8bitMicroSim Programmer's Guide

The 8bitMicroSim is a simulation of an 8bit microcomputer on the hardware level, as the name implies. It is, as such, vital to understand the CPU's instruction set and behavior before beginning to write programs for the platform.

8bitMicroSim is intended as a learning tool with the goal to provide a safe environment which takes little to no effort to set up and can be used to gain an understanding of what *really* makes our computers tick.

This short guide will take you through the instruction set, how the CPU goes through memory, its registers and flags, and how to write programs in both machine code and assembly language.

The Memory

The Memory consists of addresses and values. Each address contains an 8bit number as its value. Please be aware that this memory is not permanent and will be lost every time the computer powers off. Though we can make our lives a lot easier by putting the values we want the simulator to load as memory into a binary file and just having the Simulator read that.

The 8bitMicro (the made-up computer we are simulating) has a whopping 64kb of memory. This may seem ridiculously low by today's standards, but we will soon see that a lot can be done with just a few bytes of memory, let alone sixty-four thousand.

The first 256 bytes of memory are reserved for the stack and the next 128 for interrupt handlers. We will touch on these topics later, for now, just remember that they are reserved. You can still manipulate values in this part of the memory, of course, as you have full control over the system. Though this may break the computer until its next restart, so be careful.

The Program Counter

The Program Counter points to an address in Memory, which (hopefully) contains an instruction the CPU can execute.

On the 8bitMicro, the Program Counter starts off at position 0x0180, since everything before that is reserved (0x____ is the notation for values in hexadecimal, we will exclusively use this format from now on, as it makes working with the computer a lot easier).

With the Program Counter set, the CPU will read the value at its position in memory and interpret the value as an instruction.

If the OPCODE (the value the CPU has just read) matches up with an existing instruction, it will be executed. Otherwise, something has gone terribly wrong, and you should double-check your code.

Once an Instruction has been executed, the Program Counter is incremented, and on its next cycle, the CPU reads the value in the next address.

The Stack

As previously mentioned, a certain part of our memory is reserved for the stack. The stack can be imagined as, well, a stack. Of values. The CPU can either push values onto the stack or pull (also referred to as bump in Assembler mnemonics) them off of it. Each push or pull operation changes the Stack Pointer. The Stack Pointer is very similar to the program counter, in the sense that it points to a relevant address in memory, though the similarities end there.

The Stack Pointer starts off pointing to the very end of the stack-reserved portion of memory. We start off at the end, because that way, aside from already pointing to the current position on the stack, it also tells us how much space we have left before it overflows (loops back to the beginning, this is called a stack overflow and is generally regarded as suboptimal).

An Example:
We have saved a few values in the stack and want to know if we can still save a few more. Without having to do any math, we simply read the stack pointer. 0x0F would mean that we still have 16 (since 0x00 is also a valid address) bytes remaining. Neat.

When pushing onto the stack (with a specific set of instructions, which we will get to later), the value is written into the address the Stack Pointer is currently pointing to, then the Stack Pointer is decremented by one.

Pulling from the stack is quite literally the same set of operations but in reverse.
First, the Stack Pointer is incremented by one, and the value at its position is then written into whatever register (depending on the instruction) we want it to write to.

Interrupts

Sometimes we just want something *special* to happen. That's where interrupts come in. Interrupts can be called from our code, or from the hardware.

Say, someone presses a key on the keyboard. Ideally, you'd want the computer to react to it, so an interrupt is called.
Interrupts allow the processor to stop wherever it is, jump to somewhere else, handle the problem and, with the power of the Stack, jump right back to where it was and resume whatever it was doing before.

A portion (a third) of the reserved memory is allocated to interrupt handling. When a hardware interrupt is triggered, the CPU automatically saves the Program Counter and the value of the Accumulator to the stack and changes the Program Counter to a memory address assigned to the specific interrupt code. Any Arguments (e.g., the value of the Key pressed) will be put into the accumulator.

It is generally not a good idea to put your entire interrupt handling code in the tiny portion of memory reserved for it, but instead simply put a jump instruction to a portion of memory where you then put your actual code to deal with the interrupt. Another benefit of this is that the address the CPU jumps to can then actually be changed, so depending on your program, you could implement different routines for the same interrupt codes. Interrupts can also be enabled or disabled using the Interrupt-Enable flag and its respective instructions.

The CPU can also raise an interrupt through our code. Which, for example, can tell the BIOS to print a character on the screen. We will go into greater detail on the Interrupt instruction later.

