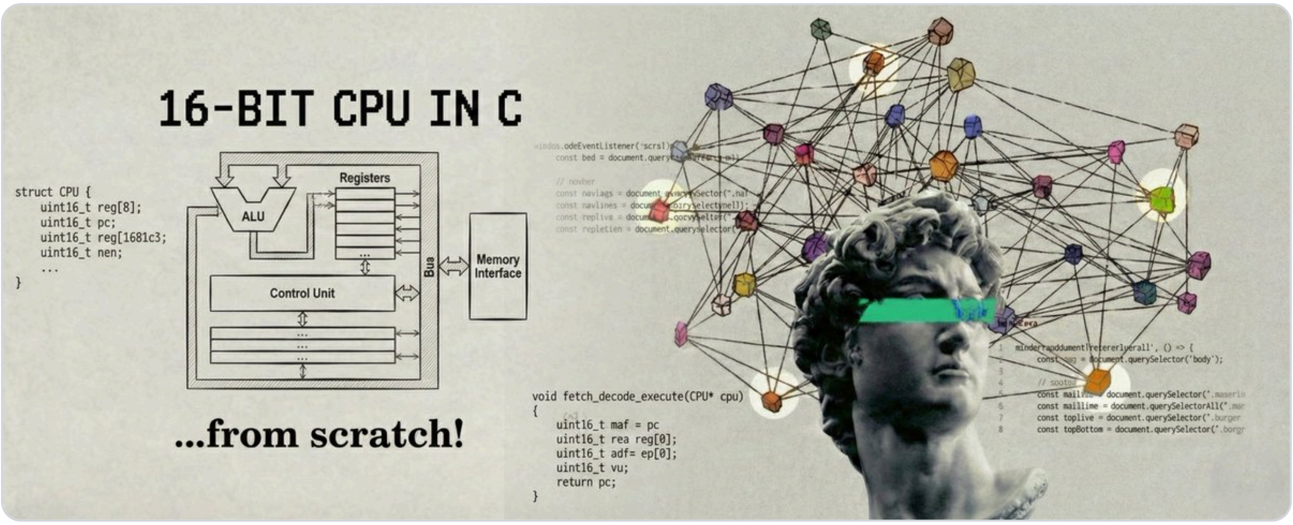
Building a 16-Bit CPU from Scratch in C (Step-by-Step)

vixhaℓ @TheVixhal

23 février 2026 à 19:32 Source: fxtwitter

https://x.com/TheVixhal/status/2026002315371745671

https://t.co/tmMaGWK4CM



```
struct CPU {
    uint16_t reg[8];
    uint16_t pc;
    uint16_t reg[1681c3];
    uint16_t nen;
    ...
}
```

```
void fetch_decode_execute(CPU* cpu)
{
    uint16_t maf = pc
    uint16_t rea reg[0];
    uint16_t adf= ep[0];
    uint16_t vu;
    return pc;
}
```

Building a 16-Bit CPU from Scratch in C might sound intimidating, but it's one of the most rewarding projects you can undertake as a programmer. In this article, I'll walk you through every step of creating a functional 16-bit CPU emulator in C.

Prerequisites:

- Basic C programming knowledge
- A C compiler (GCC, Clang, or MSVC)
- Curiosity and patience

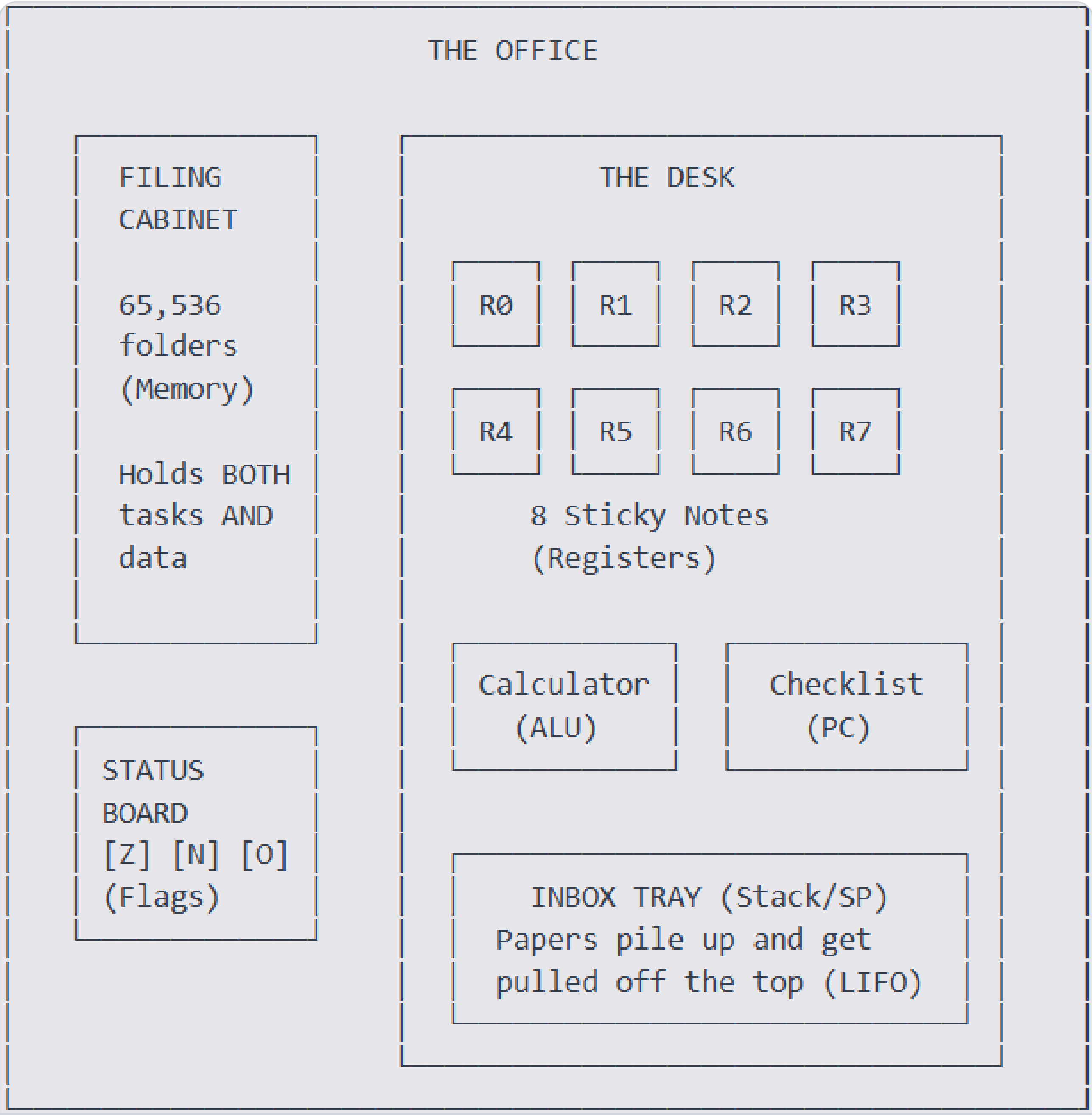
Before we write any code, let me introduce you to the analogy that will carry us through the entire article.

