Analysis of Disassembled SHA-Hash Code

May 13, 2025

1 Analysis of Disassembled SHA-Hash Code (Input: 512× '3' Characters)

By Dean Kulik Qu Harmonics. quantum@kulikdesign.com

1.1 Overview of the Disassembled Code Structure

When the 256-bit SHA hash of 512 `'3'` characters is interpreted as x86 machine code, the resulting instructions appear largely as "junk" code. Instead of a meaningful program, we see a pseudo-random mix of operations. However, **distinct patterns emerge in how this code manipulates state**, especially due to the repetitive `'3'` characters inserted in the transformations. We analyze several segments of the disassembly (the raw hash bytes, folded/combined values, and the ASCII-encoded binary) to understand their side effects on registers and memory. The focus is on how each instruction would mutate state (registers, flags, memory), any structural or repeating motifs, and whether these patterns resemble phases of a SHA-like algorithm (e.g. rounds of mixing, rotations, etc.).

Key findings: The disassembled bytes include *unusual or illegal opcodes* (e.g. `pop ss`, `daa`, `aaa`) with unique side effects, long **XOR cascades** (due to the `'3'` insertions) that sometimes form feedback loops (operations that undo each other), and even implicit **control flows** (like conditional jumps or flag-dependent operations). Some of these patterns intriguingly parallel cryptographic operations – for example, register swaps resemble state rotation, and constant XORs mimic round constant injection – albeit arising by coincidence.

1.2 Unusual and "Bad" Opcodes (State Mutations)

Several instructions in the disassembled hash output are *rare or invalid in typical programs*, yet they have clear side effects on CPU state:

- **Segment Register Ops:** The very first byte `0x17` decodes as `pop ss`, which pops a value from the stack into the SS (Stack Segment) register. This is *highly unusual* – it changes the stack segment and forces the processor to delay the next instruction until the new SS is validated. It also increments ESP (stack pointer) by 2 or 4. Shortly after, `push es` and `pop edi` appear. Pushing ES (Extra Segment) saves that segment value onto the stack, and later popping into EDI moves a 32-bit value from stack to EDI. These segment operations show **direct segment and pointer manipulation** (SS and ES are rarely touched in modern code). They don't correspond to any SHA algorithm step, but they drastically alter the execution environment (stack context). Another segment-related oddity is multiple segment override prefixes (`fs`, `gs`, `ss`) prepended to some instructions, which normally modify which

segment register is used for a memory operand. In the disassembly, we see nonsense like `fs gs fs ss gs cmp DWORD PTR ss:[edx+0x63],esp` – a result of consecutive prefix bytes. These have *no real effect* beyond the last prefix (`gs` and `ss` in this case), but they illustrate how the bytes are being interpreted as extraneous prefixes, a byproduct of the hash bytes rather than intentional code.