Registers

Our 8bitMicro CPU has eight registers, which an each hold an 8bit value.
The most prevalent of these is the Accumulator, which is the primary target of most mathematical operations. Additionally, the H and L

registers are used to represent the HIGH and LOW bytes of a 16bit address. There are certain instructions which do not require an address to be specified in following bytes, but instead construct it out of the H and L registers.

The Registers are as follows:

A – **ACC** – The Accumulator
B – A regular register used for holding data
C – A regular register used for holding data
D – A regular register used for holding data
E – A regular register used for holding data
F – The Flag Register, it holds the CPU flags
H – High byte Register, holds the first part of a 16bit address
L – Low byte Register, holds the second part of a 16bit address

Instructions to manipulate the Registers' values will be covered later.

Flags

The CPU has a Flag Register, it holds important information on the CPUs previous operation, which can be used by other instructions and conditionals.

The Flag Register's 8bit value is composed as follows:

```
7 6 5 4 3 2 1 0
S Z R I O P E C
```

```
S: Sign bit, 1 if the result of the last operation was > 0
Z: Zero bit, 1 if the result of the last operation was 0
R: Run Flag, 1 if the CPU should execute instructions
I: Interrupt enable bit
O: Overflow Flag, 1 if a Stack-Overflow has occured
P: Parity bit, 1 if the result of the last op. was even
E: Error flag, 1 if an invalid opcode was encountered
C: Carry bit, 1 if the prev. op. resulted in an overflow
```

Instruction Set

The following table shows all 8bitMicro CPU instructions, they will be explained in greater detail shortly.

XX	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X	NOP	HLT	IEN	IDN	RST	INT 1	INT 2	INT 3	INT 4	INT 5	INT 6	INT 7	INT 8			
1X	CMP A	CMP B	CMP C	CMP D	CMP E	CMP F	CMP H	CMP L	NND A	NND B	NND C	NNDD	NND E	NND F	NND H	NND L
2X	LDM	STM	LDA d16	LDL d8	STA d16	LHL d16	SHL d16		NOT A	NOT B	NOT C	NOT D	NOT E	NOT F	NOT H	NOT L
3X	MOV A,A	MOV A,B	MOV A,C	MOV A,D	MOV A,E	MOV A,F	MOV A,H	MOV A,L	MOV B,A	MOV B,B	MOV B,C	MOV B,D	MOV B,E	MOV B,F	MOV B,H	MOV B,L
4X	MOV C,A	MOV C,B	MOV C,C	MOV C,D	MOV C,E	MOV C,F	MOV C,H	MOV C,L	MOV D,A	MOV D,B	MOV D,C	MOV D,D	MOV D,E	MOV D,F	MOV D,H	MOV D,L
5X	MOV E,A	MOV E,B	MOV E,C	MOV E,D	MOV E,E	MOV E,F	MOV E,H	MOV E,L	MOV F,A	MOV F,B	MOV F,C	MOV F,D	MOV F,E	MOV F,F	MOV F,H	MOV F,L
6X	MOV H,A	MOV H,B	MOV H,C	MOV H,D	MOV H,E	MOV H,F	MOV H,H	MOV H,L	MOV L,A	MOV L,B	MOV L,C	MOV L,D	MOV L,E	MOV L,F	MOV L,H	MOV L,L
7X	JMP d16	JIE d16	JNE d16	JLT d16	JIC d16	JHL	JGT d16		AND A	AND B	AND C	AND D	AND E	AND F	AND H	AND L
8X	ADD A	ADD B	ADD C	ADD D	ADD E	ADD F	ADD H	ADD L	SUB A	SUB B	SUB C	SUB D	SUB E	SUB F	SUB H	SUB L
9X	INC A	INC B	INC C	INC D	INC E	INC F	INC H	INC L	DEC A	DEC B	DEC C	DEC D	DEC E	DEC F	DEC H	DEC L
AX	INA d16	DEA d16	INM	DEM					ORA A	ORA B	ORA C	ORA D	ORA E	ORA F	ORA H	ORA L
BX	CAL d16	CIE d16	CNE d16	CLT d16	CGT d16	CIC d16	CHL	RET	XOR A	XOR B	XOR C	XOR D	XOR E	XOR F	XOR H	XOR L
CX	SPS	SPW	SPL d8	SPA d16	SBA d16	SPM	SBM		MIL A,d8	MIL B,d8	MIL C,d8	MIL D,d8	MIL E,d8	MIL F,d8	MIL H,d8	MIL L,d8
DX	MVO A,d16	MVO B,d16	MVO C,d16	MVO D,d16	MVO E,d16	MVO F,d16	MVO H,d16	MVO L,d16	MVI A,d16	MVI B,d16	MVI C,d16	MVI D,d16	MVI E,d16	MVI F,d16	MVI H,d16	MVI L,d16
EX	MOM A	MOM B	MOM C	MOM D	MOM E	MOM F	MOM H	MOM L	MIM A	MIM B	MIM C	MIM D	MIM E	MIM F	MIM H	MIM L
FX	SPR A	SPR B	SPR C	SPR D	SPR E	SPR F	SPR H	SPR L	SBR A	SBR B	SBR C	SBR D	SBR E	SBR F	SBR H	SBR L