Complete Code: https://github.com/vixhal-baraiya/16bit-cpu

(Don't forget to ⭐ star the repository if you found this helpful!)

The Office Analogy

Imagine a **single office worker sitting at a desk**. This worker is our CPU. Everything about a CPU maps perfectly to this worker's office:



The Office Analogy: Your Mental Model for the Entire Article

Office Element	CPU Component	What It Does
The desk (8 sticky notes)	**Registers** (R0–R7)	Tiny, instant-access workspace for numbers you're working with *right now*
The filing cabinet (65,536 folders)	**Memory** (64 KB)	Huge storage that holds both program tasks and data. It is slow to access.
The calculator	**ALU**	Does the actual math: add, subtract, AND, OR, etc.
The task checklist	**Program Counter (PC)**	Points to which task you're doing right now, then moves to the next one
The status board (3 lights)	**Flags register**	Records what just happened: "Was the result zero? Negative? Overflow?"
The inbox tray	**Stack Pointer (SP)**	A pile of papers on the desk corner where you push new ones on top and pull from the top. Used for saving and restoring notes and remembering where you were.

Here's the key rule: The worker can **only do math on numbers that are on the desk** (in registers). To work with something from the filing cabinet (memory), you first have to walk over, grab the folder, and bring it to your desk. This is why registers exist. They're the fast workspace.

Each chapter that comes next connects to a different part of this office. Let’s put it all together step by step.

Chapter 1: What Does the Office Worker Actually Do All Day?

Our office worker has exactly **one routine**, repeated forever:

1. **Look at the checklist** → read the next task (FETCH)
2. **Understand the task** → “Add sticky note #0 and sticky note #1” (DECODE)
3. **Do the task** → grab the calculator, punch in the numbers (EXECUTE)
4. **Move to the next task** on the checklist (ADVANCE PC)

That’s it. That’s a CPU. This loop of fetch, decode, and execute is the heartbeat of every processor ever made, from the Intel 4004 released in 1971 to Apple’s M5.

~~~
```markdown
    ┌──────────────────────────────────────────────┐
    │  WORKER'S DAILY ROUTINE                      │
    │                                              │
    │  1. Read next task from checklist  [FETCH]   │
    │               │                              │
    │  2. Understand what it says        [DECODE]  │
    │               │                              │
    │  3. Do the work                    [EXECUTE] │
    │               │                              │
    │  4. Cross it off, move to next     [ADVANCE] │
    │               │                              │
    │               └──── repeat until "GO HOME" ──┤
    └──────────────────────────────────────────────┘
```
~~~

The worker stops only when a task says **“Go home.”** That is our HALT instruction.

Now let's design exactly how this office operates.

Chapter 2: Designing the Office Blueprint (CPU Architecture)

Before hiring our worker, we need to design the office. This is the **architecture,** the blueprint that defines every rule of how our CPU operates.

The Full Office Blueprint

Office Feature	Specification
Sticky notes on desk	8 notes (R0–R7), each holds a 16-bit number
Filing cabinet	65,536 folders (64 KB of memory), holds tasks AND data
Task checklist	Starts at task #0 (PC = 0x0000)
Status board	3 lights: Zero, Negative, Overflow
Inbox tray	Starts at the top shelf (SP = 0xFFFF, stack grows downward)

The Status Board (Flags)

On the wall next to the desk, there's a board with three indicator lights. Every time the worker uses the calculator, these lights update automatically:

- **Z (Zero)**: Lights up when the calculator's answer is exactly 0. It is used for equality checks
- **N (Negative)**: Lights up when the answer is negative (bit 15 is set in two's complement)
- **O (Overflow)**: Lights up when the numbers were too big and “wrapped around”

These status lights are how the worker makes **decisions**. More on that when we get to conditional jumps.

The Inbox Tray (Stack)

In the corner of the desk sits a spring-loaded tray called the **inbox tray**. The worker can:

- **Push** a sticky note onto the tray (the tray sinks down one slot)
- **Pop** the top note off (the tray springs up one slot)

This is a **stack:** last in, first out (LIFO). It's critical for two things:

1. **Saving and restoring** sticky note values temporarily
2. **Remembering where you were** when the boss says “go do this other thing and come back” (function calls)

The tray starts at the top (SP = 0xFFFF) and grows **downward.** Each push decreases the position, each pop increases it. This convention matches real x86 CPUs.

