Machine code format explanation

> cpu undersatbd machine code is machine code is in 0101 format typeor soem other format

Excellent question, Vipul 👏 — and yes, you're right to think of **machine code** as being related to **0101 (binary)**.

Let's go step by step very simply 👇

🧠 1. CPU understands only machine code

Your CPU (like Intel i5, M1, etc.) can **only understand machine code**, which is just **a sequence of binary 1s and 0s**.

Example:

```
10110000 01100001
```

This could mean something like:
"Move the number 97 into the register AL."
...but **the CPU sees only 1s and 0s**, not text.

⚙️ 2. What machine code looks like

Machine code = **binary instructions** for your CPU's specific architecture (like x86, ARM, etc.)

Each instruction tells the CPU to do a very low-level task like:

- Move data from memory to a register
- Add two numbers
- Jump to another instruction
- Compare values

Example (for Intel x86 CPU):

Assembly (human-readable)	Machine Code (binary)
`MOV AL, 61h`	`10110000 01100001`

So the **binary form** (`10110000 01100001`) is what CPU **actually executes**.

💬 3. So is machine code always 0101 type?

✅ Yes — **machine code is always binary (0s and 1s)**.
BUT we humans often **write or view it in hexadecimal** because binary is too long.

Example:

- Binary: `10110000 01100001`
- Hexadecimal: `B0 61`
  (both are the same, just shorter in hex)

🧩 4. How CPU reads it

When a program runs:

1. The OS loads the binary machine code into memory.
2. CPU's **instruction decoder** reads one instruction at a time (like `10110000 01100001`).
3. CPU executes it — moves data, adds numbers, etc.

So machine code is **pure binary data** that matches your CPU's "language."

⚡ 5. Summary

Concept	Description
Machine code	0s and 1s instructions that CPU executes directly
Format	Binary (but humans use Hex to read/write it easily)
Architecture specific	Yes (x86, ARM, etc. have different instruction sets)
CPU role	Reads binary instructions and executes them physically

---

If you want, I can **show you a real example** — like a simple "add two numbers" program in assembly and its **machine code translation** so you can see exactly how 0101 patterns look in real life.
Would you like me to show that?

> yes show example ad after that aslo tell me that in js code when a programs runs than it first converted to ats than byte code than machien code than how that machibe code loks in that case and will it be 01 in that case or that machine code furter changed to 01 at some other level

Perfect 👏 Vipul — this is *exactly* the kind of deep curiosity that helps you truly understand how computers run code.

Let's go step-by-step —
first we'll see what **machine code actually looks like**,
then we'll connect it to **JavaScript's pipeline** (AST → bytecode → machine code).

🧩 Part 1 – Real example of machine code (Add two numbers)

Let's take the world's simplest program in **assembly language** (human-friendly form).

🧠 Example (x86 Assembly)

```asm
MOV AL, 5     ; move 5 into register AL
ADD AL, 3     ; add 3 to AL
```

Meaning: AL = 5 + 3

⚙️ How it looks in machine code (binary + hex)

Assembly	Machine Code (binary)	Machine Code (hex)	What it does
`MOV AL, 5`	`10110000 00000101`	`B0 05`	Move 5 → AL
`ADD AL, 3`	`00000100 00000011`	`04 03`	Add 3 to AL

💡 The CPU literally receives:

```
10110000 00000101 00000100 00000011
```

That's **machine code = 0s and 1s**.
Your CPU fetches each instruction, decodes it, and executes it electrically.

⚙️ Part 2 – What happens in JavaScript

Now let's connect this to **V8 (Chrome/Node.js engine)**.

Suppose you run:

```js
let a = 5 + 3;
console.log(a);
```

Here's the full journey 👇

🧾 Step 1: JS Source Code

Your JS text (`let a = 5 + 3`) → characters → parsed by V8.

🌳 Step 2: AST (Abstract Syntax Tree)

The parser converts it into an internal tree like:

```yaml
VariableDeclaration
 └─ Identifier: a
 └─ BinaryExpression (+)
```

```
    ├─ Literal: 5
    └─ Literal: 3
```

This is *still abstract data*, not executable.

⚡ Step 3: Ignition Interpreter → Bytecode

V8's **Ignition** turns AST into *bytecode*:

```css
LdaSmi [5]
AddSmi [3]
Star r0
```

Each of these is a **V8-specific instruction**, not real CPU code yet.

Bytecode = **a compact intermediate format** for the JS engine itself.
It's not 0/1 binary for your CPU — it's *numbers representing engine instructions*.

🚀 Step 4: TurboFan → Machine Code

V8's **TurboFan JIT compiler** then takes hot (frequently executed) bytecode and converts it into **machine code for your CPU**.

👉 At this point, V8 emits *real* binary instructions like:

```
10110000 00000101 00000100 00000011
```

(similar in style to our earlier example)

✅ This machine code is **pure 0s and 1s** —
exactly the language your Intel/ARM CPU executes.

🧠 Step 5: CPU executes it

The CPU executes those 0/1 instructions directly.
No further conversion happens after this — this is the final physical form of code.

🧾 Summary

Stage	Who makes it	Example output	Type
JS source	You	`let a = 5 + 3`	Text
AST	Parser	Tree structure	Data structure
Bytecode	Ignition	`LdaSmi [5]`	V8 internal format
Machine code	TurboFan	`10110000 00000101…`	**0 and 1 binary**
Executed by	CPU	—	Electrical signals

⚡ So answering your last line:

> when it becomes machine code, will it be 01 or later changed to 01 at some other level?

✅ **When V8 generates machine code — it is already 0s and 1s.**
That *is* the form the CPU runs.
No later conversion happens.
It's already binary patterns stored in memory, fetched and executed as electrical signals.

Would you like me to **visually show** this full flow (a diagram from JS → AST → Bytecode → Machine Code → CPU)?
It'll make the whole pipeline crystal clear.