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.

Remember the html has a CF issue there's places carry flag isn't needed, you can ignore the html and read the notes, or read the html too and correct it yourself.

🌀 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 zero (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.

Before Shift: 1010_2 ($= 10_{10}$)

After Shift: 10100_2 ($= 20_{10}$)

Math: $10 * 2^1 = 20$

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

Before Shift: 00000011_2 ($= 3_{10}$)

After Shift: 00001100_2 ($= 12_{10}$)

Math: $3 * 2^2 = 3 * 4 = 12$

😲 But wait... Why Didn't Bits Fall Off when we shifted 1010?

That's a golden question — let's clear this up:

Do bits fall off when you do a **left shift**?

✅ **Yes**, but **only** if the shifted result is too big to fit inside the register you're using (e.g., 8-bit, 16-bit, 32-bit).

But here's the key: If your original number is something like 1010 (which is 4 bits), and you're working in an 8-bit register, then you've still got room to grow.

🧪 Example:

- You start with 00001010 (8-bit version of 1010)
- Shift left by 2 → 00101000
- Still 8 bits — so nothing falls off

➡ **No bits lost.**

Now, let's say you had this:

- *Start with 11001010*
- *Shift left by 2 → 00101000*

The two leftmost bits (11) just **fall off the edge** and disappear.
Because 8-bit register can't hold more than 8 bits.

TLDR:

Shifting doesn't care **how full** your number *looks* —
It only cares about the size of the register.
Bits only "fall off" when the result **overflows** that size. Otherwise, it's just moving around inside the space.

👄 Example with Bit Loss (Overflow)

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

Shift 11000000_2 (decimal 192) left by 2

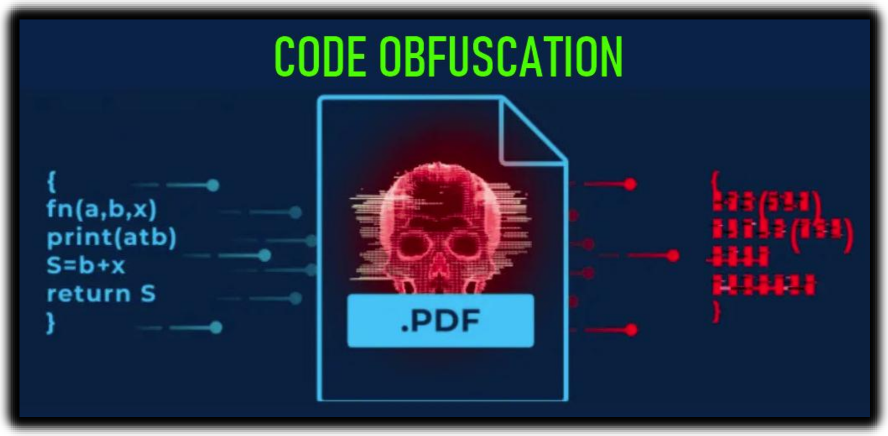
```
Start: 11000000₂    ( = 192₁₀ )

Shift: << 2

Raw Result: 1100000000₂   (10 bits - won't fit in 8-bit register)

Truncated Result: 00000000₂   (lower 8 bits only - all useful bits fell off!)
```

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

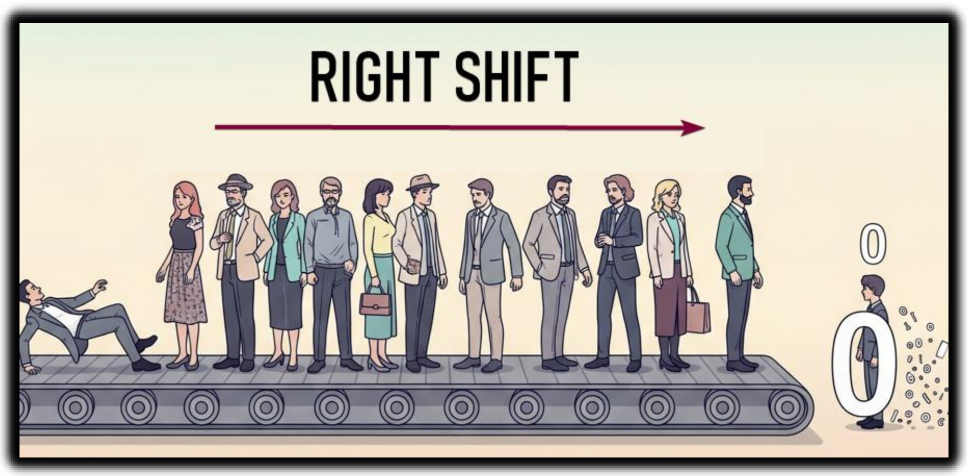
Reverse Engineering/Malware Context:

- **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:

```
Before Shift: 10000000₂   (= 128₁₀)

After  Shift: 01000000₂   (= 64₁₀)

Math: 128 ÷ 2¹ = 64
```

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.

We discuss this in depth on the next section. Keep reading ⚡

🔍 Why Is ASR 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:

```
Original: 10000000  ; -128
ASR >> 1: 11000000  ; -64
```

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

```
Original: 00000100  ; 4
ASR >> 1: 00000010  ; 2
```

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`	✅

SHIFTS AND ROTATES: THE COMPLETE BIT MANIPULATION TOOLKIT PART 2

⚡ Heads-Up for All Reverse Engineers:

When you're digging through disassembly in **Ghidra**, **IDA**, or **Binary Ninja** —
watch for these ops:
SHL, SHR, SAL, SAR, ROL, ROR, RCL, RCR

⚡ Why?

Because these are **not just random instructions** — they're **bit surgeons**.
They cut, flip, and rotate bits inside registers like pros. If you don't know what they're doing, you're missing the *plot twist* in the code.

⚡ Real Talk:

These are the tools that low-level code uses to:

- Mix up values.
- Encrypt stuff.
- Parse flags.
- Hide secrets.
- Trick analysis tools.

All of it — **through just bits**.

🧠 Get this deep in your brain:

Shifts and rotates **are the gears turning inside the CPU**.
They're not fancy, but they're **everywhere** — and once you get them,
you start seeing the **actual logic** behind compiled code, not just machine gibberish.

📣 Repeating this louder for the beginners in the back:

If you don't understand how SHL or ROR and the rest works, you're flying blind.
But once you do?
You're reading **the machine's native language**. You're dangerous now. 😤

🧾 The Bitwise Crew Crooks: Official Names vs Nicknames

💻 Opcode	🧠 Official Name	👓 Nicknames / Aliases	🔎 Used For
`SHL`	Shift Logical Left	Left Shift, LSH, SLL (MIPS), `<<` in C	Multiply by 2, bit masking
`SAL`	Shift Arithmetic Left	Same as SHL (they're literally identical on x86)	Same stuff — CPU treats them the same
`SHR`	Shift Logical Right	Right Shift, LSR (Logical Shift Right), `>>` (unsigned in C)	Divide by 2, clean bit slicing
`SAR`	Shift Arithmetic Right	Signed Right Shift, ASR (Arithmetic Shift Right)	Keeps the sign bit (for signed division)
`ROL`	Rotate Left	Left Rotate, Circular Left Shift	Cryptography, bit mixing
`ROR`	Rotate Right	Right Rotate, Circular Right Shift	Crypto, obfuscation
`RCL`	Rotate through Carry Left	Rotate Left w/ Carry	Obfuscation, crypto, trick debugging
`RCR`	Rotate through Carry Right	Rotate Right w/ Carry	Same stuff, different direction

⚠️ Why So Many Names?!

- Different **CPU architectures** = different mnemonics (MIPS, ARM, x86, etc.)
- Different **languages** = different syntax (<<, >>, etc.)
- Some just **sound cooler** (you gonna say "Rotate through carry" or just "RCL" like a pro?)

🧠 Memory Tip:

If it **has "Shift"** in the name, it probably drops bits.
If it **has "Rotate"**, it's playing hot potato with bits — nothing gets lost.
If it has **"Arithmetic"**, it's being sensitive about signs 😭

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.

We won't repeat left shifts. Simple stuff. But for right shifts, lets give it another spin, coz there's two versions.

🥱 1.1. Left Shifts?

Just SHL or SAL (they're the same instruction).

SHL (Shift Logical Left) and SAL (Shift Arithmetic Left) instructions are **literally the exact same instruction** on most modern CPUs.