Code: The Office in C

Now that the blueprint is clear, translating it to code is just writing down what we described:

```c
// cpu.h: The office blueprint

#ifndef CPU_H
#define CPU_H

#include <stdint.h>
#include <stdbool.h>

#define NUM_REGISTERS  8      // 8 sticky notes on the desk
#define MEMORY_SIZE    65536  // 64 KB filing cabinet

// Status board lights
#define FLAG_ZERO      (1 << 0)  // "The answer was zero"
#define FLAG_NEGATIVE  (1 << 1)  // "The answer was negative"
#define FLAG_OVERFLOW  (1 << 2)  // "The numbers wrapped around"

typedef struct {
    uint16_t registers[NUM_REGISTERS];  // The 8 sticky notes (R0–R7)
    uint16_t pc;       // Task checklist position
    uint16_t sp;       // Inbox tray position (stack pointer)
    uint16_t flags;    // Status board (Z, N, O lights)
    uint8_t  memory[MEMORY_SIZE];  // The filing cabinet
    bool     halted;   // Has the worker gone home?
    uint64_t cycles;   // How many tasks completed
} CPU;

#endif
```

uint16_t = a 16-bit number (each sticky note). uint8_t = a single byte (each folder in the filing cabinet is one byte wide).

Chapter 3: The Task Vocabulary (Instruction Set)

Our worker needs a fixed set of tasks they know how to perform. You can't write “make me a coffee” on the checklist. The worker only understands a specific **vocabulary of 23 task types**.

~~~
```c
// opcodes.h: The 23 task types our worker understands

#ifndef OPCODES_H
#define OPCODES_H

typedef enum {
    OP_NOP  = 0x00,  // Do nothing
    OP_LOAD = 0x01,  // Write a number on a sticky note (2-word: next word is the number)
    OP_MOV  = 0x02,  // Copy one sticky note to another
    OP_ADD  = 0x03,  // Add two sticky notes
    OP_SUB  = 0x04,  // Subtract two sticky notes
    OP_AND  = 0x05,  // Bitwise AND
    OP_OR   = 0x06,  // Bitwise OR
    OP_XOR  = 0x07,  // Bitwise XOR
    OP_NOT  = 0x08,  // Bitwise NOT (flip all bits)
    OP_SHL  = 0x09,  // Shift bits left
    OP_SHR  = 0x0A,  // Shift bits right
    OP_CMP  = 0x0B,  // Compare (subtract but only update status board)
    OP_JMP  = 0x0C,  // Jump to task #N
    OP_JZ   = 0x0D,  // Jump IF Zero light is on
    OP_JNZ  = 0x0E,  // Jump IF Zero light is off
    OP_JN   = 0x0F,  // Jump IF Negative light is on
    OP_LDR  = 0x10,  // Load from filing cabinet folder into sticky note
    OP_STR  = 0x11,  // Store sticky note into filing cabinet folder
    OP_PUSH = 0x12,  // Push onto inbox tray
    OP_POP  = 0x13,  // Pop from inbox tray
    OP_CALL = 0x14,  // Bookmark and jump to subroutine (2-word)
    OP_RET  = 0x15,  // Return from subroutine (pop bookmark)
    OP_HALT = 0x16,  // Go home (stop CPU)
} Opcode;

#endif
```
~~~

With just these 23 tasks, our worker can compute anything a modern CPU can do.

Chapter 4: Writing Tasks on the Checklist (Instruction Encoding)

Each task on the worker's checklist is a **16-bit number**. We have to squeeze the task type, which sticky notes to use, and any extra info into those 16 bits.

The Encoding Formats

With 23 task types, we need a **5-bit opcode** (supports up to 32 types). That leaves 11 bits for everything else. We use three formats:

Format R: Register operations (ADD, SUB, MOV, etc.):

~~~~
```markdown
┌───────────────┬─────────────┬─────────────┬──────────────────┐
│ TASK TYPE     │ NOTE #1     │ NOTE #2     │ EXTRA            │
│ (5 bits)      │ (3 bits)    │ (3 bits)    │ (5 bits)         │
│ Opcode        │ dst reg     │ src reg     │ imm5 (0–31)      │
└───────────────┴─────────────┴─────────────┴──────────────────┘
  Bits 15-11      Bits 10-8     Bits 7-5       Bits 4-0
```
~~~~

Format J: Jump operations (JMP, JZ, JNZ, JN):

~~~~
```markdown
┌───────────────┬────────────────────────────────────────────────┐
│ TASK TYPE     │ JUMP TARGET ADDRESS                            │
│ (5 bits)      │ (11 bits, range 0–2047)                        │
└───────────────┴────────────────────────────────────────────────┘
  Bits 15-11                  Bits 10-0
```
~~~~

Format W: Wide immediate (LOAD, CALL) uses **two words**:

~~~~
```markdown
Word 1: ┌─────────────┬─────────────┬──────────────────────┐
        │ OPCODE      │ DST REG     │ (unused)             │
        │ (5 bits)    │ (3 bits)    │ (8 bits)             │
        └─────────────┴─────────────┴──────────────────────┘

Word 2: ┌────────────────────────────────────────────────────┐
        │ FULL 16-BIT VALUE (0–65535)                        │
        └────────────────────────────────────────────────────┘
```
~~~~

Code: Encoding and Decoding

~~~
```c
// instruction.h: Reading and writing the task forms

#ifndef INSTRUCTION_H
#define INSTRUCTION_H

#include <stdint.h>
#include "opcodes.h"

// Reading a task form (decoding)
#define DECODE_OPCODE(instr)  ((Opcode)(((instr) >> 11) & 0x1F))
#define DECODE_DST(instr)     (((instr) >> 8) & 0x7)
#define DECODE_SRC(instr)     (((instr) >> 5) & 0x7)
#define DECODE_IMM5(instr)    ((instr) & 0x1F)
#define DECODE_ADDR(instr)    ((instr) & 0x7FF)