- **Illegal/Incomplete Instructions:** The disassembler flags some byte sequences as **"(bad)"**, meaning no valid instruction could be decoded. For example, at offset 0x1 the bytes `DE D6` are invalid, and at 0x5 the lone byte `C6` is an incomplete opcode (it's the prefix for *MOV r/m8, imm8* but missing the immediate). These "bad" opcodes indicate random data not aligning with any known instruction, a common occurrence when disassembling cryptographic output. They would typically cause an illegal instruction exception if executed, but here we note them to show the absence of any legitimate operation at those points (no state change beyond advancing the instruction pointer erratically).
- **I/O Port Operation:** At offset 0x3, we encounter `out 0x9B, al`. This writes the AL register to I/O port 0x9B. Its side effect is external to the CPU registers – it would send AL's value to hardware port 0x9B. This is an unusual instruction in most software (direct port access is seen in low-level drivers/OS code). Here it likely doesn't correspond to anything "SHA-like"; rather it's a random effect of the bytes `E6 9B`. Still, it exemplifies a **side effect on system state** (hardware port) rather than on general-purpose registers or memory.
- **Arithmetic Adjust Instructions:** Notably, the bytes contain ASCII-adjust opcodes: `daa` (Decimal Adjust after Addition) and `aaa` (ASCII Adjust after Addition). In the "combined number" disassembly, `0x27` appears as `daa`, and in the ASCII-hex disassembly `0x37` appears twice as `aaa`. These instructions were historically used for BCD (binary-coded decimal) arithmetic: **`DAA` adjusts the contents of AL after adding two BCD digits**, affecting AF (auxiliary carry) and CF (carry) flags. **`AAA` adjusts AL and AH after adding two ASCII (0x30–0x39) digits** in order to form a valid unpacked BCD result. In our context, their presence is coincidental but intriguing: for example, `daa` will modify AL based on the low 4-bit value and carry flag – a *bit-level side effect* adjusting half-bytes. Similarly, `aaa` will add 0x06 to AL and increment AH if the lower nibble was above 9 (or if AF was set), then set those adjust flags. In the disassembly, we see `aaa` appear consecutively: one plain and one with an `ss:` segment prefix (`ss aaa` at offset 0x22). Executed back-to-back, these would each adjust AL (likely twice, though the second's `ss:` has no effect on AAA's behavior). While BCD adjust isn't part of SHA, **the concept of a post-add adjustment** has a loose parallel to the modular reduction in SHA's word addition (SHA adds 32-bit words mod 2^{32}, automatically dropping carries). Here `AAA/DAA` drop carries above 9 in each nibble – not the same base, but still a *folding of overflow bits*. This can be seen as a kind of *"structural folding"* of a result into a constrained range (0–9 in each digit). In summary, these adjust opcodes mutate registers (AL, AH) and flags (AF, CF) in a way that *was not intended by any algorithm here*, but they do demonstrate how the disassembled bytes perform extra post-add tweaks – almost like ghost analogs of the mod 2^{32} operations in cryptographic rounds.
- **Data Transfer and Exchange:** We see ordinary moves and exchanges used in odd ways. For example, `mov cl, 0xE9` loads an immediate into CL, and soon after `cmp cl, [ebx+0x37]` uses that value in a comparison. The immediate load is straightforward state initialization (CL=0xE9) and the compare reads a byte from memory, affecting flags but not registers. More striking is the use of **XCHG (exchange)** instructions to swap registers. In the combined-number code, the sequence `xchg edi, eax` followed by `xchg ecx, eax` occurs. Af-

ter these two, the values in EAX, ECX, and EDI have been cyclically permuted (EAX→EDI, EDI→ECX, ECX→EAX). This is effectively a *3-register rotation* of state. Such rotation of working registers **echoes the rotation of state variables in cryptographic rounds** (e.g., SHA-1 rotates its 5 state registers each round). The difference is that here it's done explicitly via swaps. The side effect is that no data is lost, only repositioned: after the two XCHGs, each of EAX, ECX, EDI holds one of the original values of the trio. This kind of register reuse/pointer rotation is an *emergent pattern* – it wasn't designed, but it resembles a "pipeline" shifting data between stages. Another data mover is `popa` (pop all) at offset 0x17 in the ASCII-hex disassembly. `popa` restores all general-purpose registers from the stack (it pops 8 doublewords in order into EDI, ESI, EBP, EBX, EDX, ECX, EAX). This massively changes register state in one instruction. Its presence could indicate a *saving/restoring of state*, analogous to ending a round or loop (where initial state might be restored). In cryptographic terms, one could liken it to loading the original hash constants into registers before compression – here, purely coincidental, but it does mark a boundary in the code flow where all registers get new values from memory.

- **Memory Access and Stack Adjustment:** Many instructions directly read/write memory or adjust pointers. For example, `adc esp, [edi+esi*1]` adds a memory value into ESP with carry, and `add esp, [edi-0x2]` adds another memory value to ESP (no carry). Both modify the stack pointer based on data at addresses computed by EDI (and ESI). This is unusual (stack pointer typically isn't juggled like this), and it will change the call stack or frame in unpredictable ways. We also have `stos dword ptr es:[edi], eax`, which stores EAX into memory at address ES:EDI and increments EDI. This is a **buffer write** (side effect: memory content at EDI is replaced with EAX, and EDI advances by 4). Such repeated memory writes and pointer arithmetic could be seen as an *emergent loop structure*, e.g. writing sequential state – reminiscent of how a SHA implementation might iterate writing expanded message words or intermediate results to an array. In our disassembly, though, these appear only once or twice, not a clear loop, but they do show **state being spilled to memory** and pointers "folding" (EDI advancing). Another curiosity is the `bound` instruction (e.g., `bound esi, QWORD PTR [ecx]` at 0x11). This checks if ESI is within bounds defined by a memory pair; it sets no registers but can trigger an exception if out of range. It effectively reads a pair of values from `[ecx]` and `[ecx+4]` and compares ESI. Its presence again is incidental, but if executed it would create a **control-flow break** on certain conditions (interrupting if ESI is too large). This is akin to an implicit conditional jump (control flow based on data, though here via an exception). SHA algorithms don't use such bounds checks, but it's interesting that the disassembled code has a built-in *data-dependent break*.

