Chapter 4

BIT MANIPULATIONS

MULTIPLICATION ALGORITHM

Multiplication is like a familiar school process, but we'll break it down into simple steps to create an algorithm for a computer.

Imagine you have two 4-bit binary numbers: 1101 (13 in decimal) and 0101 (5 in decimal). We want to find their product, which should be 65 in decimal or 01000001 in binary.

Here's a step-by-step algorithm:

1. Start with the rightmost digit of the multiplier (0101). If it's 0, the answer is 0; if it's 1, the answer is the multiplicand (1101).

```
  1101
  0101
 -----
  1101
```

2. Move to the next digit in the multiplier (1). If it's 0, the answer is 0000; if it's 1, the answer is the multiplicand shifted one place to the left.

```
  1101
  0101
 -----
  1101
 0000x
```

3. Now, look at the next digit (also 1). The answer is the multiplicand shifted two places to the left.

```
  1101
  0101
 -----
  1101
 0000x
1101xx
```

4. For the last digit (0), the answer is 0000 shifted three places to the left.

```
   1101
   0101
  -----
   1101
```

```
  0000x
 1101xx
0000xxx
```

5. Add all these intermediate answers together (treating the crosses as zeros), and you get 01000001, which is 65.

This algorithm shows how binary multiplication can be done without the need for complicated multiplication operations. You just need to shift numbers and add them together. It's efficient and follows the simple rules of binary arithmetic.

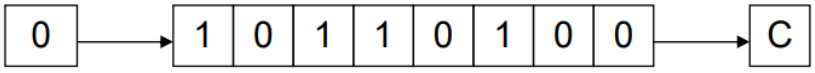
SHIFTING AND ROTATIONS

In computer architecture and assembly language programming, shifting and rotation instructions are essential tools that simplify complex tasks. These operations involve moving bits within binary numbers.

Here are some important shifting and rotation operations:

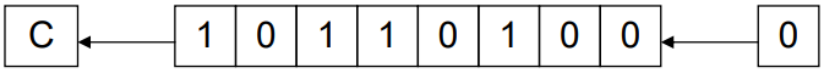
Shift Logical Right (SHR):

- Shift Logical Right inserts a zero from the left and moves every bit one position to the right.
- The rightmost bit is copied into the carry flag.



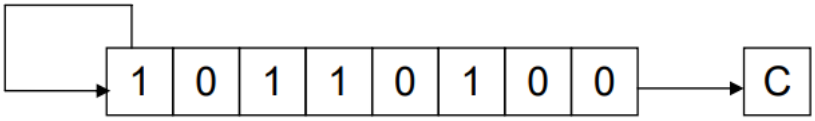
Shift Logical Left (SHL) / Shift Arithmetic Left (SAL):

- Shift Logical Left is the opposite of Shift Logical Right. It inserts a zero from the right and moves every bit one position to the left.
- The most significant bit drops into the carry flag.
- Shift Arithmetic Left is another name for Shift Logical Left.



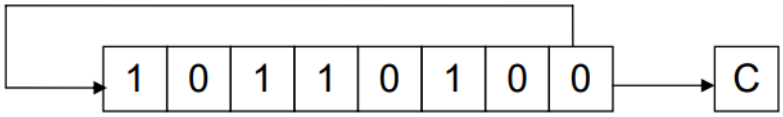
Shift Arithmetic Right (SAR):

- A signed number's most significant bit represents its sign.
- Logical right shifting could change the sign because it inserts a zero from the left.
- Shift Arithmetic Right ensures that the sign bit is retained.
- It shifts every bit to the right, copying the most significant bit to the left, and the bit dropped from the right goes into the carry flag.



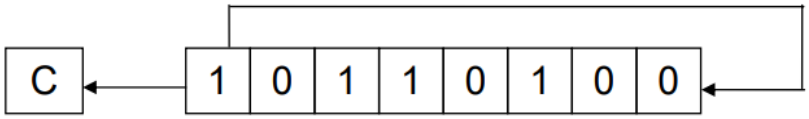
Rotate Right (ROR):

- Rotate Right moves every bit one position to the right, and the rightmost bit enters from the left.
- This bit is also copied into the carry flag.