Nothing fancy — just push bits left and pad the right with zeros. Done.
💡 **Easy = no need to repeat.**

Since the operation for both **"logical"** and **"arithmetic"** left shifts would involve bringing in zeros from the right, and the **sign bit doesn't require special handling** to maintain arithmetic correctness during a left shift (it just moves), **there's no practical difference in their execution**.

They perform the **exact same bit manipulation.** The CPU designers effectively merged them into one operation with two mnemonics for clarity or historical reasons.

1.2. Right Shifts

💻 1. Logical Right Shift (SHR - Shift Logical Right):

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.

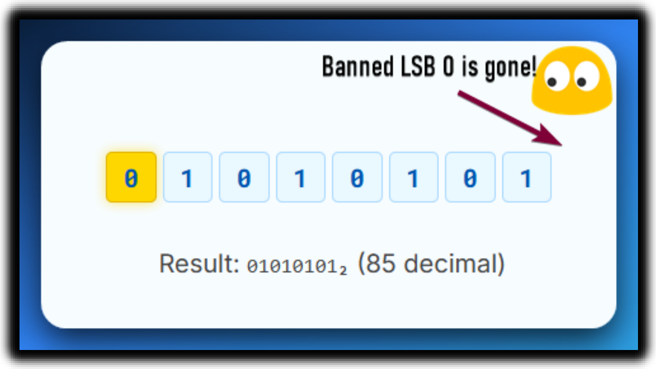
Effect: Equivalent to unsigned division by 2N.

Example (8-bit):

Initial State of the 8-bit register is 10101010_2 (170 decimal), and we're performing a SHR 1 no need for carry flag here.

Shifted everyone moves one step to the right. The LSB gets banned, falls off and we introduce a new MSB ninja, 0 to replace the fallen one, and this results to 01010101_2 (85), CF = irrelevant here.

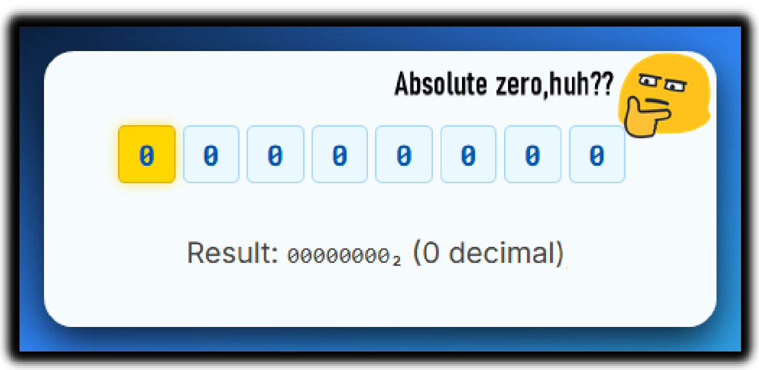
Don't get lost, sorry I drew 7 bits 💀, let me show you the real results after Ninja comes in.



Use Case: Primarily for unsigned integer division.

Quick example 2, lets shift right the number 1 and see.

Initial State: Register = 00000001_2 (1 decimal)



💻 2. Arithmetic Right Shift (SAR - Shift Arithmetic Right):

Every bit moves one position to the right. The bit that "falls off" the least significant (rightmost) end goes into the **Carry Flag (CF)**.

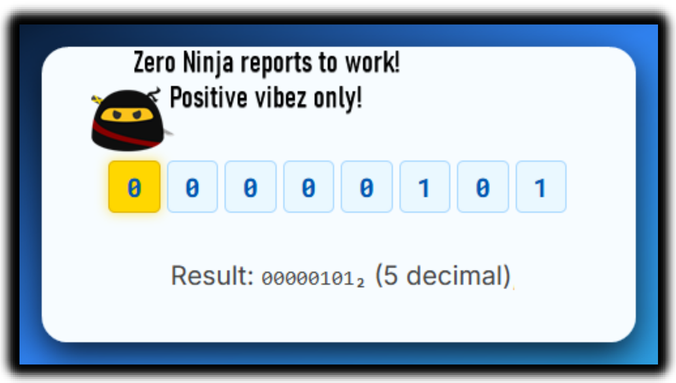
The **most significant (leftmost) bit** is filled with a copy of the **original MSB (sign bit)**. This preserves the sign of the number.

Effect: Equivalent to signed integer division by 2N.

Example (8-bit, positive):

The initial register value is 00001010_2 (10) and we're performing a SAR 1, Carry Flag, never heard of her (irrelevant here).

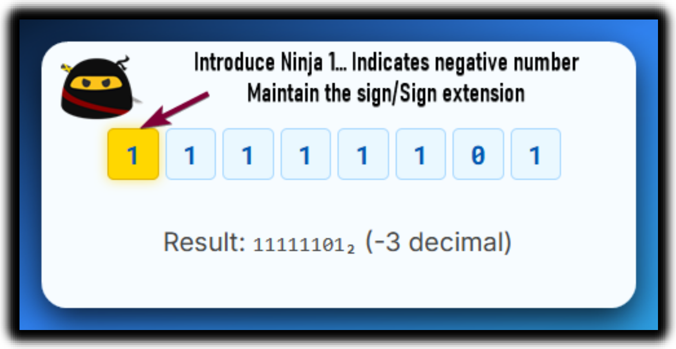
After shifting, we remember, the sign has to be maintained. So, if it was a zero, we introduce a **0 ninja,** else we introduce a **1 ninja**, indicating negative. (sign extension)



Example 2 (8-bit, negative):

The initial value for the register is **11111010_2 (-6)** and we're performing a **SAR 1**, no carry flag needed.

Original MSB was 1, so 1 is copied to maintain the sign of the new resulting number.



Use Case: For signed integer division.

Now, you're one day closer to open sourcing every app on your PC.

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)

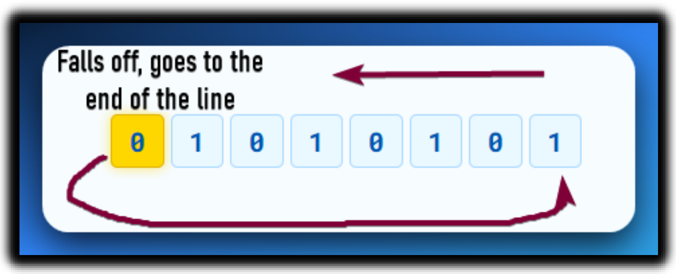
🔄 Rotate Left (ROL)

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):

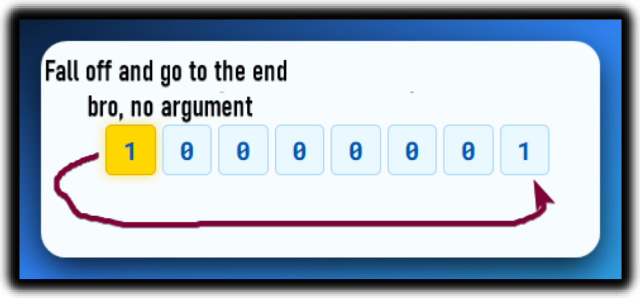
Initial register value was 01010101_2 (85) and we're doing a ROL 1 no need for a CF here.

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



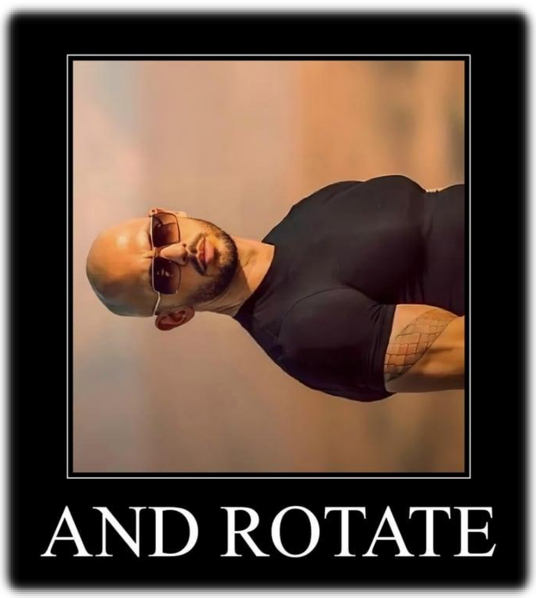
Example (8-bit):

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:



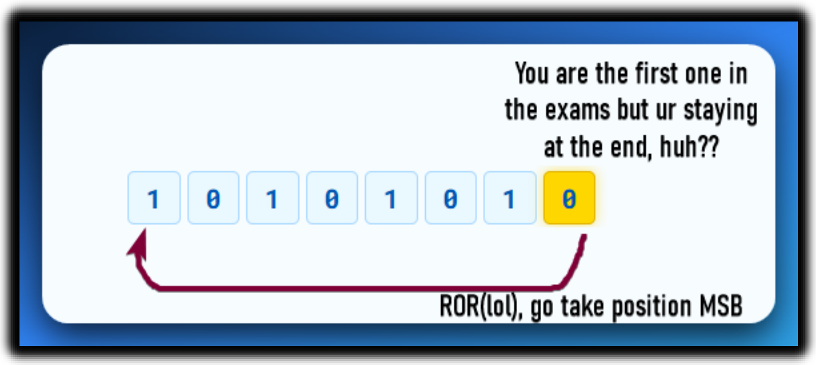
🔄 Rotate Right (ROR - Rotate Right):

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)**.

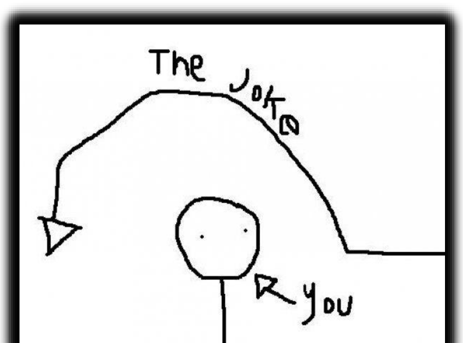
Example (8-bit):

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

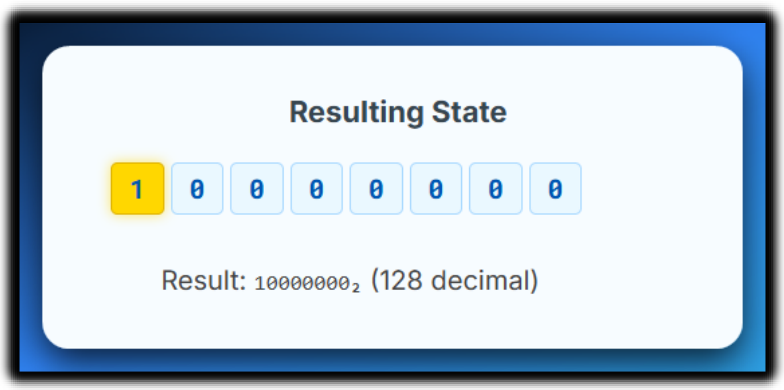


If you didn't see the joke, you're not reading this book the right way… 💀



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.

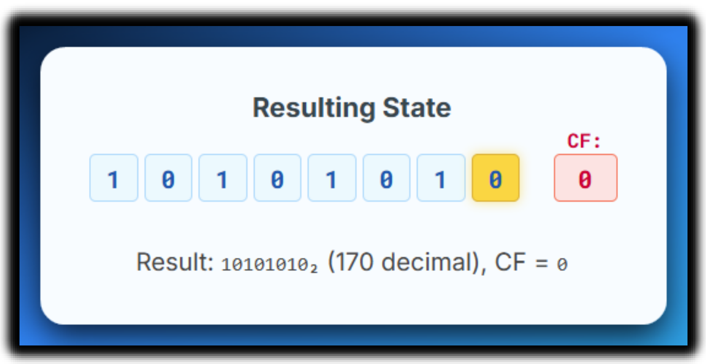
Example (8-bit, CF=0):

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.

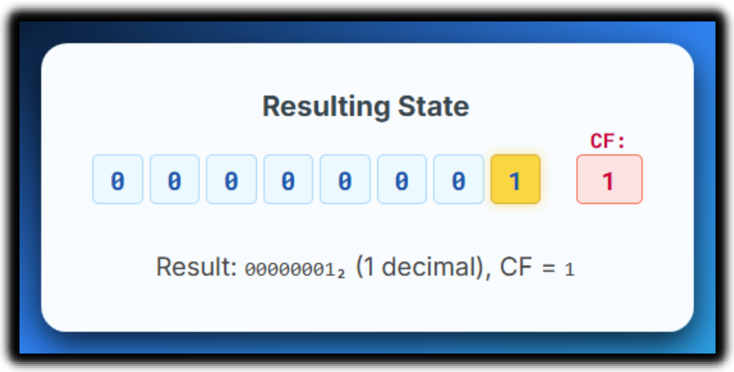


Example (8-bit, CF=1):

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).