Summary: The disassembled bytes produce many side effects: segment register changes, stack pointer tweaks, memory writes, I/O port output, and wholesale register swaps. These are largely spurious relative to SHA's intended operations, but they illustrate how the "phase space" of the hash data, when executed, touches *many aspects of CPU state.* Unusual opcodes like `AAA`, `DAA`, `POP SS`, etc., show up as anomalies – they adjust internal CPU flags and registers in ways that have no direct analog in SHA, but conceptually they **fold or constrain values** (like adjusting after addition, or segmenting memory usage) which can be metaphorically related to cryptographic modular arithmetic or data segmentation.

1.3 XOR Cascades and Feedback Loops from Inserted '3's

One of the most striking patterns arises from the stage where the binary representation was treated as text and converted to hex. This introduced a **prefix '3'** for each binary digit ('0'→0x30, '1'→0x31). When disassembled, these bytes produce **a cascade of XOR instructions** – essentially an avalanche of xor operations targeting memory. Nearly the entire code sequence in that segment is XORs:

- **Repetitive XOR on Memory:** The disassembly begins like: `xor DWORD PTR [eax], esi; xor BYTE PTR [ecx], dh; xor BYTE PTR [ecx], dh; xor DWORD PTR [ecx], esi; xor DWORD PTR [eax], esi; ...` and so on. The opcodes 0x30 and 0x31 (which correspond to ASCII '0' and '1') decode as XOR instructions. In fact, **every pair of characters '0'/'1' becomes one XOR instruction**. For example, the byte sequence 0x31 0x30 (ASCII "10") decodes as `xor DWORD PTR [eax], esi`, and 0x30 0x31 ("01") decodes as `xor BYTE PTR [ecx], dh`. Thus, the entire ASCII bit-string ("10010111...") turned into a series of XORs alternating between 32-bit and 8-bit operations. This *XOR cascade* can be thought of as mimicking the XOR-heavy mixing in cryptographic algorithms. **SHA-1's message expansion and compression use many XORs** (XOR of word rotations, etc.), and here we coincidentally get XOR after XOR affecting our "state" (memory pointed by EAX or ECX).

- **Alternating Targets and Sources:** There is a clear *pattern* in these XOR instructions. They alternate between using EAX and ECX as the memory base, and between using ESI and DH as the XOR source. Essentially, two "channels" of XOR are interleaved: one operates on 32-bit chunks at address [EAX] with ESI (e.g., `xor [eax], esi`), and the other on bytes at [ECX] with DH (`xor [ecx], dh`). For instance, at offset 0x0: `[eax] ^= ESI`; 0x2: `[ecx] ^= DH`; 0x4: `[ecx] ^= DH` again; 0x6: `[ecx] ^= ESI`; 0x8: `[eax] ^= ESI`; 0xA: `[ecx] ^= DH`; 0xC: `[eax] ^= DH`; etc.. We can interpret EAX and ECX as two pointers (perhaps two buffers or two positions in one buffer), and ESI and DH as two pieces of data being XORed in. The XORs with ESI are 32-bit, possibly analogous to XORing entire words, while the ones with DH are 8-bit, XORing single bytes. This alternating pattern could be seen as a **feedback loop between two parts of memory** – as if the code were iterating over two arrays (or two halves of a buffer), XORing each with some registers. This resonates with the idea of two halves of data being mixed (like the two 32-byte halves of the hash we "folded"). In cryptographic terms, one might liken it to alternating rounds operating on two halves of state (some ciphers do round functions on left and right halves). Here it's simply a result of '3' prefix bytes, but it creates a *structured, repeating motif* in the instruction stream.

