Binary Shifting and Multiplication

Let's discuss the concept of binary multiplication in assembly language, specifically focusing on optimizing integer multiplication through bit shifting rather than using the MUL instruction.

Bit Shifting for Multiplication:

In assembly programming, the SHL (Shift Left) instruction is often used to perform unsigned multiplication when the multiplier is a power of 2.

Shifting an unsigned integer left by 'n' bits is equivalent to multiplying it by 2^n.

This is a highly efficient way to perform multiplication when the multiplier is a power of 2 because it can be achieved through simple bit shifting operations, which are faster than a full multiplication.

Representing Non-Power-of-2 Multipliers:

For multipliers that are not powers of 2, you can express them as a sum of powers of 2. This allows you to break down the multiplication into simpler bit-shifting operations.

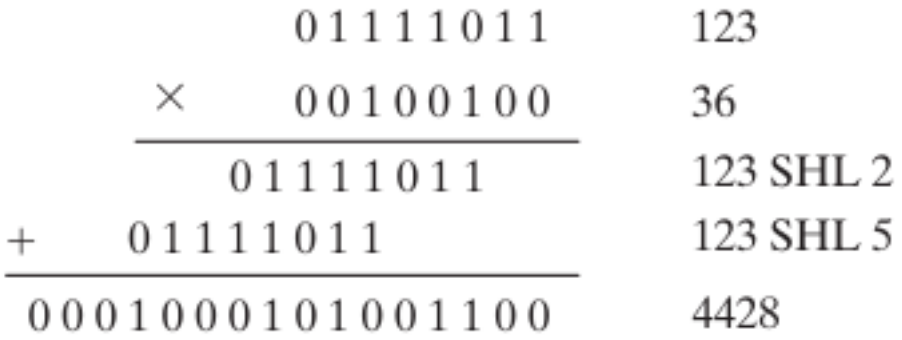
For example, to multiply a value in the EAX register by 36, you can represent 36 as $2^5 + 2^2$ and apply the distributive property of multiplication as follows:

```
142 EAX * 36 = EAX * (2^5 + 2^2)
143          = EAX * (32 + 4)
144          = (EAX * 32) + (EAX * 4)
```

This decomposition enables you to perform two separate bit-shifting operations: one to multiply by 32 (shifting left by 5 bits) and another to multiply by 4 (shifting left by 2 bits). These individual results can then be added to obtain the final product.

*Example: 123 * 36:*

The passage mentions an example of multiplying 123 by 36, resulting in 4428. This multiplication can be achieved using the approach described above. In this case, you would perform two separate bit-shifting operations.




```
146 123 * 32 (2^5) = 3936
147 123 * 4 (2^2) = 492
```

```
151 mov     eax, 123
152 mov     ebx, eax
153 shl     eax, 5      ; multiply by 2^5
154 shl     ebx, 2      ; multiply by 2^2
155 add     eax, ebx
```

This code snippet first moves the value 123 into the register EAX.

Then, it moves a copy of EAX into the register EBX.

Next, it shifts EAX left by 5 bits, which is equivalent to multiplying it by 25.

It then shifts EBX left by 2 bits, which is equivalent to multiplying it by 22.

Finally, it adds the two products together in the register EAX.

To generalize this example and create a procedure that multiplies any two 32-bit unsigned integers using shifting and addition, we can use the following algorithm:

the following algorithm:

- **Initialize the product register to 0.**
- **For each bit in the multiplier, starting with the least significant bit:**
- **If the bit is set, shift the multiplicand left by the corresponding number of bits and add it to the product register.**
- **If the bit is not set, do nothing.**
- **Return the product register.**

The following pseudocode shows this algorithm:

```
#include <stdio.h>

int multiply(int multiplicand, int multiplier) {
    int product = 0;

    for (int i = 0; i < 32; i++) {
        if ((multiplier & (1 << i)) != 0) {
            product += (multiplicand << i);
        }
    }

    return product;
}

int main() {
    int multiplicand = 123;  // Replace with your values
    int multiplier = 36;    // Replace with your values

    int result = multiply(multiplicand, multiplier);

    printf("Result: %d\n", result);
    return 0;
}
```

We define a function multiply that takes two integers, multiplicand and multiplier, as parameters and returns the product as an integer.

Inside the function, we initialize product to 0, which will hold the result.

We use a for loop to iterate from 0 to 31 to examine each bit of the multiplier.

Within the loop, we use bitwise operations to check if the i-th bit of the multiplier is set (1).

If it is, we shift the multiplicand left by i bits and add the result to the product.

Finally, the function returns the computed product.

In the main function, you can replace the values of multiplicand and multiplier with the numbers you want to multiply. When you run the program, it will calculate the product and print the result.

The following assembly code implements this algorithm:

```
183 multiply:
184   push    ebp
185   mov     ebp, esp
186   mov     eax, multiplicand
187   mov     ebx, multiplier
188   xor     ecx, ecx
189   mov     ecx, 31
190 loop:
191   shl     eax, 1
192   test    ebx, 1
193   jz      next
194   add     eax, ebx
195 next:
196   dec     ecx
197   jnz     loop
198   mov     esp, ebp
199   pop     ebp
200   ret
```

This assembly code appears to multiply two integers, multiplicand and multiplier, and the result is stored in the EAX register, which is a common convention for returning values in assembly language functions. Let's break down the code:

push ebp: This instruction saves the base pointer (EBP) on the stack to establish a new stack frame. This is a common practice at the beginning of a function.

mov ebp, esp: It sets the base pointer (EBP) to the current stack pointer (ESP). This establishes a new stack frame for the function.

mov eax, multiplicand: This instruction loads the value of multiplicand into the EAX register.

mov ebx, multiplier: It loads the value of multiplier into the EBX register.

xor ecx, ecx: This clears the ECX register to zero. It will be used as a loop counter.

mov ecx, 31: This sets ECX to 31, which is the loop iteration count. The loop will iterate for 32 bits (0 to 31).

loop:: This is a label for the loop.

shl eax, 1: It shifts the EAX register left by 1 bit, effectively multiplying it by 2. This is performed in each iteration to handle the next bit of the multiplier.

test ebx, 1: This instruction tests the least significant bit of EBX (the multiplier) to check if it's set (1).

jz next: If the least significant bit of the multiplier is not set (i.e., it's zero), the code jumps to the next label without adding EBX to the result in EAX.

add eax, ebx: If the least significant bit of the multiplier is set, it adds EBX to EAX, effectively performing an addition in each iteration.

next:: This label is used to continue with the next iteration of the loop.

dec ecx: It decrements the loop counter in ECX.

jnz loop: This instruction checks if ECX is not zero (meaning there are more bits to process in the multiplier). If it's not zero, the code jumps back to the loop label, continuing the multiplication process.

mov esp, ebp: This restores the stack pointer (ESP) to its previous value, effectively cleaning up the stack frame.

pop ebp: It restores the base pointer (EBP) to its previous value, completing the stack frame cleanup.

ret: This is the return instruction, and it returns the result in the EAX register.

Overall, the code is an assembly implementation of integer multiplication using bit-shifting and addition, and it follows a loop-based approach to handle each bit of the multiplier.