This is part 1 of an article series on emulation development in JavaScript; ten parts are currently available, and others are expected to follow.

- *Part 1: The CPU*
- *Part 2: Memory*
- *Part 3: GPU Timings*
- *Part 4: Graphics*
- *Part 5: Integration*
- *Part 6: Input*
- *Part 7: Sprites*
- *Part 8: Interrupts*
- *Part 9: Memory Banking*
- *Part 10: Timers*

The emulator described in this series is available in source form: http://github.com/Two9A/jsGB

It's often stated that JavaScript is a special-purpose language, designed for use by web sites to enable dynamic interaction. However, JavaScript is a full object-oriented programming language, and is used in arenas besides the Web: the Widgets available for recent versions of Windows and Apple's Mac OS are implemented in JavaScript, as is the GUI for the Mozilla application suite.

With the recent introduction of the `<canvas>` tag to HTML, the question arises as to whether a JavaScript program is capable of emulating a system, much like desktop applications are available to emulate the Commodore 64, GameBoy Advance and other gaming consoles. The simplest way of checking whether this is viable is, of course, to write such an emulator in JavaScript.

This article sets out to implement the basis for a GameBoy emulation, by laying the groundwork for emulating each part of the physical machine. The starting point is the CPU.

The model

The traditional model of a computer is a processing unit, which gets told what to do by a program of instructions; the program might be accessed with its own special memory, or it might be sitting in the same area as normal memory, depending on the computer. Each instruction takes a short amount of time to run, and they're all run one by one. From the CPU's perspective, a loop starts up as soon as the computer is turned on, to fetch an instruction from memory, work out what it says, and execute it.

In order to keep track of where the CPU is within the program, a number is held by the CPU called the Program Counter (PC). After an instruction is fetched from memory, the PC is advanced by however many bytes make up the instruction.

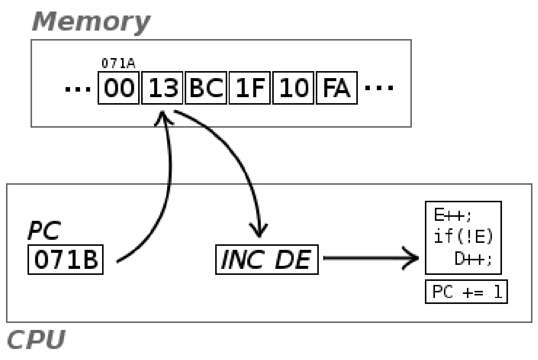

Figure 1: The fetch-decode-execute loop

The CPU in the original GameBoy is a modified Zilog Z80, so the following things are pertinent:

- The Z80 is an 8-bit chip, so all the internal workings operate on one byte at a time;
- The memory interface can address up to 65,536 bytes (a 16-bit address bus);
- Programs are accessed through the same address bus as normal memory;
- An instruction can be anywhere between one and three bytes.

In addition to the PC, other numbers are held inside the CPU that can be used for calculation, and they're referred to as registers: A, B, C, D, E, H, and L. Each of them is one byte, so each one can hold a value from 0 to 255. Most of the instructions in the Z80 are used to handle values in these registers: loading a value from memory into a register, adding or subtracting values, and so forth.

If there are 256 possible values in the first byte of an instruction, that makes for 256 possible instructions in the basic table. That table is detailed in the Gameboy Z80 opcode map released on this site. Each of these can be simulated by a JavaScript function, that operates on an internal model of the registers, and produces effects on an internal model of the memory interface.

There are other registers in the Z80, that deal with holding status: the flags register (F), whose operation is discussed below; and the stack pointer (SP) which is used alongside the PUSH and POP instructions for basic LIFO handling of values. The basic model of the Z80 emulation would therefore require the following components:

- An internal state:
  - A structure for retaining the current state of the registers;
  - The amount of time used to execute the last instruction;
  - The amount of time that the CPU has run in total;
- Functions to simulate each instruction;
- A table mapping said functions onto the opcode map;
- A known interface to talk to the simulated memory.

The internal state can be held as follows:

Z80.js: Internal state values

```
Z80 = {
    // Time clock: The Z80 holds two types of clock (m and t)
    _clock: {m:0, t:0},

    // Register set
    _r: {
        a:0, b:0, c:0, d:0, e:0, h:0, l:0, f:0,    // 8-bit registers
        pc:0, sp:0,                                // 16-bit registers
        m:0, t:0                                   // Clock for last instr
    }
};
```

The flags register (F) is important to the functioning of the processor: it automatically calculates certain bits, or flags, based on the result of the last operation. There are four flags in the Gameboy Z80:

- Zero (0x80): Set if the last operation produced a result of 0;
- Operation (0x40): Set if the last operation was a subtraction;
- Half-carry (0x20): Set if, in the result of the last operation, the lower half of the byte overflowed past 15;
- Carry (0x10): Set if the last operation produced a result over 255 (for additions) or under 0 (for subtractions).

Since the basic calculation registers are 8-bits, the carry flag allows for the software to work out what happened to a value if the result of a calculation overflowed the register. With these flag handling issues in mind, a few examples of instruction simulations are shown below. These examples are simplified, and don't calculate the half-carry flag.

