Each machine code instruction in cpu starting in 1954 had a set number of cycles

Since the mid-1950s, CPU architectures have used **fixed instruction timing**, where each machine code instruction requires a specific, documented number of clock cycles to complete.

Evolution of Instruction Cycles (Starting 1954)

In 1954, the introduction of machines like the **IBM 704** established a standardized "machine cycle" as the fundamental unit of time for operations.

- **Fixed Cycles per Instruction**: Every opcode in an instruction set was designed to take a set number of these cycles. For example, a basic "Add" might take 2 cycles, while a complex "Multiply" could take 20.
- **Deterministic Timing**: Because there were no complex features like out-of-order execution or modern caching, programmers could calculate the *exact* execution time of a program by summing the cycles of its individual instructions.
- **The IBM 704 Example (1954)**:
    - **Machine Cycle**: 12 microseconds.
    - **Basic Addition**: 2 cycles (24 microseconds).
    - **Floating-Point Multiply**: ~17 cycles (204 microseconds).
    - **Basic Multiply**: 20 cycles (240 microseconds).

The Role of the "Machine Cycle"

Early CPUs followed a strict, non-overlapping **Fetch-Decode-Execute** sequence:

1. **Fetch**: Retrieve the instruction from memory (often 1 cycle).
2. **Decode**: The control unit interprets the opcode (often happens during fetch or execution).
3. **Execute**: The ALU performs the operation. This stage varied the most in length depending on the complexity of the math (e.g., division vs. addition).

Why Cycles Varied Between Instructions

- **Data Movement**: Instructions requiring memory access (Load/Store) took more cycles because they had to wait for the slower magnetic core or drum memory.
- **Arithmetic Complexity**: Adding two numbers is a single-step logic gate operation, but multiplication and division required multiple internal shifts and additions, consuming many more cycles.

This predictable timing remained the standard for decades until **instruction pipelining** and **superscalar** designs (where multiple instructions can complete in a single cycle) became common in the late 20th century.

Back on the IBM 704 in 1954, an instruction loaded as one word.

Correct—on the **IBM 704**, both data and instructions were stored in **36-bit words**. Unlike its predecessor (the 701), which used 18-bit half-words for instructions, the 704 expanded instructions to fill the entire 36-bit word to accommodate more complex features like **indexing** and **floating-point arithmetic**.

The IBM 704 Instruction Layout

The 36-bit instruction word was strictly partitioned into specific fields that the CPU decoded during its cycles:

- **Prefix (3 bits)**: Specified the general class of the instruction (the main part of the opcode).
- **Decrement (15 bits)**: Often used as a constant value (immediate operand) or for controlling loops.
- **Tag (3 bits)**: A bitmap indicating which of the **three index registers** should be used to modify the address.
- **Address (15 bits)**: The base memory address of the operand.

Why the Full Word Mattered

- **Addressing Power**: The 15-bit address field allowed the 704 to directly address up to **32,768 words** of memory, a massive leap for 1954.
- **Indexing**: The 3-bit "Tag" allowed for automatic address modification (subtracting index register values from the base address), which made processing arrays much faster than on previous machines.
- **Scientific Foundation**: This 36-bit architecture became the standard for IBM's "scientific" line, including the 709 and 7090, and directly influenced why **FORTRAN** originally limited variable names to 6 characters (as six 6-bit characters fit perfectly into one 36-bit word).

Physically, a 36-bit instruction on the **IBM 704** was represented on an 80-column punched card using one of two primary "binary" formats. Because the card reader for the 704 was essentially a 72-bit device, it could only read **72 of the 80 columns** in a single pass.

1. Row Binary (The Compact Standard)

This was the most common format for storing compiled "object code." The 704 would read the card **row by row** rather than column by column.

- **Layout**: Each of the 12 rows on the card was treated as two 36-bit words side-by-side.
- **Capacity**: A single card could hold **24 instructions** (12 rows times 2 words per row).
- **Organization**: Columns 1–36 held the "left" word of each row, and columns 37–72 held the "right" word.
- **The "Safety" Zone**: Columns 73–80 was ignored by the machine and were almost always used for **human-readable sequence numbers**. If you dropped your deck, you could use a mechanical sorter to put them back in order using these columns.

2. "Chinese Binary" (Column Binary)

In this alternate format, the card was read **column by column**.

- **Layout**: Since a card column has 12 punch positions, exactly **three columns** were required to store one 36-bit word.
- **Usage**: This was often used for smaller "boot loaders" or diagnostic programs where the loader needed to process data one word at a time rather than filling whole rows.

3. Symbolic Form (FORTRAN / Assembly)

Before the instructions were turned into binary "holes," they existed as text on **source cards**.

- **One Card = One Statement**: Unlike the binary format, a single card represented only **one** line of code (e.g., `ADD 100`).
- **Fixed Fields**: Programmers used **coding sheets** to ensure the instruction was punched in the correct columns (e.g., Label in cols 1–5, Instruction in cols 7–72).

A program's life cycle began as a massive deck of **symbolic cards**, which the 704's compiler would process and output as a much smaller, more compact deck of **row-binary cards**.

Patching a binary card on the IBM 704 was a delicate task because, unlike symbolic source cards, binary cards contained actual machine code instructions where every hole represented a specific bit (0 or 1) in a 36-bit word.

If a programmer discovered a minor bug in their binary deck, they could often "patch" it manually to avoid a full re-assembly, which could take hours of waiting for a turn on the machine.

1. Adding "Holes" (Changing 0s to 1s)

If the correction required changing a 0 bit to a 1, the programmer used a **manual hand punch** (a small, handheld tool about the size of a stapler).

- **The Process**: They would carefully align the card and punch a single hole in the exact row and column corresponding to the bit they needed to change.
- **Risks**: One slight misalignment would ruin the instruction or create a different bit pattern entirely. Because binary cards were read as **Row Binary** (two 36-bit words per row), a single mistake could corrupt two instructions simultaneously.

2. Filling "Holes" (Changing 1s to 0s)

Changing a 1 to a 0 was more common and involved physically "erasing" a hole.

- **Tape and Glue**: Programmers would use a small piece of **adhesive tape** or "mending tape" to cover the hole.
- **The "Chad" Method**: In some cases, they would take a "chad" (the tiny rectangular bit of paper punched out from another card) and glue it into the hole they wanted to close.
- **The Friction Risk**: These patches had to be perfectly flush. If a piece of tape or a glued-in chad was too thick, it could cause the card to jam in the high-speed 704 card reader, potentially shredding the card or damaging the machine's mechanical brushes.

3. The "Patch" Card Strategy

For more complex changes, programmers rarely modified existing cards. Instead, they used a **transfer card** or a **patch card**.

- **The Detour**: They would find an unused area of memory and punch a new card containing the corrected code.
- **The Jump**: They would "patch" the original binary card by changing one instruction to a **"Transfer" (TJP or TRA)** instruction. This forced the CPU to jump to the new code, execute it, and then jump back to the original sequence.

This manual tinkering was an essential skill; a skilled 704 programmer knew their binary opcodes by heart so they could read and edit their cards just by looking at the holes.

The machine is said to have weighed 10 tons. How many were produced, at what RRP, and for how long?

IBM 704 - Wikipedia to view Wikipedia's page

** End of Page