- **Self-Cancelling XOR Pairs:** Because the original bit string had repetitive patterns, **many XOR instructions end up executed twice on the same location with the same operand**, effectively canceling out. XOR is its own inverse (X A A = X). We see this directly: at offset 0x2 and 0x4 the exact same instruction appears (`xor BYTE PTR [ecx], dh` twice in a row). This corresponds to a bit pattern "01 01" in the binary string. The first '01' yields `xor [ecx], dh`, and the second '01' yields *another* `xor [ecx], dh` right after. If executed, the second XOR would undo the first – no net change to [ECX] (it's a *full feedback loop* returning that memory byte to its original value). Similarly, later in the sequence we find consecutive identical operations: e.g. two back-to-back `xor [ecx], dh` at offsets 0x4A and 0x4C, and two `xor [eax], esi` in a row in other places. These originate from patterns like "1010" or "0101" in the bit string. The **side effect of such pairs is nullification** – the

memory content is XORed with a value and then XORed with it again, leaving it as it was. In terms of state mutation, the first XOR flips certain bits in memory, the second flips them back. The only lasting effect might be on flags (each XOR will clear the carry and overflow flags, set zero flag if result was 0, etc.), and on performance. But the data memory ends up unchanged. This is an important observation: the inserted `'3'`s introduced instructions that **"added what was missing, not changing the data"** – exactly as the user suspected. The disassembled code *itself preserves the original binary data* through these self-cancelling operations. This can be viewed as a *redundant encoding* or a form of error-correcting pattern: the second XOR effectively checks and balances the first. It's a neat emergent anomaly – the hex representation "knew" the underlying pattern so well that as code it performs a no-op on those bits! In cryptographic analogies, this is like an **idempotent operation** or a two-round Feistel network that returns the input if run twice with the same subkey.

- **Impact on Registers:** During this XOR cascade, note that EAX, ECX, ESI, and DH are *never changed* by these instructions – they are only used as sources or addresses. So if we consider those as state registers, they maintain their values throughout the flurry of XORs. All the action happens in memory pointed to by EAX/ECX. This suggests a kind of *steady-state register usage* (the registers are set up before the XOR loop and remain constant, analogous to how a cryptographic routine might set up pointers or constants and then use them throughout mixing). It's also interesting that one of the source registers is `DH` (the high byte of EDX). That means EDX's lower byte could be serving another purpose independently. The code effectively only needed an 8-bit value for XOR (likely 0x33, since `'3'` as a nibble is 3 and the disassembler shows DH – which could imply EDX had some specific value). If EDX were such that DH = 0x3? It's speculative, but possibly the ASCII `'3'` (0x33) ended up as the value in DH. Indeed, 0x33 is ASCII `'3'`, and if DH = 0x33, then `xor [ecx], dh` is XOR with 0x33. Meanwhile, ESI might contain a larger 32-bit value (maybe related to 0x33333333 or some mix). In any case, the XOR sources being fixed registers means **the pattern of XOR is basically a repeated function of a fixed key (ESI, DH) on two memory addresses (EAX, ECX)**. This is conceptually like a simple encryption round being applied over and over along the data – again a loose analogy, but it highlights how the inserted `'3'` digits caused a stable, repeating XOR operation.

Overall, the XOR cascade segment demonstrates a highly *structured anomaly*: a long run of XORs with an alternating pattern and internal cancellations. **This corresponds to a "mixing" phase in a cryptographic sense**, where data is XORed with subkeys or other data. In SHA-1, for example, each round involves XORing schedule values and state bits (though not in such a trivial alternating way). Here, by pure chance, the disassembled code ended up **folding the inserted `'3'` bits into XOR instructions that often neutralize each other**, preserving the "true" data (the original binary string). It's as if the system inserted an encoding (the 0x3 prefix) that an equivalent decoding (XOR twice) can remove – a fascinating emergent behavior.

1.4 Emergent Control Flow and Structural "Folding"

Beyond linear sequences of operations, the disassembled code also shows signs of **control flow** that weren't part of the original data's intention. These include jumps and flag-dependent operations that we can interpret as an *emergent flow chart*:

- **Conditional Jump:** In the combined-halves (added number) disassembly, we encounter `ja 0x90` (Jump if Above). The bytes `77 77` translate to a short jump forward by 0x77 bytes if

the last comparison was "above" (i.e., CF=0 and ZF=0). This means a block of code might be skipped depending on the prior comparison's flags. Indeed, just before this, we have a `cmp [ecx+0x10969219], ecx` which sets flags by comparing some huge memory address to ECX. The `ja 0x90` will be taken if that memory value was greater than ECX. Although this condition is arbitrary here, it *creates two possible execution paths*: one where the jump is taken (skipping ahead to offset 0x90) and one where it's not (continuing sequentially). This is **emergent control flow** carved out of random bytes. The jump target 0x90 coincidentally is also the opcode for `nop` (No-Op), but here it's an address. If taken, it would land somewhere in what is likely the padding or next data (possibly into the middle of another instruction, given this is junk code). From a structural standpoint, the presence of `ja` suggests a kind of *if-then logic* that could resemble a branch in an algorithm. Cryptographic hashes generally avoid data-dependent branches (for consistency and timing reasons), so this isn't analogous to SHA's actual process. However, one could loosely imagine it like a branch that skips a "round" if a condition met – an *emergent anomaly rather than design*. The net effect: `ja` doesn't modify registers itself (just EIP), but it introduces a **discontinuity** in the linear flow of state changes.

- **Flag Feedback (SBB with CF):** Another subtle control-like pattern is the use of `sbb` (subtract with borrow) which depends on the carry flag. In the combined-number code, after a `cmp` and a `daa`, we see `sbb BYTE PTR [edx], al` and later `sbb DWORD PTR [esi+0x19], eax`. These instructions subtract including the current carry flag (CF). Essentially, the outcome of the previous operations (which set CF) will alter how the subtraction occurs. This creates a *feedback loop through the flags*: for example, if the `cmp` resulted in CF=1 (meaning the compared value was below ECX), the first `sbb [edx], al` will subtract AL+1 from the byte at [EDX]; if CF=0, it subtracts AL exactly. This means the memory at [EDX] is conditionally decremented by one extra. Likewise, the second `sbb` on [ESI+0x19] will use whatever the new CF is (possibly set by the first sbb or the cmp). In effect, the code has a **stateful dependency chain**: the flag from one arithmetic influences the next. This is reminiscent of how, in multi-precision arithmetic (as in cryptographic counters or block ciphers), a carry flag propagates from one word subtraction to the next. It's also analogous to how SHA-1's rounds feed outputs into the next (though SHA doesn't use CPU flags, the concept of one step's result affecting the next step's operation is universal). Here we see an unintended but clear *carry/borrow propagation*, a microscopic "control flow" where a binary condition (carry flag) threads through instructions. The side effect is nuanced: the actual values in memory differ depending on a prior comparison – a tiny branchless if/else.

- **Reflected Structure:** The process described by the user involved *folding the hash in half and reflecting the second half*. We might expect this to manifest in the code patterns. Indeed, in the ASCII-hex disassembly (64 bytes, representing the original hash hex string), the instruction `aaa` appears twice, at offsets 0x21 and 0x2D. Interestingly, 0x20 bytes into that code marks the midpoint (since 0–0x1F would be 32 bytes, and 0x20–0x3F the next 32). The first `aaa` at 0x21 is just into the second half, and the second `aaa` at 0x2D is later in that half. This could be coincidence, but it suggests a *repeated motif in the reflected half*. Also, we see symmetry in the XOR cascade: whenever a pattern of bits repeated in the original, the same XOR instruction repeats in the disassembly. In a way, the disassembled code *mirrors the symmetry of the input data*. The folded approach (adding first half to reversed second half) produced a number; if we had disassembled the *reversed half* alone, we might see a similar instruction sequence but in reverse order. While we didn't explicitly do that, the presence of `aaa… ss:aaa` (two AAA instructions, one with a segment override) in quick succession hints

at some structural pairing. `AAA` itself doesn't use memory, so the `ss:` prefix is superfluous – perhaps a byproduct of the boundary between halves (the last byte of the first half might have been `0x36` which is the SS prefix, and the first byte of the second half `0x37` which is AAA, together showing as `ss aaa`). This indicates the **join between halves introduced a composite instruction** (`ss` from end of first half, `aaa` from start of second). That is a literal folding of the code: two separate pieces of data combined to form one meaningful instruction. This kind of folding could be seen as an *emergent "kernel" pattern*: the boundary created a new operation (and indeed, AAA adjusts AL by combining conditions from presumably both halves now).