// Writing a task form (encoding): Format R
static inline uint16_t ENCODE_REG(Opcode op, uint8_t dst, uint8_t src, uint8_t imm5) {
    return ((uint16_t)(op & 0x1F) << 11) | ((uint16_t)(dst & 0x7) << 8) |
           ((uint16_t)(src & 0x7) << 5) | (imm5 & 0x1F);
}

// Writing a task form (encoding): Format J
static inline uint16_t ENCODE_JUMP(Opcode op, uint16_t address) {
    return ((uint16_t)(op & 0x1F) << 11) | (address & 0x7FF);
}

#endif
```
~~~

Example: “ADD sticky note #2 and sticky note #5”

~~~
```markdown
OPCODE = ADD  = 00011   (task type 3)
DST    = R2   = 010     (destination)
SRC    = R5   = 101     (source)
IMM5   = 0    = 00000

Combined: 00011 010 101 00000 = 0x1AA0
```
~~~

Chapter 5: The Filing Cabinet (Memory)

The filing cabinet sits across the room from the desk. It has **65,536 folders**, numbered 0x0000 to 0xFFFF. Each folder holds one byte (8 bits).

But our sticky notes hold 16-bit numbers. That equals two bytes. So storing one sticky note's value requires **two adjacent folders.**

Which Folder Gets Which Half?

When the worker stores the value 0x1234 starting at folder #0010:

~~~
```markdown
Filing cabinet:
  Folder #0010:  0x34  ← Low byte (the "small end")
  Folder #0011:  0x12  ← High byte (the "big end")
```
~~~

This is called little-endian, where the "little end" (low byte) goes in the first folder. It may feel backwards, but it is the same convention that x86 processors use.

The Filing Cabinet Holds BOTH Tasks AND Data

This is important: the same filing cabinet stores the **task checklist** (program code) AND any **data** the worker needs to store or retrieve. The bottom of the cabinet (low addresses) typically holds the program. The worker can store results higher up using STR and retrieve them with LDR.

Opening Day: Resetting the Office

When the office first opens (CPU powers on), everything starts clean:

- All sticky notes are blank (registers = 0)
- All folders are empty (memory = 0)
- The checklist starts at task #0 (PC = 0)
- The inbox tray starts at the top shelf (SP = 0xFFFF)
- All status lights are off (flags = 0)

~~~
```c
// memory.c: The filing cabinet operations

#include <string.h>
#include <stdio.h>
#include "cpu.h"

// Opening day: reset the entire office
void cpu_init(CPU *cpu) {
    memset(cpu->registers, 0, sizeof(cpu->registers));
    memset(cpu->memory, 0, sizeof(cpu->memory));
    cpu->pc = 0x0000;
    cpu->sp = 0xFFFF;
    cpu->flags = 0;
    cpu->halted = false;
    cpu->cycles = 0;
}

// Walk to the cabinet, grab two folders, combine them into a 16-bit value
uint16_t mem_read16(CPU *cpu, uint16_t address) {
    uint16_t low  = cpu->memory[address];
    uint16_t high = cpu->memory[address + 1];
    return (high << 8) | low;
}

// Walk to the cabinet, split a 16-bit value across two folders
void mem_write16(CPU *cpu, uint16_t address, uint16_t value) {
    cpu->memory[address]     = value & 0xFF;
    cpu->memory[address + 1] = (value >> 8) & 0xFF;
}
```
~~~

Chapter 6: The Calculator (ALU)

The calculator on the desk is the Arithmetic Logic Unit. It is the only part of the office that actually performs calculations. Everything else, including the filing cabinet, the checklist, and the sticky notes, exists to supply numbers to the calculator and store the results.

The worker punches two numbers in, presses an operation button, and gets a result. But the calculator also does something extra: it **automatically updates the status board** on the wall.

How the Status Board Gets Updated

Every time the calculator produces an answer:

- If the answer is **0** → the **Zero light** turns on (useful for equality: 5 **-** 5 = 0 means they were equal)

- If the answer's **top bit is 1** → the **Negative light** turns on (in two's complement, this means negative)
- If the answer **wrapped around** (e.g., 32767 + 1 = -32768) → the **Overflow light** turns on

The worker does not manually flip these lights. The calculator updates them automatically. That is exactly how real CPU hardware works.

```c
// alu.c: The calculator on the desk

#include "cpu.h"
#include "opcodes.h"
#include <stdio.h>

// The calculator automatically updates the status board after every
operation
void update_flags(CPU *cpu, uint16_t result, uint16_t a, uint16_t b,
Opcode op) {
    cpu->flags = 0;  // All lights off first

    if (result == 0)     cpu->flags |= FLAG_ZERO;      // Zero light
    if (result & 0x8000) cpu->flags |= FLAG_NEGATIVE;   // Negative light

    // Overflow: both inputs had the same sign, but result has a
different sign
    if (op == OP_ADD) {
        if ((a ^ result) & (b ^ result) & 0x8000)
            cpu->flags |= FLAG_OVERFLOW;
    } else if (op == OP_SUB || op == OP_CMP) {
        if ((a ^ b) & (a ^ result) & 0x8000)
            cpu->flags |= FLAG_OVERFLOW;
    }
}

// Press a button on the calculator
uint16_t alu_execute(CPU *cpu, Opcode op, uint16_t a, uint16_t b) {
    uint16_t result = 0;

    switch (op) {
        case OP_ADD: result = a + b;           break;
        case OP_SUB:
        case OP_CMP: result = a - b;           break;
        case OP_AND: result = a & b;           break;
        case OP_OR:  result = a | b;           break;
        case OP_XOR: result = a ^ b;           break;
        case OP_NOT: result = ~a;              break;
        case OP_SHL: result = a << (b & 0xF);  break;
        case OP_SHR: result = a >> (b & 0xF);  break;
        default:
            fprintf(stderr, "Calculator error: unknown operation 0x%X\n",
op);
            return 0;
```

~~~
    }

    update_flags(cpu, result, a, b, op);
    return result;
}
```
~~~

Chapter 7: The Worker's Daily Routine (Fetch-Decode-Execute)

This is the heartbeat of our CPU, the routine our office worker repeats for every single task.

How the Worker Makes Decisions

The CMP task is special. The worker punches two numbers into the calculator and presses subtract, but **throws away the answer**. All that matters is the status board:

~~~
```markdown
Worker reads: CMP R0, R1

R0 = 5, R1 = 5:  calculator shows 5 - 5 = 0 → Zero light ON
R0 = 7, R1 = 3:  calculator shows 7 - 3 = 4 → Zero light OFF, Negative OFF
R0 = 3, R1 = 7:  calculator shows 3 - 7 = -4 → Zero light OFF, Negative ON
```
~~~

Then the next task checks the board:

- JZ → jump if Zero light is on (they were equal)
- JNZ → jump if Zero light is off (they were not equal)
- JN → jump if Negative light is on (first was less than second)

This is how every if statement you have ever written works at the hardware level. The CPU does not have an “equals” button. Instead, it subtracts the values and checks whether the result is zero.

Two-Word Instructions

LOAD and CALL are special because they consume two slots on the checklist. The first slot contains the task form, which includes the opcode and register. The second slot holds the full 16-bit number. The worker reads both and then advances the checklist pointer by 4 bytes instead of 2.

How the Inbox Tray Works (Stack)

When the worker sees PUSH R0:

1. Move the inbox tray down one slot (SP -= 2)
2. Place the value from sticky note R0 into the tray

When the worker sees POP R0:

1. Take the top value from the inbox tray, write it on sticky note R0
2. Move the tray up one slot (SP += 2)

When the boss says CALL meeting_room (call a subroutine):

1. Push the current checklist position onto the inbox tray (so you remember where to come back)
2. Jump to the subroutine's position on the checklist

When the subroutine finishes with RET:

1. Pop the saved checklist position from the inbox tray
2. Jump back to where you were

Code: The Execution Engine

```c
// cpu_execute.c: The worker's daily routine

#include <stdio.h>
#include <inttypes.h>
#include "cpu.h"
#include "opcodes.h"
#include "instruction.h"

extern uint16_t alu_execute(CPU *cpu, Opcode op, uint16_t a, uint16_t b);
extern uint16_t mem_read16(CPU *cpu, uint16_t address);
extern void mem_write16(CPU *cpu, uint16_t address, uint16_t value);

// One tick of the clock: the worker handles one task
void cpu_step(CPU *cpu) {
    if (cpu->halted) return;

    // FETCH: read the task form from the filing cabinet
    uint16_t instruction = mem_read16(cpu, cpu->pc);

    // DECODE: read the fields on the task form
    Opcode opcode = DECODE_OPCODE(instruction);
    uint8_t dst   = DECODE_DST(instruction);
    uint8_t src   = DECODE_SRC(instruction);
    uint8_t imm5  = DECODE_IMM5(instruction);
    uint16_t addr = DECODE_ADDR(instruction);

    // Advance checklist BEFORE executing (jumps will override if needed)
    cpu->pc += 2;

    // EXECUTE: do what the task form says
    switch (opcode) {
        case OP_NOP:  break;  // Coffee break

        case OP_LOAD: {  // "Write this number on sticky note #dst" (2-
word)
            // The next word on the checklist IS the 16-bit number
            cpu->registers[dst] = mem_read16(cpu, cpu->pc);
            cpu->pc += 2;  // Skip past the number word
            break;
        }

        case OP_MOV:  // "Copy note #src onto note #dst"
            cpu->registers[dst] = cpu->registers[src];
            break;
        case OP_ADD:  // "Add note #dst and note #src, result on note
#dst"
            cpu->registers[dst] = alu_execute(cpu, OP_ADD,
                cpu->registers[dst], cpu->registers[src]);
            break;

        case OP_SUB:
            cpu->registers[dst] = alu_execute(cpu, OP_SUB,
                cpu->registers[dst], cpu->registers[src]);
            break;

        case OP_AND:
            cpu->registers[dst] = alu_execute(cpu, OP_AND,
                cpu->registers[dst], cpu->registers[src]);
            break;

        case OP_OR:
            cpu->registers[dst] = alu_execute(cpu, OP_OR,
                cpu->registers[dst], cpu->registers[src]);
            break;

        case OP_XOR:
            cpu->registers[dst] = alu_execute(cpu, OP_XOR,
                cpu->registers[dst], cpu->registers[src]);
            break;

        case OP_NOT:
            cpu->registers[dst] = alu_execute(cpu, OP_NOT,
                cpu->registers[dst], 0);
            break;

        case OP_SHL:
            cpu->registers[dst] = alu_execute(cpu, OP_SHL,
                cpu->registers[dst], imm5);
            break;

        case OP_SHR:
            cpu->registers[dst] = alu_execute(cpu, OP_SHR,
                cpu->registers[dst], imm5);
            break;

        case OP_CMP:  // Use calculator but THROW AWAY the result
            alu_execute(cpu, OP_CMP,
                cpu->registers[dst], cpu->registers[src]);
            break;
        case OP_JMP:  // "Skip to task #N on checklist"
            cpu->pc = addr;
            break;

        case OP_JZ:   // "Skip to task #N IF Zero light is on"
            if (cpu->flags & FLAG_ZERO) cpu->pc = addr;
            break;

        case OP_JNZ:  // "Skip to task #N IF Zero light is off"
            if (!(cpu->flags & FLAG_ZERO)) cpu->pc = addr;
            break;

