LR35902 Documentation

Composed By: Harry Alvarado

October 15, 2025

1 LEC 1: High-Level Details

1. **Name**: LR35902 (official manufacturer name) or SM83

   The CPU (Central Processing Unit) has three main jobs:

   - Fetch instructions from memory
   - Decode what those instructions mean
   - Execute them (math, moving, logic)

   The GB CPU is composed of the following subsystems: Figure 1 shows a simplified model of the

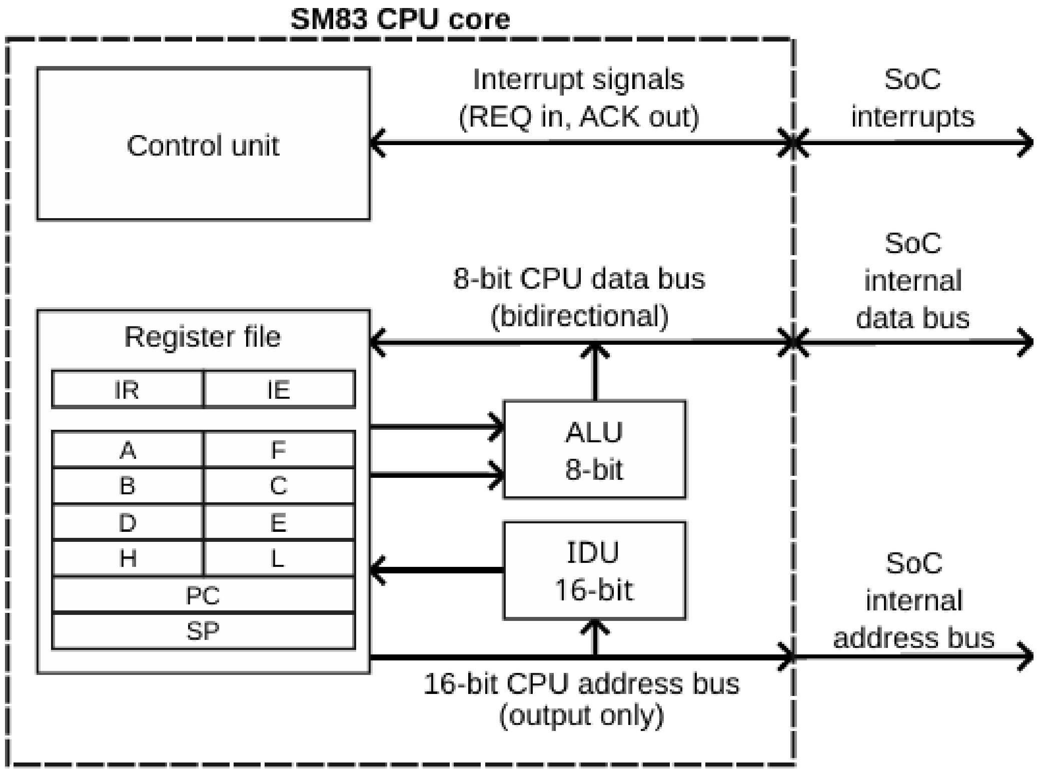


Figure 1: Simple model of the SM83 CPU core

SM83 CPU core. The core interacts with the rest of the SoC (System on Chip) using interrupt signals, an 8-bit bidirectional data bus, and a 16-bit address bus controlled by the CPU core. Their functions are described below:

- **Registers:** They are "ultra-fast" storage cells built right into the CPU used to hold temporary/intermediate data. They are essential because they allow quick access and storage (due to their small sizes) as opposed to relying on a slower (due to its enormous size) main memory. There are five types of registers.

  - *General-Purpose*: The purpose is to hold numbers that the processor is currently working with. In the GB, there are 5 labeled B,C,D,E,H,L, each of which is an 8-bit (byte) register. Which means they can store a number from $0 - 2^8$. But sometimes the CPU needs to work with bigger numbers or memory addresses, so in the GB we can pair registers and treat them like a 16-bit unit. What happens in these cases is that the data stored is connected end-to-end where one register represents the high byte and the other the lower byte. The GB only allows 4 register pairs: AF, BC, DE, HL. They serve as the destinations for most operations. The collection of general purpose registers is also known as the register file.
  - *Accumulator (A)*: Special-purpose register used to hold one of the operands for arithmetic/logical operations and store the results of those operations.
  - *Flag Register (F)*: A register whose bits record the status of the last ALU operation. It essentially tells the CPU what happened in the last calculation. We label its bits from 0-7, and each bit except the 3-0 stores meaningful information. The first three are always 0. Here I list the bit number, its name, and its meaning:

bit	name	meaning
7	Z-Zero	1 if the result was 0
6	N-Subtract	1 if the operation was subtraction
5	H-Half-carry	1 if a carry happend from bit 3→4
7	C-Carry	1 if there was an overflow beyond 8 bits

  - *Program Counter (PC)*: A 16-bit register that holds the address of the next instruction to execute in memory.
  - *Stack Pointer (SP)*: A 16-bit register that points to the top of the stack in RAM (Random Access Memory).

