SHIFTS AND ROTATES

Shifts and rotates are fundamental operations in low-level programming and are incredibly important for anyone delving into reverse engineering or malware analysis.

They're the silent workhorses behind many **optimizations**, **obfuscations**, and even **cryptographic routines.** Let's break them down in detail.

🌀 Shifts and Rotates: Manipulating Bits with Precision

When you're working in assembly or low-level C, you **don't loop to multiply, you shift.**

You *don't call fancy functions to rotate bits*, you **ROL/ROR** like a digital warrior.
Let's break it down crystal clear:

At their core, shifts and rotates are operations that directly manipulate the individual bits within a binary value.

Think of them as **specialized tools** for rearranging the 0s and 1s that make up your data.

Understanding how they work is like **learning the secret language of processors** and the clever tricks malware authors use.

1. Shift Operations: Moving Bits In and Out

A **shift operation** involves moving all the bits of a binary value either to the left or to the right by a specified number of positions. Bits that move off one end are discarded, and new bits are introduced on the other end.

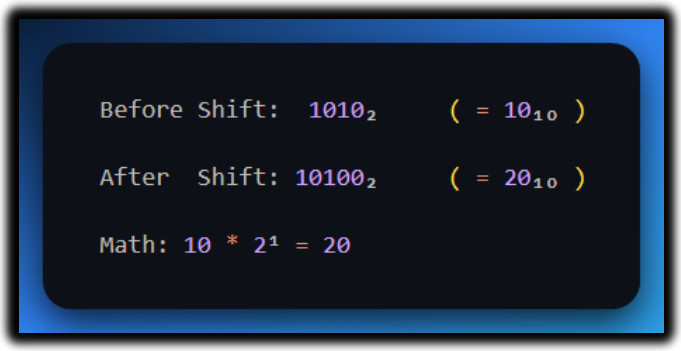
1.1. Left Shift (<<)

Concept: Imagine a line of people (bits). When you left-shift, everyone takes a step to their left. The person at the far left (most significant bit) falls off the line, and a new person (a 0) joins at the far right (least significant bit).

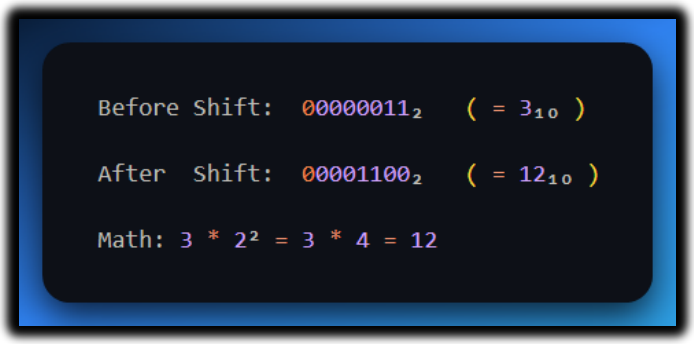
How it Works: Bits move to the left. The leftmost bit(s) are discarded. The rightmost bit(s) are filled with zeros.

Effect on Value: Shifting a binary value left by N positions is equivalent to **multiplying** that value by 2N.

Example 1: Shift 1010_2 (decimal 10) left by 1 position.



Example 2: Shift 00000011_2 (decimal 3) left by 2 positions.



😳 But wait... Why Didn't Bits Fall Off?

You asked the golden question. Let's break it down:

So, do bits fall off on a left shift?

✅ **Yes**, *if* you shift a number beyond the size limit of the register (e.g., 8-bit, 16-bit, 32-bit).

But in your example:

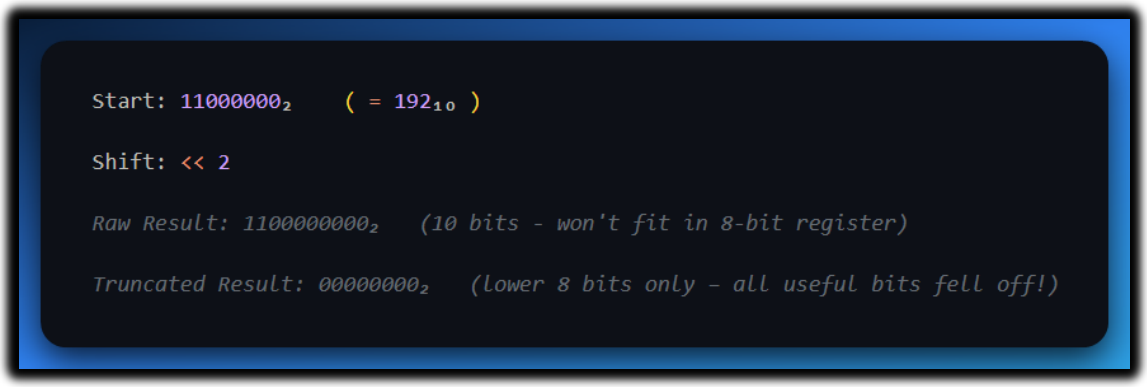
- You started with **8 bits**: 00000011
- Shifted it **left by 2**: becomes 00001100
- Still **fits within 8 bits** → No overflow, no lost bits

Only the leftmost bits "fall off" if the result is too big to fit in the given bit size (aka **overflow**).