        case OP_JN:   // "Skip to task #N IF Negative light is on"
            if (cpu->flags & FLAG_NEGATIVE) cpu->pc = addr;
            break;

        case OP_LDR:  // "Go to cabinet folder [src], bring value to note
#dst"
            cpu->registers[dst] = mem_read16(cpu, cpu->registers[src]);
            break;

        case OP_STR:  // "File note #src's value into cabinet folder
[dst]"
            mem_write16(cpu, cpu->registers[dst], cpu->registers[src]);
            break;

        case OP_PUSH:  // "Place note #dst's value on the inbox tray"
            cpu->sp -= 2;
            mem_write16(cpu, cpu->sp, cpu->registers[dst]);
            break;

        case OP_POP:   // "Take top value from inbox tray, write on note
#dst"
            cpu->registers[dst] = mem_read16(cpu, cpu->sp);
            cpu->sp += 2;
            break;

        case OP_CALL: {  // "Bookmark your place, jump to subroutine" (2-
word)
            uint16_t target = mem_read16(cpu, cpu->pc);
            cpu->pc += 2;  // Skip past the address word
            // Push return address (current PC = instruction after CALL)
            cpu->sp -= 2;
            mem_write16(cpu, cpu->sp, cpu->pc);
            cpu->pc = target;
```

~~~
            break;
        }

        case OP_RET:   // "Pull bookmark from inbox tray, jump back"
            cpu->pc = mem_read16(cpu, cpu->sp);
            cpu->sp += 2;
            break;

        case OP_HALT:  // "Go home"
            cpu->halted = true;
            printf("Worker went home after %" PRIu64 " tasks.\n", cpu-
>cycles);
            break;

        default:
            fprintf(stderr, "UNKNOWN TASK 0x%04X at checklist position
0x%04X\n",
                    instruction, cpu->pc - 2);
            cpu->halted = true;
            break;
    }

    cpu->cycles++;
}

// The worker keeps working until they see "Go home"
void cpu_run(CPU *cpu) {
    while (!cpu->halted) {
        cpu_step(cpu);
    }
}
```
~~~

Chapter 8: Hiring a Secretary (The Assembler)

Right now, to create a checklist, we'd have to write raw hexadecimal task forms like 0x1AA0. Nobody wants to do that.

So we hire a **secretary** (the assembler). You tell the secretary in plain English: “Add sticky note R2 and sticky note R5.” The secretary writes the formal task form: 0x1AA0.

~~~
```markdown
You say:             "LOAD R1, 1000"
Secretary writes:     0x0900 0x03E8   ← two-word task form (instruction + value)
```
~~~

The Forward-Reference Problem

What if the checklist says “Jump to the meeting task”, but the meeting task hasn't been written yet? The secretary doesn't know what position it will be at.

Solution: the secretary reads through the checklist **twice**:

- **Pass 1**: Skim through, just writing down where each labeled task will be: “The loop task starts at checklist position #20.” Importantly, the secretary knows that LOAD and CALL take **4 bytes** (two words) while everything else takes **2 bytes** (one word).
- **Pass 2**: Write the actual task forms, now knowing all label positions.

This is called a **two-pass assembler,** the same technique used by the earliest real assemblers.