	System Instructions
	Data Assignment Instructions
	Math instructions
	Program Counter Instructions
	Stack Instructions
	Bitwise Instructions

Data Assignment Instructions

LDM – Load A from Memory Address, Load value from address specified in H and L Registers into Accumulator [1 byte]

STM – Store A into Memory Address, Store value of Accumulator in address specified in H and L Registers [1 byte]

LDA – Load A from Memory Address, Load value from address specified in the next two bytes into Accumulator [3 bytes]

STA – Store A into Memory Address, Store value of Accumulator in address specified in the next two bytes [3 bytes]

LDL – Load Literal, Load value in next byte in Accumulator [2 bytes]

LHL – Load H and L, Load an address (next two bytes) into the H and L registers [3 bytes]

SHL – Store H and L, Store the values in H and L into an address (next two bytes) [3 bytes]

MOV – Move value between registers, [destination], [source] [1 byte]

MIL – Move in Literal, moves value in next byte into register [2 bytes]

MVO – Move value out, move value from specified register into address specified in next two bytes [3 bytes]

MVI – Move value in, move value from address specified in next two bytes into specified register [3 bytes]

MOM – Move out Memory, move value from specified register into address specified by H and L registers [1 byte]

MIM – Move in Memory, move value from address specified by H and L registers into specified register [1 byte]

Program Counter Instructions

JMP – Jump, Change Program Counter to address specified in the next two bytes [3 bytes]

JIE – Jump if equal, JMP but only if 0 flag set [3 bytes]

JNE – Jump if not equal, JMP only if 0 flag not set [3 bytes]

JLT – Jump if less than, JMP if sign flag set [3 bytes]

JGT – Jump if greater than, JMP if sign flag not set [3 bytes]

JHL – Jump to H L address, JMP to the address specified in H and L registers [1 byte]

JIC – Jump if carry, JMP if carry flag set [3 bytes]

CAL – Call, JMP but Program Counter is pushed to stack [3 bytes]

CIE – Call if equal, JIE but Program Counter is pushed to stack [3 bytes]

CNE – Call if not equal, JNE but Program Counter is pushed to stack [3 bytes]

CLT – Call if less than, JLT but Program Counter is pushed to stack [3 bytes]

CGT – Call if greater than, JGT but Program Counter is pushed to stack [3 bytes]

CIC – Call if carry, JIC but Program Counter is pushed to stack [3 bytes]

CHL – Call to HL address, JHL but Program Counter is pushed to stack [3 bytes]

RET – Return, pulls last two bytes from stack and jumps to the address specified by them [1 byte]

Math Instructions

CMP – Compare A with, compares the value in Accumulator with other Register (Accumulator not changed) [1 byte]

ADD – Add to A, adds value of other Register to Accumulator [1 byte]

SUB – Subtract from A, subtracts value of other Register from Accumulator [1 byte]

INC – Increment Register, adds 1 to specified register [1 byte]

DEC – Decrement Register, subtracts 1 from specified register [1 byte]

INA – Increment Address, adds 1 to value in address specified in the next two bytes [3 bytes]

DEA – Decrement Address, subtracts 1 from value in address specified in the next two bytes [3bytes]

INM – Increment HL Address, adds 1 to value in address specified by H and L Registers [1 byte]

DEM – Decrement HL Address, subtracts 1 from value in address specified by H and L Registers [1 byte]

Stack Instructions

SPR – Stack Pointer Read, writes value of Stack pointer into Accumulator [1 byte]

SPW – Stack Pointer Write, sets Stack Pointer to value of Accumulator
[1 byte]

SPL – Stack push literal, push value in next byte onto stack [2 bytes]

SPA – Stack push address, push value in address specified in next two bytes onto stack [3 bytes]

SBA – Stack bump address, pull value from stack into address specified in next two bytes [3 bytes]