👄 Example with Bit Loss (Overflow)

Let's force a fall-off using an 8-bit register:

Shift 11000000_2 (decimal 192) left by 2

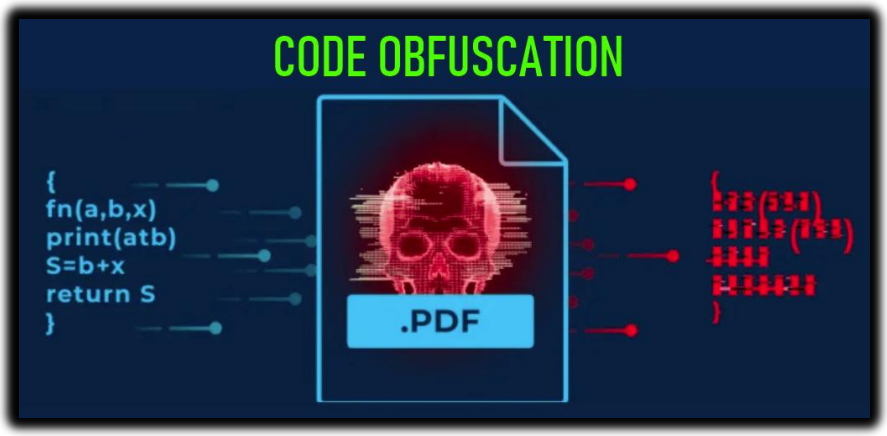


So yeah — **bits fall off only when there's no room** to keep them after the shift.

Term	Meaning
Left Shift	Moves bits to the left, multiplies by powers of 2
Bit Fall-off	Happens **only if shifted result exceeds bit limit (e.g., 8 bits)**
Padded Zeroes	Empty spots on the right are filled with `0` after shift

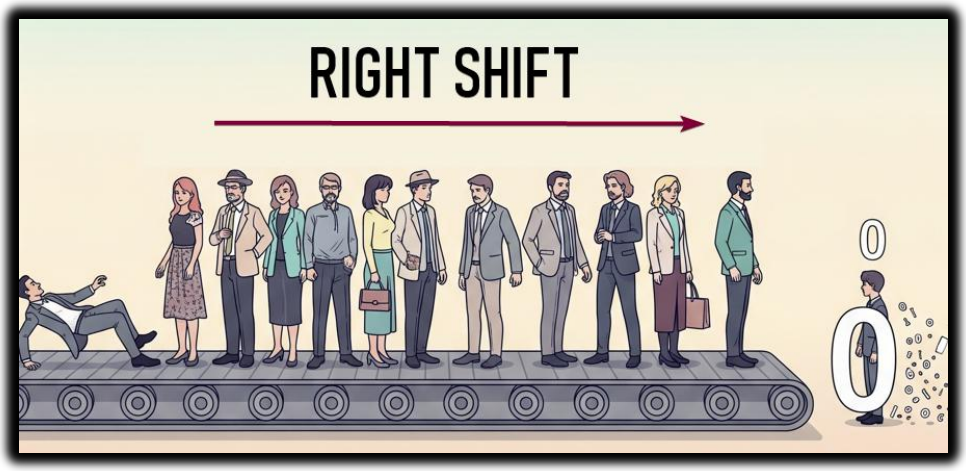
Reverse Engineering/Malware Context:

- **Fast Multiplication:** Compilers often replace multiplications by powers of 2 with left shifts because they are significantly faster for the CPU.
- **Bit Packing:** Used to quickly move bits into higher positions when assembling larger values from smaller components.
- **Obfuscation:** Can be part of simple arithmetic obfuscation to hide true values.



1.2. Right Shift (>>)

Concept: Now, everyone takes a step to their right. The person at the far right (least significant bit) falls off the treadmill, and a new person joins at the far left (most significant bit). The crucial part here is *who* joins at the left.

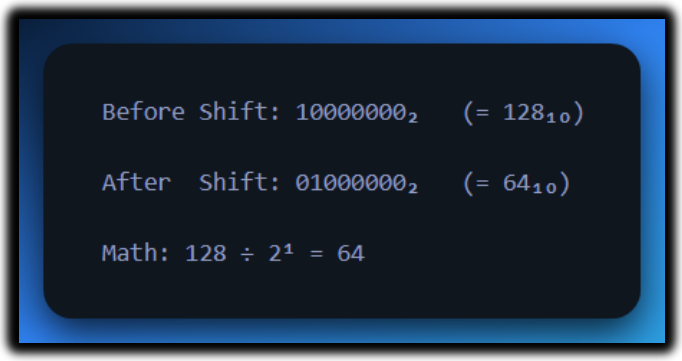


How it Works: Bits move to the right. The rightmost bit(s) are discarded. The leftmost bit(s) are filled based on the type of right shift:

1. Logical Right Shift (LSR):

Fills the leftmost bits with 0s. This is typically used for **unsigned numbers**.