Code: The Secretary (Assembler)

```c
// assembler.c: The secretary who translates plain English to task forms

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "instruction.h"

int parse_register(const char *str) {
    while (*str == ' ' || *str == '\t') str++;
    char clean[16];
    strncpy(clean, str, 15); clean[15] = '\0';
    char *comma = strchr(clean, ',');
    if (comma) *comma = '\0';
    int len = strlen(clean);
    while (len > 0 && clean[len-1] <= ' ') clean[--len] = '\0';
    if ((clean[0]=='R'||clean[0]=='r') && clean[1]>='0' && clean[1]<='7' && !clean[2])
        return clean[1] - '0';
    return -1;
}

int parse_immediate(const char *str) {
    while (*str == ' ' || *str == '\t') str++;
    return atoi(str);
}

typedef struct { char name[32]; uint16_t address; } Label;

int assemble(const char *source, uint16_t *output, int max_words) {
    Label labels[256];
    int label_count = 0, word_count = 0;

    // PASS 1: Skim through, record label positions
    // LOAD and CALL are 2-word (4 byte) instructions; everything else is 1-word (2 byte)
    char *src = malloc(strlen(source) + 1);
    strcpy(src, source);
    char *line = strtok(src, "\n");
    uint16_t addr = 0;
    while (line) {
        char *t = line;
        while (*t == ' ' || *t == '\t') t++;
        if (*t && *t != ';') {
            char *colon = strchr(t, ':');
            if (colon && colon[1] == '\0') {
                *colon = '\0';
                strncpy(labels[label_count].name, t, 31);
                labels[label_count].name[31] = '\0';
                labels[label_count].address = addr;
                label_count++;
            } else {
                char mn[16];
                sscanf(t, "%15s", mn);
                for (int i = 0; mn[i]; i++) mn[i] = toupper(mn[i]);
                if (!strcmp(mn, "LOAD") || !strcmp(mn, "CALL"))
                    addr += 4;  // Two-word instruction
                else
                    addr += 2;  // One-word instruction
            }
        }
        line = strtok(NULL, "\n");
    }
    free(src);

    // PASS 2: Write the actual task forms
    src = malloc(strlen(source) + 1);
    strcpy(src, source);
    line = strtok(src, "\n");
    while (line && word_count < max_words) {
        char *t = line;
        while (*t == ' ' || *t == '\t') t++;
        if (!*t || *t == ';' || strchr(t, ':')) { line = strtok(NULL,
"\n"); continue; }

        char mn[16], a1[32] = "", a2[32] = "";
        sscanf(t, "%15s %31[^,], %31s", mn, a1, a2);
        for (int i = 0; mn[i]; i++) mn[i] = toupper(mn[i]);

        if      (!strcmp(mn,"NOP"))  output[word_count++] =
ENCODE_REG(OP_NOP,0,0,0);
        else if (!strcmp(mn,"HALT")) output[word_count++] =
ENCODE_REG(OP_HALT,0,0,0);
        else if (!strcmp(mn,"RET"))  output[word_count++] =
ENCODE_REG(OP_RET,0,0,0);
        else if (!strcmp(mn,"LOAD")) {
            output[word_count++] = ENCODE_REG(OP_LOAD,
parse_register(a1), 0, 0);
            output[word_count++] = (uint16_t)parse_immediate(a2);
        }
        else if (!strcmp(mn,"MOV"))  output[word_count++] =
ENCODE_REG(OP_MOV, parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"ADD"))  output[word_count++] =
ENCODE_REG(OP_ADD, parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"SUB"))  output[word_count++] =
ENCODE_REG(OP_SUB, parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"AND"))  output[word_count++] =
ENCODE_REG(OP_AND, parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"OR"))   output[word_count++] =
ENCODE_REG(OP_OR,  parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"XOR"))  output[word_count++] =
ENCODE_REG(OP_XOR, parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"NOT"))  output[word_count++] =
ENCODE_REG(OP_NOT, parse_register(a1), 0, 0);
        else if (!strcmp(mn,"SHL"))  output[word_count++] =
ENCODE_REG(OP_SHL, parse_register(a1), 0, parse_immediate(a2));
        else if (!strcmp(mn,"SHR"))  output[word_count++] =
ENCODE_REG(OP_SHR, parse_register(a1), 0, parse_immediate(a2));
        else if (!strcmp(mn,"CMP"))  output[word_count++] =
ENCODE_REG(OP_CMP, parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"LDR"))  output[word_count++] =
ENCODE_REG(OP_LDR, parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"STR"))  output[word_count++] =
ENCODE_REG(OP_STR, parse_register(a1), parse_register(a2), 0);
        else if (!strcmp(mn,"PUSH")) output[word_count++] =
ENCODE_REG(OP_PUSH, parse_register(a1), 0, 0);
        else if (!strcmp(mn,"POP"))  output[word_count++] =
ENCODE_REG(OP_POP,  parse_register(a1), 0, 0);
        else if (!strcmp(mn,"JMP") || !strcmp(mn,"JZ") ||
!strcmp(mn,"JNZ") || !strcmp(mn,"JN")) {
            Opcode jop = !strcmp(mn,"JMP") ? OP_JMP : !strcmp(mn,"JZ") ?
OP_JZ :
                         !strcmp(mn,"JNZ") ? OP_JNZ : OP_JN;
            uint16_t target = 0;
            int found = 0;
            for (int i = 0; i < label_count; i++)
                if (!strcmp(a1, labels[i].name)) { target =
labels[i].address; found = 1; break; }
            if (!found && a1[0]) target = (uint16_t)atoi(a1);
            output[word_count++] = ENCODE_JUMP(jop, target);
        }
        else if (!strcmp(mn,"CALL")) {
            uint16_t target = 0;
            int found = 0;
            for (int i = 0; i < label_count; i++)
                if (!strcmp(a1, labels[i].name)) { target =
```

~~~
labels[i].address; found = 1; break; }
            if (!found && a1[0]) target = (uint16_t)atoi(a1);
            output[word_count++] = ENCODE_REG(OP_CALL, 0, 0, 0);
            output[word_count++] = target;
        }
        else fprintf(stderr, "Secretary can't understand: %s\n", mn);

        line = strtok(NULL, "\n");
    }
    free(src);
    return word_count;
}
```
~~~

Chapter 9: The Worker's First Real Assignment

Let’s give our worker a job that actually uses the full office: the desk, the calculator, the filing cabinet, and the inbox tray.

Task: Compute the sum of 1 + 2 + 3 + … + 100, which equals 5050.

~~~
```markdown
Sticky note R0 = 0         (running total)
Sticky note R1 = 100       (count down from 100)  ← Now possible!
Sticky note R2 = 1         (subtract by 1 each time)
Sticky note R3 = 0         (for comparing against zero)
Sticky note R4 = 1000      (filing cabinet folder for storing the answer)

loop:
  Add R1 to R0              (total += counter)
  Subtract R2 from R1       (counter -= 1)
  Compare R1 with R3        (is counter == 0? check status board)
  If Zero light is OFF, jump to loop

Store R0 into cabinet folder [R4]   (save answer in memory)
Push R0 onto inbox tray             (demonstrate stack: save R0)
Write 0 on R0                       (wipe R0 clean)
Pop inbox tray back to R0           (restore R0, it's 5050 again!)
Load from cabinet folder [R4] to R5 (verify memory works)
Go home                             (R0 = 5050, R5 = 5050)
```
~~~

Code: The Full Program