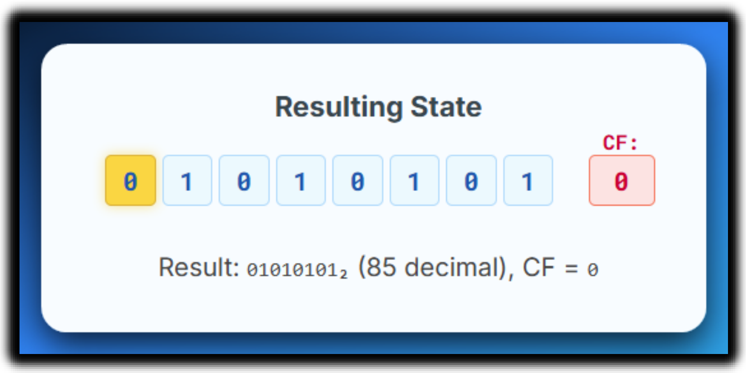
Rotate Right Through Carry (RCR - Rotate Through Carry Right):

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.

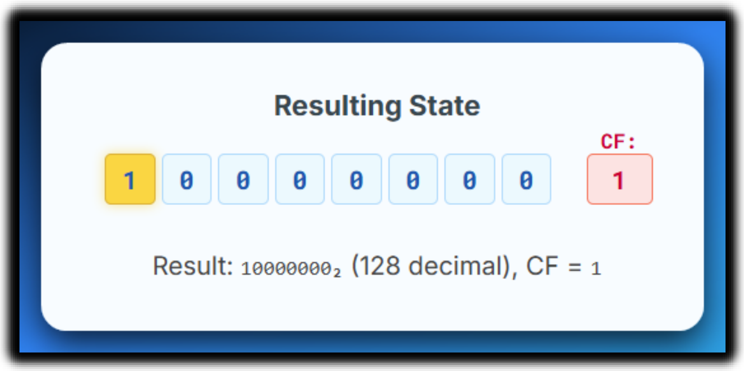


Example (8-bit, CF=1):

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.*

⚙️ Bitwise Ops in Reverse Engineering — The Real Talk:

When you're reversing code, **bitwise operations** (like shifts and rotates) are huge. Why? Because they're the CPU's way of doing low-level, bit-by-bit magic.

Pay extra attention when you see them, especially with the **Carry Flag (CF).** It's like the CPU's sticky note that bits can jump into or out of, making things spicy.

Here's where you can apply these bit-manipulation tricks...

Carry Flag Awareness

If a shift or rotate touches the Carry Flag, it's probably doing something important, maybe even sneaky. Keep an eye on it.

🧠 When Numbers Get Too Big... Use bit manipulation operations

Sometimes, a number is just *too big* to fit in one CPU box (register). When that happens, the **Carry Flag (CF)** acts like a **handoff signal** — it passes the extra "leftover" part of the number to the next register, kinda like **train cars linking up** to carry more stuff.

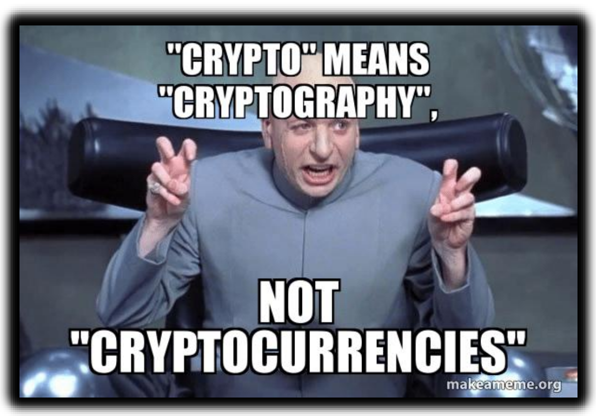
So basically:

"Oops, too much! Here, you take the rest 👉" — says the Carry Flag to the next register.

🕵️ Hiding & Mixing Data — Obfuscation

- **Hashes & Checksums:**
  These are like *digital fingerprints* for files or data. To make them, the CPU often uses things like **bit shifts**, **rotations**, and the **Carry Flag** to mix up the bits and create a unique result. It's simple scrambling — not super-secret, just enough to tell if something changed.