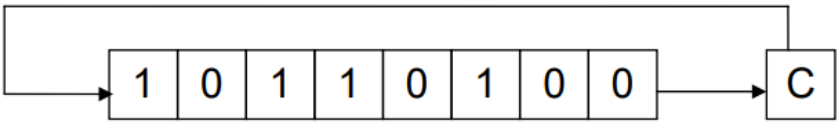
Rotate Left (ROL):

- Rotate Left is the reverse of Rotate Right.
- The most significant bit is copied to the carry flag and inserted from the right, causing every bit to move one position to the left.



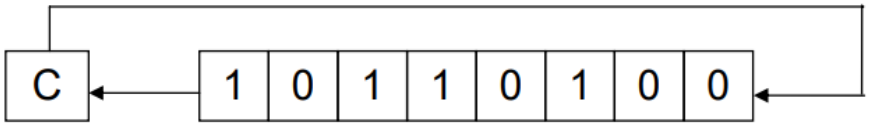
Rotate Through Carry Right (RCR):

- In Rotate Through Carry Right, the carry flag is inserted from the left.
- Every bit moves one position to the right, and the rightmost bit is dropped into the carry flag.
- This operation effectively performs a nine-bit or seventeen-bit rotation instead of the usual eight or sixteen bits.



Rotate Through Carry Left (RCL):

- Rotate Through Carry Left is the reverse of Rotate Through Carry Right.
- The carry flag is inserted from the right, causing every bit to move one position to the left, and the most significant bit occupies the carry flag.



NOTE: The left shifting operation is basically multiplication by **2** while the right shifting operation is division by **2**. However for signed numbers division by two can be accomplished by using shift arithmetic right and not shift logical right.

MULTIPLICATION IN ASSEMBLY LANGUAGE

Multiplication is a fundamental mathematical operation, and when we perform it in assembly language, we use a specialized algorithm that involves bit manipulation. Let's break down this multiplication algorithm into simpler steps:

1. **Set Up:** First, we initialize our result to zero.
2. **Iterate Through Bits:** We need to consider each bit in the multiplier (the number we're multiplying by). We start with the rightmost bit and move left. In our example, we have a 4-bit multiplier, so we iterate four times.
3. **Check the Bit:** For each bit in the multiplier, we check if it's a 1 or 0. We do this by performing a right shift operation on the multiplier. If the rightmost bit becomes 1, we set a carry flag; otherwise, the carry flag remains clear.
4. **Add to Result (Conditional):** If the carry flag is set (indicating the current multiplier bit is 1), we add the multiplicand (the number we're multiplying) to the result. Otherwise, we skip this addition step.
5. **Shift Multiplicand:** Regardless of whether we added the multiplicand or not, we perform a left shift on the multiplicand. This prepares it for the next bit of the multiplier.
6. **Repeat:** We decrement a bit count (initially set to the number of bits in the multiplier), and if there are bits left to check, we repeat the process from step 3.
7. **Terminate:** Finally, after all iterations, we have our result.

The example provided in assembly language code demonstrates a 4-bit multiplication. The multiplicand and multiplier are constants, and the result is an 8-bit register because eight bits can fit in a byte. If you were performing multiplication with larger numbers, you'd adjust the size of the result accordingly.

This multiplication algorithm is scalable, meaning you can use it for different-sized operands by adjusting the number of iterations and the size of the result. It relies on shifting and carry flags to check and accumulate the result, making it a fundamental technique in assembly language programming.

```
[org 0x100]

jmp start

multiplicand: db 13     ; 4-bit multiplicand (8-bit space)
multiplier: db 5       ; 4-bit multiplier
result: db 0           ; 8-bit result
```

```
start:
mov cl, 4              ; Initialize bit count to four
mov bl, [multiplicand] ; Load multiplicand in bl
mov dl, [multiplier]   ; Load multiplier in dl

checkbit:
shr dl, 1              ; Move rightmost bit in carry
jnc skip               ; Skip addition if bit is zero
add [result], bl       ; Accumulate result

skip:
shl bl, 1              ; Shift multiplicand left
dec cl                 ; Decrement bit count
jnz checkbit           ; Repeat if bits left

mov ax, 0x4c00         ; Terminate program
int 0x21
```