Shift 10000000_2 (128) right by 1:



2. Arithmetic Right Shift (ASR):

Fills the leftmost bits with the value of the **original Most Significant Bit (MSB)**. This preserves the sign of the number and is typically used for **signed numbers**.

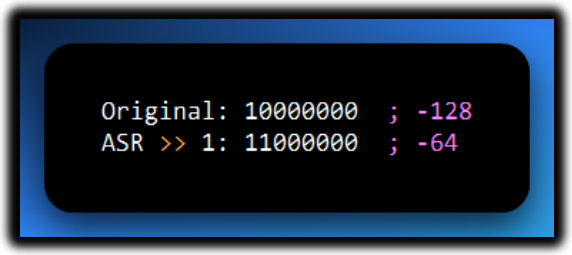
- ASR shifts the bits to the **right**, like a normal right shift.
- But the **leftmost bit (MSB)** — which is the **sign bit** in two's complement — is **copied** to fill in the blank spots on the left.
- This preserves whether the number was **positive or negative** after the shift.

🔍 Why Is This Important?

- In signed binary numbers (two's complement), **1 in the MSB** means the number is **negative**.
- If you just shoved in 0s during a right shift, you'd change a negative number into a positive — which is **completely wrong**.
- So, we **extend the sign** using the MSB during the shift — that's called **sign extension**.

🔢 Example 1: Shifting a Negative Number

Let's take **-128** in 8-bit two's complement:

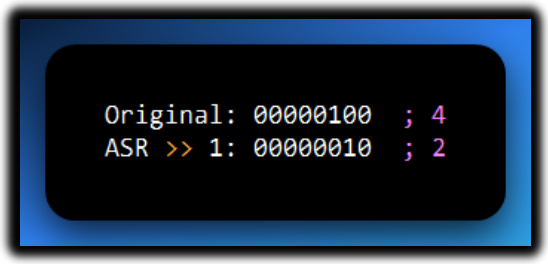


Step-by-step:

1. The 1 at the MSB (sign bit) says this number is negative.
2. ASR shifts everything right one place.
3. The new bit that comes in on the left is also a 1 (copied from the MSB).
4. The number remains **negative** — now it's -64.

🧠 If we used **logical shift** (fill with 0), we'd get 01000000 → **+64**, which is **wrong**.

🔢 Example 2: Shifting a Positive Number



Since the MSB is 0, the shift just fills with 0s — same as logical right shift.

✅ Key Concept: Divide by Powers of Two

Arithmetic right shift **divides a signed number by 2^n**, using integer division:

Binary	Decimal	ASR >> 1	Result
11111110	-2	11111111	-1
10000000	-128	11000000	-64
00001000	8	00000100	4
00000101	5	00000010	2 (floor of 2.5)

Here are the three key uses of the Arithmetic Right Shift (ASR) in a reverse engineering, malware, or hacking context, explained in simple sentences:

- **Fast Division:** ASR is used for very quick division of signed numbers by powers of two, as it's much faster than traditional division operations.
- **Bit Field Extraction:** It helps you isolate and grab specific groups of bits or flags from a larger value, discarding the rest.
- **Sign Preservation:** ASR is crucial for maintaining the correct positive or negative sign of a number when you're manipulating its bits or extending it to a larger size.

🔧 Assembly Tip:

Operation	Instruction
Logical Right	SHR
Arithmetic Right	SAR
Signed? Use SAR	✅
Unsigned? Use SHR	✅

2. Rotate Operations: The Circular Dance

A **rotate operation** is similar to a shift, but with a key difference: the bits that are shifted out of one end are **moved to the other end**, effectively wrapping around. No bits are lost, and no new bits are introduced. It's like a *circular conveyor belt of bits.*

2.1. Left Rotate (ROL)

- **Concept:** Imagine a circular queue of people. When you left-rotate, the person at the front moves to the back of the line, and everyone else steps forward.
- **How it Works:** Bits move to the left. The leftmost bit(s) that are shifted out are reinserted at the rightmost position(s).
- **Example (4-bit):** Rotating 1010_2 (decimal 10) one position to the left: 1010_2 ROL 1 = 0101_2 (decimal 5) (The 1 on the far left wraps around to the far right.)
- **Example (8-bit):** Rotating 10000001_2 one position to the left: 10000001_2 ROL 1 = 00000011_2