- **Crypto Routines (aka the Secret Sauce):**
  If you spot instructions like **ROL, ROR, RCL, or RCR** (especially inside loops or using weird numbers like **7** or **13**) — that's not by accident. That's usually **encryption** or **obfuscation.** The code is hiding something — either for security, compression, or making it hard to reverse-engineer.



🧩 Custom Data Vibes — Breaking It Down

- **Bitstream Parsing (aka Unpacking Stuff):**
  When you're dealing with weird file formats or network data, you don’t just read it like plain text. You use shifts and rotates to pick out the exact bits you want — like zooming in on tiny parts of a puzzle.

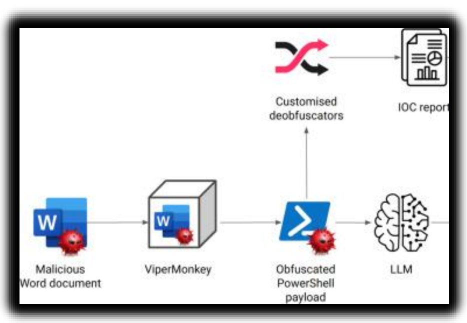
- **Data Extraction (aka Field Tripping):**
  Sometimes, one value holds multiple pieces of info mushed together. That’s common in network packets or compressed formats. To break it apart, you use bit shifts and bit masks to grab just what you need — piece by piece, clean and surgical.

🦠 Malware’s Dirty Tricks

- **Obfuscation (Malware’s Camo):**
  Malware hides its secrets (like strings & API calls) using shifts and rotates. Weird patterns? Repeated rotates? That’s probably it hiding something sketchy.

- **Manual Deobfuscation (Un-scrambling):**
  No auto-magic here. You might have to step through every shift and rotate by hand to uncover what it’s doing. Pure pain.

- **Anti-Debugging (Hide & Seek):**
  Some malware uses bit tricks to confuse debuggers, break VMs, or mess with analysis tools. It’s like code yelling, “You won’t catch me, nerd.” VMProtect, Themida.

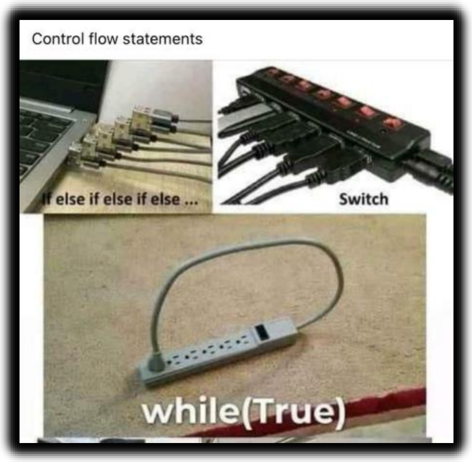


⚙️ CPU Efficiency & Control

- **Control Flow Hacks:**
  Shifts can change CPU flags, which affects **jumps, loops, and decision-making**. That means code can *sneakily switch its logic* mid-run.

- **Speed Tricks (CPU Cheat Codes):**
  Want to multiply or divide by 2 fast? Use a **bit shift**. Compilers do this all the time — it's way faster than normal math.

- **Bit Masking (Flip Those Bits):**
  Need to **turn a bit on/off** or just **check if it's set**? Use a **shift + bitmask combo**. Classic low-level control move.



🧨 Cryptography Red Flag Checklist

If you see these in a binary... something's **being hidden** 👀

- 🔁 **Rotates in loops?**
  Yeah, that ain't compression — that's cryptography.

- 🧮 **Rotate counts like 7, 13, or 17?**
  Not random. Those are *prime-time* crypto numbers.

- 💥 **Combo moves like XOR, ADD, SUB, NOT?**
  That's textbook obfuscation — trying to fry your brain.

- 📦 **Operands pulled from memory?**
  They're not just doing math... they're **unpacking secrets**.

An **opcode** tells the CPU what to do, while an **operand** tells it what to do it to. In the assembly instruction MOV EAX, 5, MOV is the **opcode** and EAX and 5 are the **operands**.

Big Picture:

Once you get these bit-level moves, you're not just reading assembly — you're going to be executing raw bit-level strategies in your ASM code.