```c
// main.c: Hire the worker, give them a job

#include <stdio.h>
#include <inttypes.h>
#include "cpu.h"
#include "instruction.h"

extern void cpu_init(CPU *cpu);
extern void cpu_run(CPU *cpu);
extern uint16_t mem_read16(CPU *cpu, uint16_t address);
extern void mem_write16(CPU *cpu, uint16_t address, uint16_t value);
extern int assemble(const char *source, uint16_t *output, int max);

void load_program(CPU *cpu, uint16_t *words, int count) {
    for (int i = 0; i < count; i++)
        mem_write16(cpu, i * 2, words[i]);
    printf("Filed %d words into the cabinet (%d bytes)\n", count, count *
2);
}

void cpu_dump(CPU *cpu) {
    printf("\n=== DESK STATUS ===\n");
    for (int i = 0; i < NUM_REGISTERS; i++)
        printf("  Sticky note R%d = %d (0x%04X)\n", i, cpu->registers[i],
cpu->registers[i]);
    printf("  Checklist position: 0x%04X\n", cpu->pc);
    printf("  Inbox tray position: 0x%04X\n", cpu->sp);
    printf("  Status board: [%c%c%c]\n",
        (cpu->flags & FLAG_OVERFLOW) ? 'O' : '-',
        (cpu->flags & FLAG_NEGATIVE) ? 'N' : '-',
        (cpu->flags & FLAG_ZERO)     ? 'Z' : '-');
    printf("  Tasks completed: %" PRIu64 "\n", cpu->cycles);
}

int main(void) {
    printf("=== 16-BIT CPU EMULATOR ===\n");
    printf("=== The Office Worker Simulation ===\n\n");

    CPU cpu;
    cpu_init(&cpu);

    const char *program =
        "; Sum of 1 to 100: answer ends up on sticky note R0\n"
        "; Demonstrates: full LOAD, memory access, stack operations\n"
        "LOAD R0, 0\n"
        "LOAD R1, 100\n"
        "LOAD R2, 1\n"
        "LOAD R3, 0\n"
        "LOAD R4, 1000\n"
        "loop:\n"
        "ADD R0, R1\n"
        "SUB R1, R2\n"
        "CMP R1, R3\n"
        "JNZ loop\n"
        "STR R4, R0\n"
        "PUSH R0\n"
        "LOAD R0, 0\n"
        "POP R0\n"
        "LDR R5, R4\n"
        "HALT\n";

    printf("Checklist:\n%s\n", program);

    uint16_t machine_code[512];
    int count = assemble(program, machine_code, 512);

    printf("Task forms (machine code):\n");
    for (int i = 0; i < count; i++)
        printf("  Word [%02d] @ 0x%04X: 0x%04X\n", i, i * 2,
machine_code[i]);

    load_program(&cpu, machine_code, count);
    printf("\n=== Worker starts... ===\n");
    cpu_run(&cpu);
    cpu_dump(&cpu);

    // Verify results
    printf("\n=== VERIFICATION ===\n");
    uint16_t mem_result = mem_read16(&cpu, 1000);
    printf("  R0 (register)    = %d %s\n", cpu.registers[0],
cpu.registers[0] == 5050 ? "✓" : "✗");
    printf("  R5 (from memory) = %d %s\n", cpu.registers[5],
cpu.registers[5] == 5050 ? "✓" : "✗");
    printf("  Memory[1000]     = %d %s\n", mem_result, mem_result == 5050
? "✓" : "✗");

    if (cpu.registers[0] == 5050 && cpu.registers[5] == 5050 &&
mem_result == 5050)
        printf("\n✓ SUCCESS! All three match: 1+2+...+100 = 5050\n");
    else
```

~~~
        printf("\n✗ ERROR: something went wrong\n");

    return 0;
}
```
~~~

Compile & Run

~~~
```shell
gcc -o cpu16 main.c cpu_execute.c alu.c assembler.c memory.c -Wall -
Wextra -std=c11
./cpu16
```
~~~

Expected output:

~~~
```markdown
=== 16-BIT CPU EMULATOR ===
=== The Office Worker Simulation ===

...
Worker went home after 411 tasks.

=== DESK STATUS ===
  Sticky note R0 = 5050 (0x13BA)
  ...
  Sticky note R5 = 5050 (0x13BA)
  ...

=== VERIFICATION ===
  R0 (register)     = 5050 ✓
  R5 (from memory) = 5050 ✓
  Memory[1000]      = 5050 ✓

✓ SUCCESS! All three match: 1+2+...+100 = 5050
```
~~~

Chapter 10: What We Learned

1. Subtraction Is Equality

CPUs do not have an “equals” button. The worker subtracts two values and checks whether the Zero light on the status board is on.
If 5 equals 5, then 5 minus 5 equals 0, and the Zero light turns on.
Every if statement you have ever written works this way underneath.

2. The Desk Is Everything

The most important performance lesson is this: the worker can only do math on what is on the desk, which means the registers. Walking to the filing cabinet, which represents memory, is slow.

This is why real CPUs use caches. They act like a small bookshelf placed right next to the desk so the worker does not have to walk to the cabinet as often.

3. Encoding Is a Puzzle

Fitting the task type, the registers, and a literal value into 16 bits is a real constraint.

When we moved from 4-bit opcodes to 5-bit opcodes, we gained more instructions, increasing from 16 to 23. However, that extra bit had to come from somewhere, which meant losing one bit elsewhere in the encoding.

Real ISA designers at ARM and Intel spend years thinking about these trade-offs.

4. Two-Word Instructions Are a Real Thing

When 16 bits are not enough to hold both an opcode and a full value, real CPUs use multi-word instructions.

On x86, instructions can range from 1 to 15 bytes in length. Our LOAD and CALL instructions using two words follow the same idea.

5. The Stack Makes Code Reusable

Without PUSH, POP, CALL, and RET, every piece of logic would be a straight line.

The inbox tray, which represents the stack, allows the worker to save their place, jump to another section, and then return. That is what makes functions possible.

6. Flags Turn a Calculator Into a Computer

Without the status board, the worker could only follow the checklist in order.

The status board enables decisions, and decisions are what separate a calculator from a computer. Now, with the JN instruction, the worker can even ask whether the result was negative.

Congratulations 🎉 You have just built 16-Bit CPU from scratch in C.

The next time someone talks about CPU, you won’t just nod along. You built one from scratch, in pure C, with no dependencies.

Complete Code: https://github.com/vixhal-baraiya/16bit-cpu

(Don’t forget to ⭐ star the repository if you found this helpful!)

Keep building. Keep learning.