2.2. Right Rotate (ROR)

- **Concept:** The opposite of a left rotate. The person at the back of the circular queue moves to the front, and everyone else steps backward.
- **How it Works:** Bits move to the right. The rightmost bit(s) that are shifted out are reinserted at the leftmost position(s).
- **Example (4-bit):** Rotating 1010_2 (decimal 10) one position to the right: 1010_2 ROR 1 = 0101_2 (decimal 5) (The 0 on the far right wraps around to the far left.)
- **Example (8-bit):** Rotating 00000011_2 one position to the right: 00000011_2 ROR 1 = 10000001_2

2.3. Rotate Through Carry (RCL/RCR) - An Advanced Twist

- Some architectures (like x86) also have **"rotate through carry"** instructions (RCL, RCR). These rotations **include the CPU's Carry Flag** in the rotation. It's like having an extra bit outside your main register that also participates in the circular movement.
- **RE/Malware Context:** Often used in cryptographic algorithms (especially block ciphers like AES, DES), checksum calculations, and string obfuscation routines. If you see ROL or ROR in assembly, especially repeatedly, it's a **strong hint** that some form of **data transformation** or **encryption is occurring.** Malware frequently uses these for simple XOR-based encoding or custom encryption schemes.

3. Importance in RE/Malware Analysis

- **Performance Optimization:** Shifts are CPU-efficient ways to perform multiplication and division by powers of 2. Recognizing them helps you understand the compiler's optimizations.
- **Bit Manipulation:** Directly setting, clearing, or testing individual bits within a byte or word (e.g., checking flags in a status register or parsing bitmasks).
- **Obfuscation Techniques:** Malware authors use shifts and rotates to obfuscate constants, API names, or small code snippets. A string might be XORed with a key that is then rotated, making the original string harder to identify in static analysis.
- **Cryptographic Primitives:** Rotates are fundamental building blocks in many symmetric-key cryptographic algorithms (e.g., in AES's ShiftRows, or various custom block ciphers). Identifying these patterns can help you recognize the algorithm being used.
- **Checksums and Hashes:** Simple checksums or hash functions might use a series of shifts and XORs to generate a unique value for data integrity checks.
- **Anti-Analysis:** Sometimes, shifts and rotates are used in anti-debugging or anti-VM techniques to perform calculations that are difficult for automated tools to follow without proper emulation.

When you're in Ghidra, looking at assembly code, keep an eye out for instructions like SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR.

These instructions are direct **indicators of bit-level manipulation**, and understanding their effect is crucial for deciphering the program's logic.

SHIFTS AND ROTATES: THE COMPLETE BIT MANIPULATION TOOLKIT

These operations are the fundamental ways CPUs rearrange bits within registers or memory.

Mastering them is like understanding the very gears of a machine, essential for dissecting compiled code.

I think we need to repeat sometimes in order for those left behind to just get it fully. Let's go! ⚡ 😤

1. Shift Operations: Bits In, Bits Out (The Conveyor Belt)

Shift operations move bits to the left or right. Bits falling off one end are lost, and new bits are introduced on the other. The key distinction is how the empty positions are filled.

1.1. Left Shifts

Logical Left Shift (SHL - Shift Logical Left):

Concept: Every bit moves one position to the left. The bit that "falls off" the most significant (leftmost) end goes into the **Carry Flag (CF)**. The least significant (rightmost) bit is filled with a 0.

Effect: Equivalent to multiplying by 2N.

Example (8-bit): 01010101_2 (85) SHL 1 CF <- 0 (original MSB) 01010101 010101010 (shifted) 0101010 0 (new LSB is 0) Result: 10101010_2 (170), CF = 0

Note: On x86, SHL and SAL (Shift Arithmetic Left) are identical in operation.

1.2. Right Shifts

Logical Right Shift (SHR - Shift Logical Right):

Concept: Every bit moves one position to the right. The bit that "falls off" the least significant (rightmost) end goes into the **Carry Flag (CF)**. The most significant (leftmost) bit is filled with a 0.

Effect: Equivalent to unsigned division by 2N.

Example (8-bit): 10101010_2 (170) SHR 1 CF <- 0 (original LSB) 10101010 01010101 (shifted, new MSB is 0) Result: 01010101_2 (85), CF = 0

Use Case: Primarily for unsigned integer division.

Arithmetic Right Shift (SAR - Shift Arithmetic Right):

Concept: Every bit moves one position to the right. The bit that "falls off" the least significant (rightmost) end goes into the **Carry Flag (CF)**. The most significant (leftmost) bit is filled with a copy of the **original MSB (sign bit)**. This preserves the sign of the number.

Effect: Equivalent to signed integer division by 2N.

Example (8-bit, positive): 00001010_2 (10) SAR 1 CF <- 0 00000101_2 (5), CF = 0

Example (8-bit, negative): 11111010_2 (-6) SAR 1 CF <- 0 11111101_2 (-3), CF = 0 (original MSB was 1, so 1 is copied)

Use Case: For signed integer division.

2. Rotate Operations: The Circular Dance (Bits Never Die!)

Rotate operations move bits around a circular path. Bits shifted out from one end are re-inserted at the other end. No bits are lost or introduced.

