bytes and carries the bit that was shifted out from the middle byte into the low byte.

As a result, this code has shifted the entire array of bytes by one bit to the right, and the Carry flag holds the value of the bit that was shifted out from the low byte.

The code can indeed be adapted to handle arrays of words or doublewords by changing the data types and adjusting the number of bytes processed. Additionally, by using a loop, you can shift arrays of arbitrary size efficiently.