SPM – Stack push Memory, push value from address specified in H and L registers onto stack [1 byte]

SBM – Stack bump Memory, pull value from stack into address specified in H and L registers [1 byte]

SPR – Stack push Register, Push value of specified register onto stack [1 byte]

SBR – Stack bump Register, Pull value from stack into specified register [1 byte]

Bitwise Instructions

AND - Bitwise AND operation, Performs a bitwise AND between Accumulator and specified Register [1 byte]

ORA - Bitwise OR operation, Performs a bitwise OR between Accumulator and specified Register [1 byte]

XOR - Bitwise XOR operation, Performs a bitwise XOR between Accumulator and specified Register [1 byte]

NND - Bitwise NAND operation, Performs a bitwise NAND between Accumulator and specified Register [1 byte]

NOT - Bitwise NOT operation, Performs a bitwise NOT on specified Register [1 byte]

System Instructions

NOP – No operation, do nothing [1 byte]

HLT – Halt execution. Stop. [1 byte]

INT – Raise Interrupt, Interrupt code depends on first four bits of the instruction [0b 0000 0101 - 0b 0000 1100] [1 byte]

IEN – Interrupt enable, set interrupt enable flag to 1 [1 byte]

IDN – Interrupt disable, set interrupt enable flag to 0 [1 byte]

RST – Reset (soft), set Program Counter to starting positions and all Registers to 0 [1 byte]

Programming the 8bitMicro

As promised, we will now touch on writing programs for the 8bitMicro.

Machine Code

Machine Code, colloquially known as Machine Language, is the most basic method of programming. The program is already in the "language" the CPU understands. The values we insert into memory for the CPU to process are the OPCODEs shown previously, literals (values to be inserted somewhere) or addresses (two literals, one being the high, the other the low byte).

Assembly Language

Since writing programs in Machine Code is rather cumbersome and not really human-friendly, we can write our code in assembly, using mnemonics (short, three-character words that represent an instruction and/or arguments). This makes writing code a lot easier for us humans, with the downside that the computer can no longer understand it. To turn our assembly code into a set of instructions the computer can execute, we need to turn it back into machine code. This can be done manually or through a program called an *assembler*.

Fibonacci sequence in Machine Language

For our first program we won't be using an assembler. Instead, we'll write it in machine code. To make our lives a bit easier doing this however, we can still write it in assembly, we'll just have to *assemble* it into machine code manually.

```
0x0180: MIL H, 0x01     ;load high byte of target address
0x0182: MIL L, 0xB1     ;load low byte of target address
0x0184: LDL 0x01        ;load value into A
0x0186: MIL B, 0x00     ;load value into B
0x0188: SPR A           ;Push value of A onto Stack
0x0189: STM             ;store value of A in HL-address
0x018A: INC L ;increment low byte of target address by one
0x018B: ADD B           ;add value of B to A
0x018C: SBR B ;Pull (bump) value from Stack into B
0x018D: JIC 0x0193      ;jump if carry bit set
0x0190: JMP 0x0188      ;jump
0x0193: HLT             ;halt program execution
```

Now that we have our program, let's turn it into machine code.

Since our Program Counter starts off pointing to `0x0180`, we'll start our program there. The Mnemonic `MIL H` is assembled into the Machine Code instruction `CE`, while the value `0x01` sits in the next byte. As such, our instruction takes up two bytes of memory, that's why the next line starts at `0x0182`. We continue doing this and assigning values to memory addresses until we reach the end of our code.

The assembled program, therefore, looks like this

```
XXX0 X0 X1 X2 X3 X4 X5 X6 X7 X8 X9 XA XB XC XD XE XF
0180 CE 01 CF B1 23 01 C9 00 F0 21 97 81 F9 74 01 A0
0190 70 01 88 01 00 00 00 00 00 00 00 00 00 00 00 00
```

Finally, when we insert this machine code into our computer's memory and activate run mode, the Fibonacci sequence is written into memory until we encounter an overflow (the number becomes too great to store in a byte) and execution is halted.

Once the computer has stopped and we inspect the memory dump, we can see that everything has gone to plan, and the desired values are in memory (Addresses `0x01B0` – `0x01BD` contain the desired Fibonacci sequence). We can also verify that our desired exit condition, the carry bit being `1`, is fulfilled.

```
Flag Register Breakdown:
S Z R I O P E C
1 0 0 0 0 1 0 1

...
01B0 00 01 01 02 03 05 08 0d 15 22 37 59 90 e9 00 00
...
```

More stuff to come here :)