2.1. Simple Rotates (Without Carry Flag)

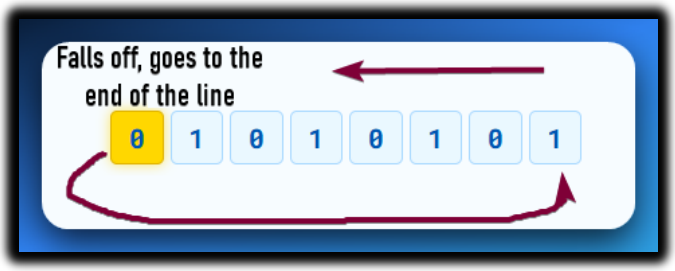
🔄 Rotate Left (ROL)

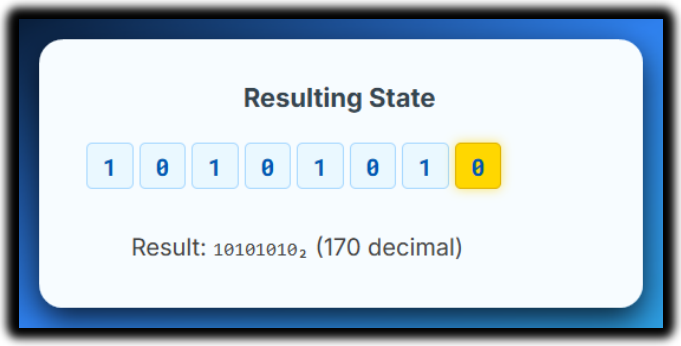
Shifts all bits to the left. The leftmost bit (MSB) *wraps around* to the rightmost bit (LSB), like a conveyor belt. Also, the original MSB gets saved in the Carry Flag (CF) — like a side note for the CPU.

Example (8-bit):

Initial register value was 01010101_2 (85) and we're doing a ROL 1 no need for a CF here.

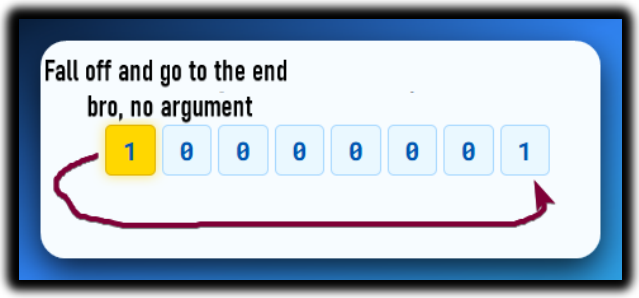
Original MSB 0, wraps to LSB. Result: 10101010_2 (170), CF = 0





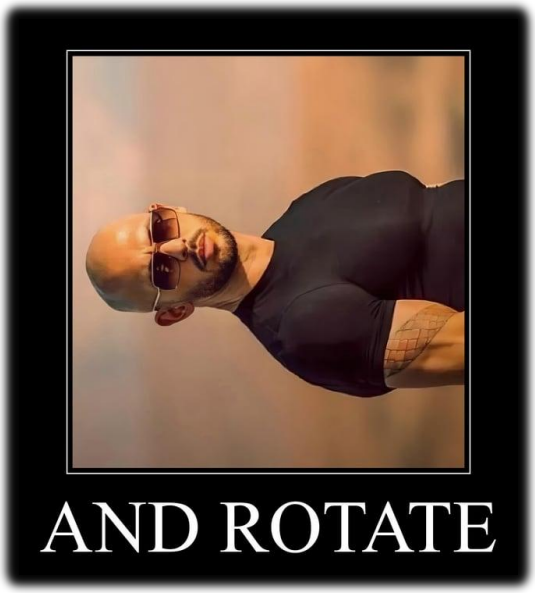
Example (8-bit):

Initial register holds 10000001_2 (129) we are performing a ROL 1, carry flag is not being used at all so we leave it as **?**



Original MSB goes and becomes the LSB (Least Significant Bit) resulting in 00000011_2 (3) and **no bits are lost.**

I need this space gone:



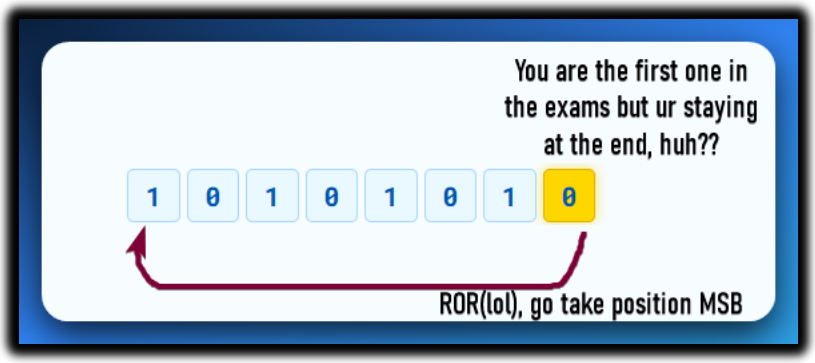
🔄 Rotate Right (ROR - Rotate Right):

The bits shift right, and the bit that "falls off" the LSB end wraps around and becomes the new MSB. The original LSB also goes into the **Carry Flag (CF)**.

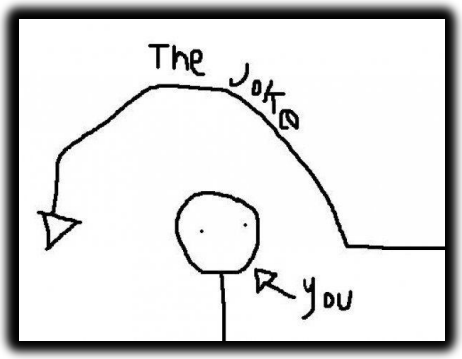
Example (8-bit):

Initial State: Register = 10101010_2 (170 decimal) and we're performing a ROR 1 and it has not Carry Flag, just like the Rotate Left.

Original LSB 0 wraps to MSB. This Results to 01010101_2 (85), CF = 0

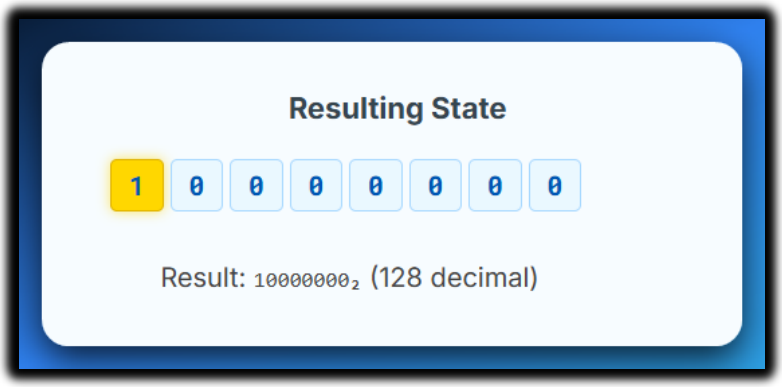


If you didn't see the joke, you're not reading this book the right way… 💀



Example (8-bit):

Initial register state is 00000001_2 (1 decimal) and we're performing a Rotate Right, ROR 1, no Carry Flags here.



2.2. Rotates Through Carry (Including the Carry Flag!)

These are particularly interesting because they involve the CPU's **Carry Flag (CF)** as an extra bit in the rotation. This effectively makes the rotation operate on N+1 bits (where N is the size of the operand, e.g., 8, 16, 32, 64 bits).

Rotate Left Through Carry (RCL - Rotate Through Carry Left):

Concept: The bits shift left. The bit that "falls off" the MSB end goes into the **Carry Flag (CF)**. Simultaneously, the **original value of the Carry Flag** is inserted into the LSB position. It's like a circular shift involving the register and the Carry Flag as a single, larger bit string.

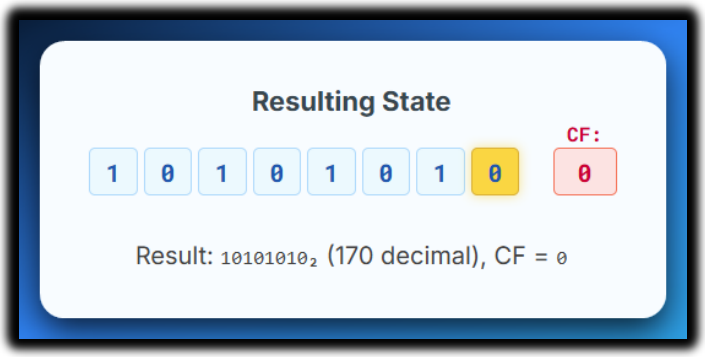
Example (8-bit, CF=0):

Initial state of the register is 01010101_2 (85), the carry flag is 0 and we're doing RCL 1.

So, when we say rotate left by carry, we mean, go and stand at the left most side of the 8 bits, that's where the MSB with 0 is.

Then tell everyone in the line to move one step to the left.

MSB (0) moves into the carry flag (CF). Original Carry Flag value (0), moves to LSB.

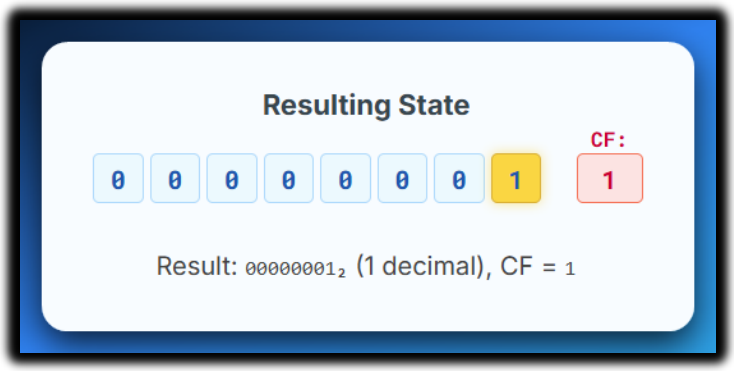


Example (8-bit, CF=1):

Initial state of the register is 10000000_2 (128), the Carry Flag is 1 and we're doing RCL 1.

MSB (Most Significant Bit) with the value (1) moves into the Carry Flag.

Original Carry Flag which had the value (1) moves to LSB (Least Significant Bit).



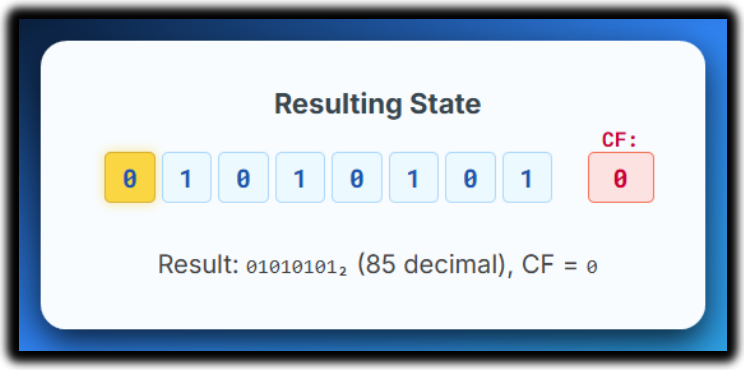
Rotate Right Through Carry (RCR - Rotate Through Carry Right):

The bits shift right. The bit that "falls off" the LSB end goes into the **Carry Flag (CF)**. Simultaneously, the **original value of the Carry Flag** is inserted into the MSB position.

Example (8-bit, CF=0):

The initial state of the register is 10101010_2 (170), the Carry Flag CF=0 and we're performing a RCR 1.

LSB (0) moves to Carry Flag. Original Carry Flag (0) moves to MSB.

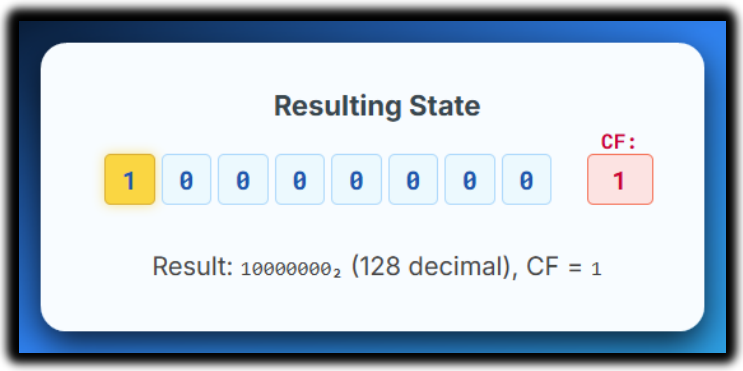


Example (8-bit, CF=1):

Initial state of the register is 00000001_2 (1), the Carry Flag is 1 and we're doing RCL 1.

LSB (1) moves into Carry Flag.

Original CF (1) moves to MSB.



3. The Carry Flag's Crucial Role

The **Carry Flag (CF)** is a single bit in the CPU's FLAGS register. It's often used to:

- *Indicate an overflow in addition or underflow in subtraction.*
- *Store the last bit shifted out in shift operations.*
- *Act as an "extra bit" in rotate-through-carry operations.*

In reverse engineering, pay close attention to instructions that modify or read the Carry Flag, especially in conjunction with shifts and rotates. This is a common pattern in:

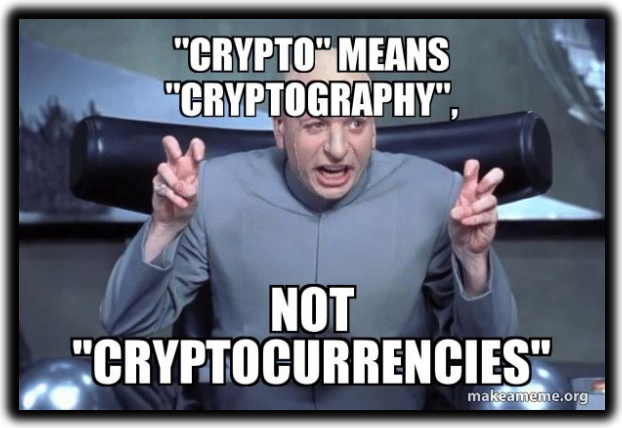
- **Arbitrary Precision Arithmetic:** When numbers are too large for a single register, operations are chained using the Carry Flag.
- **Checksums and Hashes:** Many simple algorithms use shifts and rotates, often incorporating the Carry Flag, to mix bits.
- **Cryptographic Algorithms:** Rotates (especially through carry) are fundamental operations in many block ciphers and stream ciphers for bit mixing and diffusion. If you see RCL or RCR in a loop, you're likely looking at a cryptographic primitive or a custom obfuscation routine.
- **Bitstream Processing:** For handling data streams where bits need to be precisely moved and extracted.

Check the html associated with this topic.

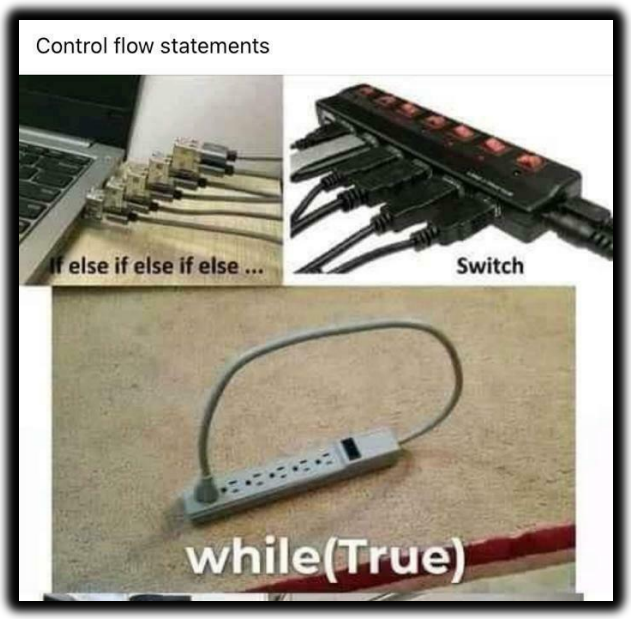
Practical Implications for RE/Malware Analysis

Identifying Obfuscation: Repeated shifts and rotates, especially with varying counts, are common in malware to obfuscate strings, API calls, or small code blocks. Your goal is to reverse these operations.

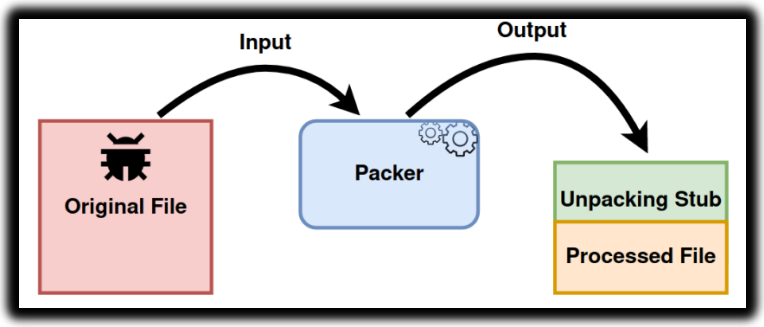
Recognizing Crypto: If you see ROL/ROR or RCL/RCR instructions, particularly in loops or with fixed rotation counts (e.g., rotate by 7, rotate by 13), it's a strong indicator of a custom encryption algorithm or a standard cryptographic primitive.



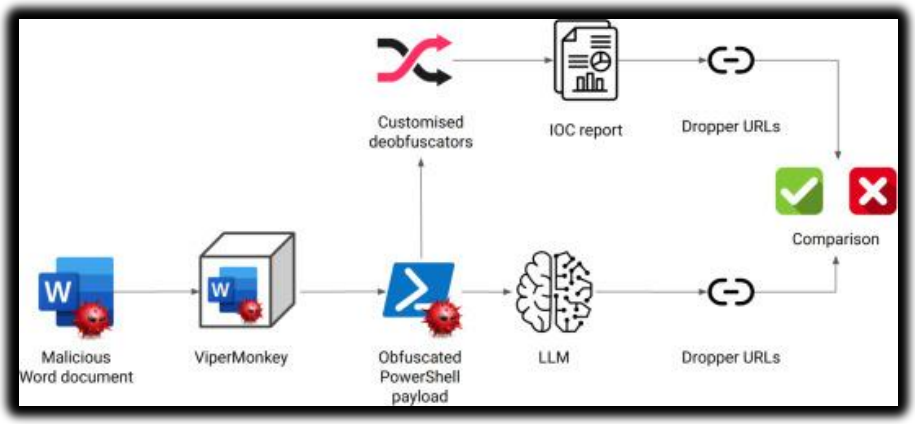
Understanding Control Flow: Shifts can be used to set or clear flags that influence conditional jumps.



Data Parsing: When dealing with packed binary data or custom file formats, shifts and rotates are used to extract specific bit fields.



Manual Deobfuscation: Sometimes, you'll need to manually trace the effect of these operations on a value to deobfuscate it. This is where your understanding of each type becomes paramount.



By understanding these nuances, you're not just seeing assembly instructions; you're seeing the **precise bit-level logic** that the program is executing. Assembly is difficult but once you get the flow you'll be feared by all terminals including the AI powered ones like warp (not sponsors, just wondering how AI reached into my CMD)…