- **Arithmetic Logic Unit (ALU)**: The ALU performs the math and logic operations: +, -, AND, OR, XOR, comparison, bit shifts, and etc. Not much to say at the high level.
- **IDU:** A dedicated 16-bit Increment/Decrement Unit that operates independently of the ALU. It performs address increments/decrements on 16-bit values, outputting results back to the register file.
- **Control Unit (CU)**: This entity decides what happens when. It takes an instruction and generates the control signals that make all the other parts cooperate in the right order. The control unit decodes the executed instructions and generates control signals for the rest of the CPU core. It is also responsible for checking and dispatching interrupts.
- **Buses**: The CPU communicates with memory and peripherals through parallel electrical paths, which transfer a fixed number of bits at a time. These pathways are called buses. And there are three types:
  - 16-bit Address Bus: CPU's way of telling memory which memory location to access. One-way bus from CPU to Memory.
  - 8-bit Data Bus: carries the data from to/from memory. CPU sends data to be written and Memory send data that was requested.
  - Interruption Bus: signals interruptions from peripheral which tells the CPU to stop what it's doing and act in accordance to the signal. A peripheral that has access to this is an input controller.
- **Clock**: DMG Clock 4.194304 MHz; CPU runs at 1/4 master per machine cycle (M-cycle = 4 T-cycles)

  Explanation: In a chip, there is a multitude of operations occurring at the same time. Whether it is a simple transistor or some circuitry, its signals need to be coordinated because practical operations are order/time-dependent. If hardware ran as fast as electrons can move, then it would be impossible to coordinate anything. Therefore, we need a clock to set a pace (a speed) and synchronize (make sure things happen in order or at the same time) for related hardware components. It is implemented with a repeating electrical signal (up and down wave) caused by a crystal oscillator due to its voltage properties. You should think about it as creating this time unit for which everything else will be based on. How many times per second this wave goes up and down would be the frequency of the clock, measured in Hertz (Hz $\equiv \frac{1 cycle}{1 second}$). For the GB, the crystal vibrates at 4.194304 MHz. Recall that M (Mega) is 10^6. The timekeeper is the crystal as it sends out perfectly spaced "ticks" (another word for cycle) to set a pace.

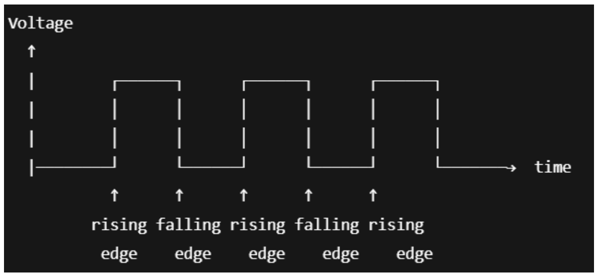


Figure 2: Clock Wave

The chip's logical units need to react in accordance with these pulses. The circuitry should update discretely, only when a specified clock edge arrives (usually the rising edge). Every subsystem on a motherboard ultimately divides or counts these pulses. Every tick can trigger:

- CPU to fetch/execute a micro-operation
- Memory read/write a byte
- graphics hardware to draw a pixel
- audit unit to move to the next waveform

In the GB, a cycle is called a T-cycle, and it is the smallest unit of time for which it operates. This is produced by the crystal, and it is referred to as the master clock. The CPU uses the pace unit of an M-cycle, which is 4 T-cycles. Each instruction takes a fixed number of M-cycles, and within it, each micro-operation consumes some T-cycles. Internally, the CPU's control unit counts those ticks so it knows exactly when it latches new data or signals "done."

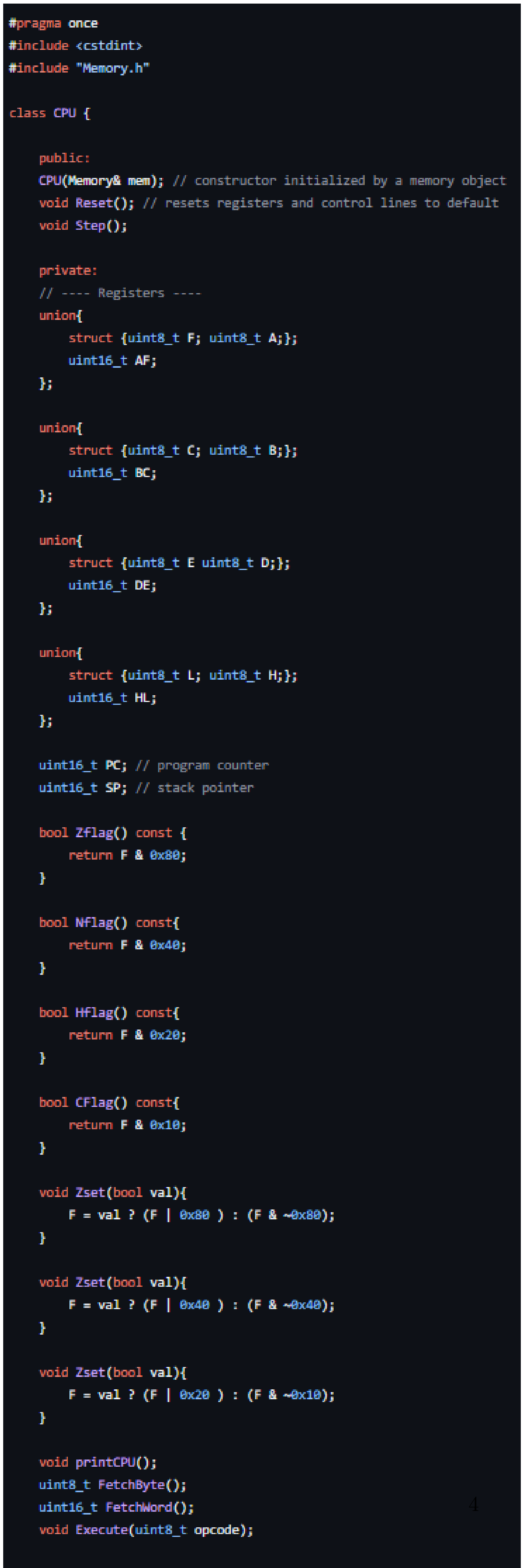
- **Memory**: The Game Boy does not include a hard drive. Instead, it relies on several types of memory chips, each with a specific purpose:

  (a) **RAM (Random Access Memory)**
  This is the main working memory used by the CPU while the system is running. It stores:
  - Temporary variables and runtime data
  - The program stack
  - Buffers for graphics and audio

  The original Game Boy includes ≈ 8 KB of internal RAM, and some cartridges provide additional external RAM (up to 32 KB or more).

  (b) **ROM (Read-Only Memory)**
  The ROM is located on the game cartridge and contains the program code and static game data (e.g., graphics, maps, sound tables). The CPU executes instructions directly from ROM, which functions similarly to a hard drive but is read-only and non-volatile.

  (c) **Save RAM (Battery-Backed SRAM)**
  Some cartridges include a small amount of battery-backed RAM (typically 2 KB to 32 KB) to store save data. This memory remains powered even when the Game Boy is turned off, preserving the player's progress. While not a hard drive, it serves as persistent storage.



```cpp
#pragma once
#include <cstdint>
#include "Memory.h"

class CPU {

    public:
    CPU(Memory& mem); // constructor initialized by a memory object
    void Reset(); // resets registers and control lines to default
    void Step();

    private:
    // ---- Registers ----
    union{
        struct {uint8_t F; uint8_t A;};
        uint16_t AF;
    };

    union{
        struct {uint8_t C; uint8_t B;};
        uint16_t BC;
    };

    union{
        struct {uint8_t E uint8_t D;};
        uint16_t DE;
    };

    union{
        struct {uint8_t L; uint8_t H;};
        uint16_t HL;
    };

    uint16_t PC; // program counter
    uint16_t SP; // stack pointer

    bool Zflag() const {
        return F & 0x80;
    }

    bool Nflag() const{
        return F & 0x40;
    }

    bool Hflag() const{
        return F & 0x20;
    }

    bool CFlag() const{
        return F & 0x10;
    }

    void Zset(bool val){
        F = val ? (F | 0x80 ) : (F & ~0x80);
    }

    void Zset(bool val){
        F = val ? (F | 0x40 ) : (F & ~0x40);
    }

    void Zset(bool val){
        F = val ? (F | 0x20 ) : (F & ~0x10);
    }

    void printCPU();
    uint8_t FetchByte();
    uint16_t FetchWord();
    void Execute(uint8_t opcode);
```

Explanation:
This code defines the internal structure of the CPU class. I used a special data type called **union** which is similar to a struct, but all the members share the same memory location. Therefore, we can only define one member at a time. These unions and structs are anonymous members, meaning I didn't give them a name, but I can access them directly within the class. These union definitions define our 8-bit registers and 16-bit pairs.
I have also defined some basic functions for the flag, which will either set a bit or tell me if there is a 1 at a bit location for F. I've also declared a printCPU() function, which I'll implement to see how the CPU is changing in the terminal. Additionally, two fetching functions will act like our 8-bit data bus and 16-bit address bus.

2 Task Distribution – Phase 1 (Instruction Set Development)

We are going to start implementing the instruction set following those mentioned in the CPU Hardware Document in GitHub. You can ignore the cycles information.

- **Areli**: Pg. 20 – 23
- **Brian**: Pg. 24 – 27
- **Cris**: Pg. 28 – 31
- **Daniel**: Pg. 32 – 35
- **Hafsah**: Pg. 36 – 39
- **Johnathan**: Pg. 40 – 43
- **Olamilekan**: Pg. 44 – 47
- **Patryk**: Pg. 48 – 51