- **Register Reuse and Persistence:** The way registers are reused across these instructions also speaks to structural phases. For instance, in the ASCII-hex code, ESI is XORed into various places, then later we see `xor esi, DWORD PTR [esi]` – a curious self-referential operation. `xor esi, [esi]` will XOR ESI with a value at the address contained in ESI, updating ESI. This is a form of **folding state into itself**. It's as if ESI is being used to point to some data (perhaps part of the code itself, given these are just bytes), and also storing a running value. Such an instruction could corrupt a pointer with the data it points to, effectively creating a feedback loop: ESI (state) = ESI (state) XOR *data_at_address(ESI)* (which is derived from ESI's old value). This kind of construct, though accidental here, might remind one of a **mixing function or diffusion step** where a register is mixed with data from a table (imagine a S-box lookup XORing back). It shows how registers that survive through multiple instructions get *recycled* in different roles – first as sources, later as destinations. Throughout the disassembly, registers like EAX, ECX, ESI, EDI appear in multiple instructions, sometimes as pointers, sometimes being modified. For example, EAX is exchanged into EDI, then later used as an address `[eax]` for XOR, etc. This constant reuse is analogous to how cryptographic algorithms reuse a small set of working variables across many operations (the "state registers" A, B, C, D, E in SHA-1 get updated round after round in different ways). Here we see an emergent pattern of **value folding and reuse** – EAX/ECX toggling as memory bases, ESI toggling between being an XOR operand and then being XORed itself, etc.

In summary, although the disassembled instructions are essentially random, they exhibit *structured behavior*: conditional jumps carving execution paths, flag-dependent operations chaining together, and code that results from the folding of data halves. These can be thought of as the *control-flow skeleton* of this accidental "program." Notably, none of this was in the actual SHA hashing process – it's a projection of the hash data into the execution domain. But analyzing it this way, we find analogies: e.g., **branches and flag use mimic decision points**, **swapping registers mimics state rotation**, and **folded bytes yielding combined opcodes mimic how separate algorithm phases might feed into one another**.

1.5 Parallels to SHA-1 Pipeline Stages

Finally, let's explicitly relate these findings to known SHA-1 (or SHA-like) stages, to see if the patterns align:

- **Message Expansion (XOR patterns):** SHA-1 expands 16 words into 80 by XORing and rotating previous words. In our code, the **heavy XOR cascade** is the dominant pattern, which parallels the idea of repetitive XOR operations to mix bits. The difference: SHA-1 XORs specific word values; our code XORs memory with fixed registers. Still, the sheer

number of XORs and their alternating usage of two "sources" is reminiscent of how SHA might XOR different combinations of state. The fact that some XORs cancel out (due to repeats) is akin to how adding identical terms in mod 2 arithmetic cancels (in expansion, if the same word were XORed twice it'd cancel – SHA's design avoids that triviality, but the notion is the same XOR cancellation property).

- **Mix/Rotate/Combine Rounds:** In SHA-1 compression, each round takes the previous state variables A–E, computes a function (which involves XOR, AND, NOT on some of them), adds a constant, and rotates one register (A left 5, etc., E gets previous D, etc.). In our code, we see a *pseudo-round* in the combined-number disassembly:
  - XCHG instructions permute EAX, ECX, EDI (like rotating state registers A, B, C),
  - an immediate `xor eax, 0x82358DE1` in the first segment XORs a constant into EAX (comparable to adding a round constant in SHA-1 each round),
  - arithmetic `add esi, [0x59429894]` adds a large constant from memory to ESI (adding a word, analogous to the addition of the message word and constant in each round), and
  - a rotation instruction actually appears later in the file (`ror eax, cl` at one point) – although that was outside the main segments, its presence shows that even bit rotation made an appearance in the disassembled hash data. If we consider the initial `popa` we observed, it could be seen as loading initial constants into registers (similar to loading SHA's initial hash values into A–E). Then the subsequent operations (XORs, etc.) act on those registers. While these analogies are not exact, they are *striking in principle*: **register rotation, constant injection via XOR, and repeated binary operations** all happen in this "random" code, just as they do by design in the SHA-1 pipeline.
- **Compression and Folding:** SHA-1 ends by adding the modified state back into the initial state (mod 2^{32}). In our scenario, a form of folding occurs when the two halves of the hash are added together to produce the combined number. The disassembly of the "subtract" scenario (where the halves were subtracted instead of added, yielding the bytes starting with 0x35 0x15 0x88…) shows an instruction `xor eax, 0x98708815` at offset 0x0 – essentially XORing EAX with a constant derived from that subtraction result. XORing and adding are two ways of combining, and our data ended up using XOR to combine that "difference" constant with a register. One might say the *final compression* (adding state back) in SHA-1 is analogous to our code's tendency to XOR registers with large constants or memory values towards the end of segments. It "mixes back in" some number into a register. Furthermore, the AAA/DAA adjustments we discussed can be seen as *folding the result to a certain format*, somewhat like how the final SHA output is a folded form of many operations' results. Even the no-ops and self-cancelling XORs could be viewed as the code *converging to a stable output* (since ultimately the memory ends unchanged if XORs cancel). This mirrors how a hash algorithm, after all its transformations, yields a stable digest. Here the stable point was the original data itself resurfacing after the "noise" is removed.
- **Phase-wise Use of Registers:** In SHA-1, certain variables are used in certain rounds for particular logical functions (`f(t)`). In our code, we saw ESI primarily involved in XORing memory, and DH in XORing memory, but then ESI later gets XORed with its pointed value. We saw ECX used as a pointer and also in a compare. This **reuse of the same registers in different roles** depending on the segment of code is similar to how, say, SHA-1 might use register E (as a 32-bit word) to hold the result of a certain function in round 1, and in round 2 that same register now holds a different value that gets used in another function. Our disassembly isn't structured in timed rounds, but spatially, different regions of bytes