Z80.js: Instruction simulations

```
Z80 = {
    // Internal state
    _clock: {m:0, t:0},
    _r: {a:0, b:0, c:0, d:0, e:0, h:0, l:0, f:0, pc:0, sp:0, m:0, t:0},

    // Add E to A, leaving result in A (ADD A, E)
    ADDr_e: function() {
        Z80._r.a += Z80._r.e;                       // Perform addition
        Z80._r.f = 0;                               // Clear flags
        if(!(Z80._r.a & 255)) Z80._r.f |= 0x80;     // Check for zero
        if(Z80._r.a > 255) Z80._r.f |= 0x10;        // Check for carry
        Z80._r.a &= 255;                            // Mask to 8-bits
        Z80._r.m = 1; Z80._r.t = 4;                 // 1 M-time taken
    }

    // Compare B to A, setting flags (CP A, B)
    CPr_b: function() {
        var i = Z80._r.a;                           // Temp copy of A
        i -= Z80._r.b;                              // Subtract B
        Z80._r.f |= 0x40;                           // Set subtraction flag
        if(!(i & 255)) Z80._r.f |= 0x80;            // Check for zero
        if(i < 0) Z80._r.f |= 0x10;                 // Check for underflow
        Z80._r.m = 1; Z80._r.t = 4;                 // 1 M-time taken
    }

    // No-operation (NOP)
    NOP: function() {
        Z80._r.m = 1; Z80._r.t = 4;                 // 1 M-time taken
    }
};
```

Memory interfacing

A processor that can manipulate registers within itself is all well and good, but it must be able to put results into memory to be useful. In the same way, the above CPU emulation requires an interface to emulated memory; this can be provided by a memory management unit (MMU). Since the Gameboy itself doesn't contain a complicated MMU, the emulated unit can be quite simple.

At this point, the CPU only needs to know that an interface is present; the details of how the

Gameboy maps banks of memory and hardware onto the address bus are inconsequential to the processor's operation. Four operations are required by the CPU:

MMU.js: Memory interface

```
MMU = {
    rb: function(addr) { /* Read 8-bit byte from a given address */ },
    rw: function(addr) { /* Read 16-bit word from a given address */ },

    wb: function(addr, val) { /* Write 8-bit byte to a given address */ },
    ww: function(addr, val) { /* Write 16-bit word to a given address */ }
};
```

With these in place, the rest of the CPU instructions can be simulated. Another few examples are shown below:

Z80.js: Memory-handling instructions

```

    // Push registers B and C to the stack (PUSH BC)
    PUSHBC: function() {
        Z80._r.sp--;                           // Drop through the stack
        MMU.wb(Z80._r.sp, Z80._r.b);           // Write B
        Z80._r.sp--;                           // Drop through the stack
        MMU.wb(Z80._r.sp, Z80._r.c);           // Write C
        Z80._r.m = 3; Z80._r.t = 12;           // 3 M-times taken
    },

    // Pop registers H and L off the stack (POP HL)
    POPHL: function() {
        Z80._r.l = MMU.rb(Z80._r.sp);          // Read L
        Z80._r.sp++;                           // Move back up the stack
        Z80._r.h = MMU.rb(Z80._r.sp);          // Read H
        Z80._r.sp++;                           // Move back up the stack
        Z80._r.m = 3; Z80._r.t = 12;           // 3 M-times taken
    }

    // Read a byte from absolute location into A (LD A, addr)
    LDAmm: function() {
        var addr = MMU.rw(Z80._r.pc);          // Get address from instr
        Z80._r.pc += 2;                        // Advance PC
        Z80._r.a = MMU.rb(addr);               // Read from address
        Z80._r.m = 4; Z80._r.t=16;             // 4 M-times taken
    }
```

Dispatch and reset

With the instructions in place, the remaining pieces of the puzzle for the CPU are to reset the CPU when it starts up, and to feed instructions to the emulation routines. Having a reset routine allows for the CPU to be stopped and "rewound" to the start of execution; an example is shown below.

Z80.js: Reset

```
    reset: function() {
        Z80._r.a = 0; Z80._r.b = 0; Z80._r.c = 0; Z80._r.d = 0;
        Z80._r.e = 0; Z80._r.h = 0; Z80._r.l = 0; Z80._r.f = 0;
        Z80._r.sp = 0;
        Z80._r.pc = 0;      // Start execution at 0

        Z80._clock.m = 0; Z80._clock.t = 0;
    }
```

In order for the emulation to run, it has to emulate the fetch-decode-execute sequence detailed earlier. "Execute" is taken care of by the instruction emulation functions, but fetch and decode require a specialist piece of code, known as a "dispatch loop". This loop takes each instruction, decodes where it must be sent for execution, and dispatches it to the function in question.

Z80.js: Dispatcher

```
while(true)
{
    var op = MMU.rb(Z80._r.pc++);              // Fetch instruction
    Z80._map[op]();                            // Dispatch
    Z80._r.pc &= 65535;                        // Mask PC to 16 bits
    Z80._clock.m += Z80._r.m;                  // Add time to CPU clock
    Z80._clock.t += Z80._r.t;
}

Z80._map = [
    Z80._ops.NOP,
    Z80._ops.LDBCnn,
    Z80._ops.LDBCmA,
    Z80._ops.INCBC,
    Z80._ops.INCr_b,
    ...
];
```

Usage in a system emulation

Implementing a Z80 emulation core is useless without an emulator to run it. In the next part of this series, the work of emulating the Gameboy begins: I'll be looking at the Gameboy's memory map, and how a game image can be loaded into the emulator over the Web.

The complete Z80 core is available at: http://imrannazar.com/content/files/jsgb.z80.js; please feel free to let me know if you encounter any bugs in the implementation.

Imran Nazar <tf@imrannazar.com>, Jul 2010.

Article dated: 22nd Jul 2010