This code implements the multiplication algorithm for 4-bit numbers as previously explained. It uses `shr` to shift bits right, `shl` to shift bits left, and conditional jumps (`jnc` and `jnz`) to check and accumulate the result. The final result is stored in the `result` variable.

EXTENDED OPERATIONS

In computer programming, it's crucial to be able to work with larger numbers, especially for tasks like multiplication of large integers. The native word size of a processor might not be sufficient for these operations. However, with clever use of available instructions, we can extend the capability of our processor.

EXTENDED SHIFTING

Extended shifting allows us to shift a larger number (e.g., 32 bits) in memory by shifting it word by word (16 bits at a time). This is necessary when we don't want to lose significant bits during a shift. For example, if we shift a 16-bit number left by one position, the leftmost bit might be lost.

Left Extended Shifting:

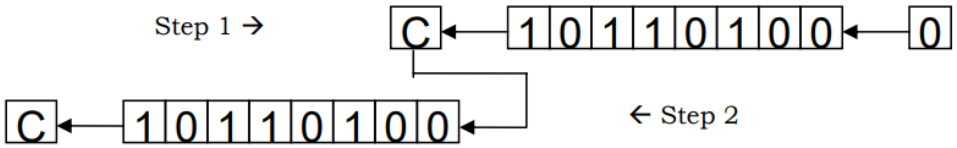
In left extended shifting, we shift a 32-bit number to the left, preserving all its bits. Here's how it's done:

```
num1: dd 40000  ; 32-bit space where the number is stored

shl word [num1], 1    ; Shift the lower 16 bits left by one position
rcl word [num1+2], 1  ; Rotate through carry left to shift the upper 16 bits
```

- The `dd` directive reserves a 32-bit space in memory, but we store a 16-bit value in it initially.
- We use two instructions to perform the left extended shift. The first instruction (`shl`) shifts the lower 16 bits to the left by one position, and the most significant bit of that word is dropped into the carry flag.

- The second instruction (`rcl`) rotates through carry left, effectively shifting the upper 16 bits. The bit saved in the carry is pushed into the least significant bit of the next word (the lower half), which joins the two 16-bit words.
- After these instructions, the final carry will always be zero for this example.



Right Extended Shifting:

Right extended shifting is similar but in the opposite direction. It allows us to perform right shifts on a 32-bit number. Here's how it works:

```
num1: dd 40000  ; 32-bit space where the number is stored

shr word [num1+2], 1  ; Shift the upper 16 bits right by one position
rcr word [num1], 1    ; Rotate through carry right to shift the lower 16 bits
```

- In right extended shifting, we shift the upper 16 bits to the right and then rotate through carry right to shift the lower 16 bits.
- These instructions ensure that the most significant bit (carry) of the upper 16 bits is transferred to the least significant bit of the lower 16 bits, preserving all bits in the process.

NOTE: Extended shifting is not limited to 32 bits; it can be applied to any number of bits, even up to 1024 bits or more. You would simply repeat the second instruction (rotate through carry) as needed to achieve the desired effect.

EXTENDED ADDITION AND SUBTRACTION

Extended addition and subtraction are techniques that allow us to perform arithmetic operations on larger numbers, such as 32-bit numbers, even when the processor's native word size is limited. In these operations, we utilize the `ADC` **(Add with Carry)** and `SBB` **(Subtract with Borrow)** instructions, which take into account the carry flag to handle larger data.

ADC (Add with Carry):

- In normal addition, only the destination and source operands are added together. However, with `ADC`, the carry flag is first added to the destination operand and then the source operand is added to the result.
- This allows for the handling of carry from previous additions.

Suppose we want to add two 32-bit numbers and store the result in the destination memory. The algorithm would look like this:

```
dest: dd 40000  ; 32-bit destination
src: dd 80000   ; 32-bit source
```

```
mov ax, [src]     ; Load the lower 16 bits of the source into AX
add word [dest], ax  ; Add lower 16 bits to the destination
mov ax, [src+2]   ; Load the upper 16 bits of the source into AX
adc word [dest+2], ax  ; Add upper 16 bits to the destination with carry
```

SBB (Subtract with Borrow):

- In normal subtraction, only the destination and source operands are subtracted. However, with `SBB`, if there was a borrow from a previous subtraction (carry flag set), it is first subtracted from the destination operand, and then the source operand is subtracted from the result.
- This allows for the handling of borrows from previous subtractions.

Algorithm for Extended Subtraction:

If we want to subtract one 32-bit number from another and store the result in the destination memory, the algorithm would be similar to extended addition:

```
dest: dd 40000  ; 32-bit destination
src: dd 80000   ; 32-bit source

mov ax, [src]     ; Load the lower 16 bits of the source into AX
sub word [dest], ax  ; Subtract lower 16 bits from the destination
mov ax, [src+2]   ; Load the upper 16 bits of the source into AX
sbb word [dest+2], ax  ; Subtract upper 16 bits from the destination with borrow
```

EXTENDED MULTIPLICATION

To perform extended multiplication, where the result might be larger than the word size, we combine extended shifting and addition. Here's an example of multiplying two 16-bit numbers to produce a 32-bit result:

```
[org 0x0100]
jmp start

multiplicand: dd 1300  ; 16-bit multiplicand stored in a 32-bit space
multiplier: dw 500     ; 16-bit multiplier
result: dd 0           ; 32-bit result

start:
mov cl, 16             ; Initialize bit count to 16
mov dx, [multiplier]   ; Load multiplier into dx

checkbit:
shr dx, 1              ; Move rightmost bit in carry
jnc skip               ; Skip addition if bit is zero
mov ax, [multiplicand] ; Load multiplicand's lower 16 bits
add [result], ax       ; Add lower bits to result
```

```
mov ax, [multiplicand+2] ; Load multiplicand's upper 16 bits
adc [result+2], ax     ; Add upper bits to result along with carry

skip:
shl word [multiplicand], 1
rcl word [multiplicand+2], 1 ; Shift multiplicand left
dec cl                  ; Decrement bit count
jnz checkbit            ; Repeat if bits left

mov ax, 0x4c00          ; Terminate program
int 0x21
```

In this code, the multiplicand and result are stored in 32-bit spaces, while the multiplier is a 16-bit word. Extended shifting and addition operations are used to correctly accumulate the result.

BITWISE LOGICAL OPERATIONS

In computer programming, bitwise logical operations are fundamental operations that work at the level of individual bits within binary numbers. These operations include:

AND Operation:

- **AND** performs a logical bitwise AND operation between two operands (usually bytes or words) and stores the result in the destination operand.
- The result bit is set if and only if both corresponding bits in the original operands are set to 1. Otherwise, the result bit is cleared.
- For example, **"and ax, bx"** performs a bitwise AND operation between the values in registers AX and BX, setting each bit in AX only if the corresponding bits in both AX and BX are 1.

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Operation:

- **OR** performs a logical bitwise OR operation between two operands (bytes or words) and stores the result in the destination operand.
- The result bit is set if either or both corresponding bits in the original operands are set to 1. If both bits are 0, the result bit is cleared.
- For instance, **"or ax, bx"** performs a bitwise OR operation between the values in registers AX and BX, setting each bit in AX if either the corresponding bit in AX or BX is 1.

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR Operation (Exclusive OR):

- **XOR** performs a logical bitwise exclusive OR operation between two operands (bytes or words) and stores the result in the destination operand.
- The result bit is set if the corresponding bits in the original operands have opposite values (one is set, the other is cleared). Otherwise, the result bit is cleared.
- For example, **"xor byte [mem], 5"** performs a bitwise XOR operation between the values.

X	Y	X xor Y
0	0	0
0	1	1
1	0	1
1	1	0

NOT Operation:

- **NOT** inverts or complements the bits of a single operand (byte or word). It doesn't require two operands like the other operations.
- For each bit in the operand, if it's 0, NOT changes it to 1, and if it's 1, NOT changes it to 0.
- For instance, **"not ax"** negates all the bits in register AX.

MASKING OPERATIONS

Masking operations are a common use case for bitwise logical operations like AND, OR, and XOR. They involve selectively modifying specific bits in a binary value while leaving others unchanged. Here are some key masking operations:

Selective Bit Clearing (Using AND):

- The AND operation can be used to selectively clear (set to 0) certain bits in a destination operand.
- This is done by using a source operand that contains a mask where the bits to be cleared are set to 0, and the bits to be retained retain their old values.

```
Original Value: 10111010
Mask:           11100111
AND
Result:         10100010
```

Selective Bit Setting (Using OR):

- The OR operation can be used for selective bit setting (setting specific bits to 1) in a destination operand.
- A source operand containing a mask is used, where the bits to be set are set to 1, and the bits to be left unchanged are set to 0.

```
Original Value: 10111010
Mask:           00000100
OR
Result:         10111110
```

Selective Bit Inversion (Using XOR):

- XOR (Exclusive OR) can be used as a masking operation to invert specific bits.
- A source operand with a mask is used, where the bits to be inverted are set to 1, and the bits to be left unchanged are set to 0.
- Unlike NOT, which inverts all bits, XOR allows selective bit inversion.

```
Original Value: 10111010
Mask:           00100000
XOR
Result:         10011010
```

TEST Operator

The `TEST` operator in assembly language is used to perform a bitwise AND operation between two values without storing the result. It's often used for bit testing to check if specific bits are set or cleared. If the result of the `TEST` operation is zero, it indicates that the tested bits were cleared in the value.

Here's a simple example:

```
[org 0x0100]
 jmp start
multiplicand: dd 1300    ; 16bit multiplicand 32bit space
multiplier: dw 500       ; 16bit multiplier
result: dd 0             ; 32bit result
start:
 mov cl, 16        ; initialize bit count to 16
 mov bx, 1         ; initialize bit mask
checkbit:
 test bx, [multiplier]   ; test right most bit
 jz skip                 ; skip addition if bit is zero
 mov ax, [multiplicand]
 add [result], ax        ; add less significant word
 mov ax, [multiplicand+2]
 adc [result+2], ax      ; add more significant word
```

```
skip:
 shl word [multiplicand], 1
 rcl word [multiplicand+2], 1   ; shift multiplicand left
 shl bx, 1                      ; shift mask towards next bit
 dec cl                         ; decrement bit count
 jnz checkbit                   ; repeat if bits left

 mov ax, 0x4c00 ; terminate program
 int 0x21
```

This way, the `TEST` instruction helps us efficiently check specific bits within a value without modifying the original value.

Code: Multiply two 32 bits numbers

```
[org 0x0100]

jmp Start

multiplicand_low: dd 0x00001300
multiplicand_high: dd 0x00000000
multiplier_low: dw 0x0500
multiplier_high: dw 0x0000
result_low: dd 0x00000000
result_high: dd 0x00000000

Start:
mov cx, 32
mov ax, [multiplier_high]
mov bx, [multiplier_low]
mov dx, 0

checkBit:
shr ax, 1
rcr bx, 1
jnc skip

mov ax, [multiplicand_low]
add [result_low], ax
adc dx, 0

mov ax, [multiplicand_low + 2]
adc [result_low + 2], ax

mov ax, [multiplicand_high]
```

```
adc [result_high], ax

mov ax, [multiplicand_high + 2]
adc [result_high + 2], ax

skip:
shl dword [multiplicand_low], 1
rcl dword [multiplicand_high], 1

dec cx
jnz checkBit

mov ax, 0x4c00
int 0x21
```