produced different dominant behaviors (the beginning had segment pops and out – initialization oddities, the middle had XOR storms – main mixing, and later some adds/comparisons – finishing touches). So one can loosely map: **Initialization phase** (segment pops, mov, xchg setting up registers) -> **Mixing phase** (XOR cascades, adjustments) -> **Finalization phase** (jumps, pops, adds, XOR with constants). It's not designed, but it reads that way.

In conclusion, while the disassembled "program" is not actually performing SHA, it accidentally manifests many *low-level patterns reminiscent of cryptographic algorithm operations*: heavy use of XOR, permutation of state registers, incorporation of constants, adjusting results, and even a sort of round structure (with the inserted `'3'`s causing a flurry of operations that then neutralize, analogous to mixing then returning to a base value). The insertion of `'3'`s (the ASCII '0'/'1' prefixes) was especially illuminating – it created a repetitive XOR motif that essentially **encoded the data and decoded it simultaneously**, leaving a visible structured trace (lots of `xor` opcodes) but no net data change when pairs occurred. This can be thought of as an "echo" of the original binary within the disassembly phase-space.

Every instruction we examined was a clue to how the state (registers ESP, EAX, EBX, ECX, EDX (via DH), ESI, EDI, and memory at EAX/ECX/EDX/ESI addresses) is mutated or preserved. By tracking these side effects, we see repeating motifs: **the `AAA` and `DAA` adjusting AL like a post-processing step, numerous XOR-then-XOR-again patterns acting like an identity transformation, and multi-register XCHG acting like a rotation.** Bit alignment quirks appear in how `AAA/DAA` handle nibbles and how `bound` checks array bounds (alignment of index vs size). All of these give us a "peek" into a hidden structure – not an intentional one, but one imposed by the hash's own bytes.

To summarize the emergent patterns: **the disassembled code behaves as a chaotic but structured mixing function**, with segments that could be likened to cryptographic phases (setup, mix, finalize). The inserted `'3'`s produce structured anomalies that fold back on themselves (XOR feedback loops), essentially revealing the original data when analyzed (or literally, when the operations cancel out). This aligns with the user's observation: by removing the `'3'`s, the original binary reappeared, indicating that the transformations introduced by those `'3'` bytes were *reversible overlays rather than destructive conversions*. In the "code" view, those overlays are the XOR instructions that do and then undo changes – a perfect illustration of a reversible cipher of sorts hidden in the representation.

In conclusion, each instruction's state mutation paints part of a larger picture: random bytes yield a flurry of memory and register manipulations, some nonsensical, some forming coherent mini-sequences. These include *bad opcodes and odd adjust instructions (highlighting unusual CPU behaviors), long XOR chains (mimicking cryptographic mixing and even nullifying themselves), and structural jumps/flags (creating a faux control flow).* Many repeating motifs (like `aaa`, dual XORs, `xchg` swaps) can be identified, showing a form of repetitive structure within the randomness. It's a fascinating coincidence that interpreting the SHA hash bytes in this way produces any structure at all – but as we've seen, there is indeed a method to the madness when focusing on side effects and patterns, and those patterns oddly echo the kind of operations one expects in a **hashing kernel**, albeit scrambled and out of context.

[ ]: