

## Introduction

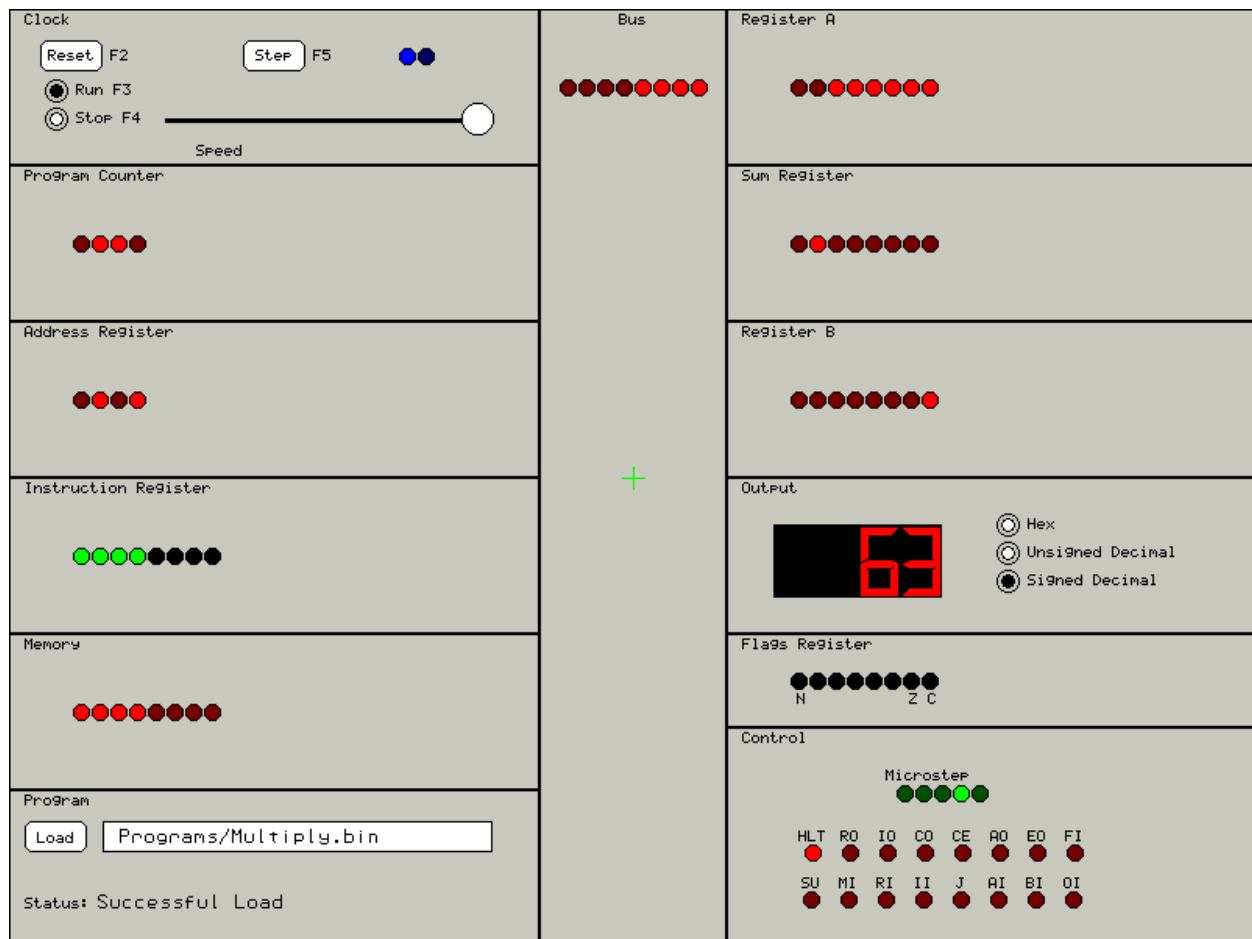
The computer scientist A.P. Malvino wrote an excellent textbook on microprocessor architecture called *Digital Computer Electronics*, which is out of print now but available in used editions on Amazon and elsewhere. He described a tiny “starter” microprocessor called SAP1: “Simple as Possible 1”. In later chapters, this model was elaborated and expanded until it was very nearly identical to the Intel 8085 8-bit microprocessor.

Ben Eater has a great series of videos on YouTube where he builds the SAP1 entirely out of breadboards and TTL chips. See his YouTube playlist titled “Building an 8-Bit Breadboard Computer.”

I realize that many, if not all of the Maximite enthusiasts have likely acquired electronics and microprocessor smarts that exceed the content I am talking about. But I found it highly informative and useful as a teaching device for the oldest of my grandkids, who is interested in computers.

So I built an SAP1 emulation for the Color Maximite 2. Its visual appearance owes a lot to Ben Eater’s breadboard computer, including the various modules and the blinking lights. Ben had to do all of his program assembly by hand, but I also wrote an assembler for the very limited instruction set of the SAP1.

Figure 1 – SAP1 Emulator Main Screen



My SAP1 distribution includes the SAP1 emulator, the Assembler, and several sample programs in both their assembler source code (.asm) and assembled binary (.bin) versions.

### How to use the SAP1 Emulation

The SAP1 emulator can be run using a mouse or just the keyboard.

After you start the SAP1 computer, you will want to load a program. To do this, press the F12 function key or click the Load button at the bottom left of the screen. You will see a list of the binary versions of the available programs in the Programs directory. Use the Up and DOWN Arrow Keys on the keyboard to scroll through those programs and press the ENTER key to load one. You should see the program name and the message “Successfully Loaded” at the bottom left of the screen.

If you are using a mouse, you will find the SAP1 Clock Controls at the top left of the screen. If you are using just a keyboard, see the keyboard instructions two paragraphs below. You can either single-step or run a program. To single\_step, just press the STEP button. Each press will cause the clock signal to go high and then low again. You can watch this transition by looking at the 2 blue lights in the clock module.

To run the program, press the RUN button. The clock will start cycling, and, depending on which program was loaded, you will see the lights in various modules turn on and off as the SAP1 works through the program steps. Press the STOP button to halt the program in place. If you want to run the program again from the beginning, press the RESET button and then the START button again.

If you are controlling using just the keyboard, here is the complete list of keyboard commands:

| Key         | Mode              | Command                       |
|-------------|-------------------|-------------------------------|
| UP arrow    | Loading a program | Move up program list          |
| UP arrow    | Normal            | Change digital output format  |
| DOWN arrow  | Loading a program | Move down program list        |
| DOWN arrow  | Normal            | Change digital output format  |
| LEFT arrow  | Normal            | Change clock speed slower     |
| RIGHT arrow | Normal            | Change clock speed faster     |
| ENTER       | Loading a Program | Select program to load        |
| F1          | Normal            | List these commands on screen |
| F2          | Normal            | Reset Computer                |
| F3          | Normal            | Start Clock                   |
| F4          | Normal            | Stop Clock                    |
| F5          | Normal            | Step Clock 1 cycle            |
| F8          | Normal            | Show an execution trace       |
| F12         | Normal            | Load a program file           |
| Escape      | Normal            | Quit SAP1 program             |

By default, the clock runs very slowly. You can use the SPEED slider to set the clock speed. Use the Left and Right Arrow keys to drag the speed control, with Right being faster. This works for both mouse and

keyboard users. Note that the speed change has no effect if the clock is running. Stop the clock, set the speed, and start the clock again to get the effect.

## SAP1 Architecture

The SAP1 has only a 4-bit memory bus, which means the RAM memory is only 16 6-bit values. So you have to be economical and ingenious to squeeze both programs and data into this tiny storage. The rest of the SAP1 registers are 8 bits wide. Looking at the screen, you can see ALL of its components and their current states. Moving down the left side of the screen and then down the right side, the modules are:

**The Clock Module**, which we have already discussed.

**The Program Counter**. This 4-bit register shows the next memory address to be accessed for reading or writing.

**The Address Register**. This 4-bit register shows the current memory address being accessed for reading or writing.

**The Instruction Register**. This 8-bit register is divided into 2 sections: the left 4 bits are the instruction opcode value, and the right 4 bits are the instruction operand.

**The Memory Register**. This 8-bit register shows the value of the RAM location being accessed.

**Program**. This is where you load programs when using a mouse or keyboard, as discussed above.

**The Bus**. All of the functional elements of the SAP1 are connected by an 8-bit Tri-state bus. Any element can be a bus master and assert a value (4-bit or 8-bit) onto the bus, and any other element can be a bus client and load the value being expressed on the bus. The control logic and control bits (see below) control the bus and ensure that there is at most a single bus master and a single bus client.

**Register A**. The accumulator. This 8-bit register works in conjunction with Register B and the Sum Register to perform addition and subtraction.

**Sum Register**. It is hardwired to always contain the sum of the A and B registers. Subtraction is accomplished by doing a 2's complement negation of the B value.

**Register B**. Gets loaded from memory when the ADD or SUB instruction are executed.

**Output**. A simulated 3-character 7-segment display whose value will be changed to the contents of Register A when the OUT instruction is executed. You can view the output value as signed integer, unsigned integer, or hexadecimal; click the radio buttons next to the display or use the UP and DOWN keyboard keys.

**Flags Register**. This 8-bit register only uses the two least-significant bits to hold the Carry flag (bit 0) and the Zero flag (bit 1). The Carry flag is set whenever the ADD or SUB instruction causes the Sum register to overflow. The Zero flag is set whenever the A register has a zero value. The "Negative" flag is set whenever the most significant bit of the A register is 1. These three flags are used by the JC, JZ, and JN instructions, respectively to make conditional jumps, making the SAP1 Turing equivalent.

**Control.** The functioning of the SAP1 is governed by the control logic and control signals. There are 16 control signals. These signals control:

1. The input and output of each module. The control logic assures that only one output and one input are enabled at a time so that there are no bus conflicts.
2. The input enable for the Flags register so that flags are set at the right time in an instruction and not overwritten before the JC and JZ instructions use them.
3. The HLT signal that stops the clock when the HLT instruction is executed.

The SAP1 is a modern microcoded processor which uses an internal ROM that encapsulates the control logic. Its values are played out by a microstep clock and register. Each instruction takes one or more microsteps plus two microsteps for the instruction fetch and decode. Microsteps are done on the falling edge of the clock, whereas bus cycles and memory accesses happen on the rising edge. You can see the entire microcode ROM at the end of the SAP1 source code in a series of data statements.

### The SAP1 Instruction Set

The 4-bit Instruction Register opcode space means that the SAP1 can have up to 16 instructions. However, only 12 instructions are implemented:

| OpCode | Mnemonic | Action  |
|--------|----------|---|
| 00     | NOP      | No op. Does nothing in 3 clock cycles (2 for fetch, 1 for instruction) No operand.  |
| 01     | LDA addr | Load a value from memory address 'addr' into the A Register.  |
| 02     | ADD addr | Add the value in memory address 'addr' to the A Register. The memory value will be loaded into the B Register. The Sum Register will then compute the sum, which is then put into the A Register on the next clock cycle.   |
| 03     | SUB addr | Subtract the value in memory address 'addr' from the A Register. The memory value will be loaded into the B Register and then 2's complemented; that is, its 1's complement will be formed and then 1 added to it. The Sum Register will compute the difference, which is then put into the A Register on the next clock cycle. |
| 04     | STA addr | Copy the contents of the A Register into the memory at address 'addr'.  |
| 05     | LDI val  | Load immediate: the 4-bit value 'val' is put into the A register. This value can only range from 0 to 15. Negative values are not allowed.  |
| 06     | J addr   | Unconditional jump to address 'addr'  |
| 07     | JC addr  | Conditional jump to address 'addr' if the Carry Bit is set in the Flags Register.   |
| 08     | JZ addr  | Conditional jump to address 'addr' if the Zero Bit is set in the Flags Register   |
| 09     | JN addr  | Conditional jump to address 'addr' if the Negative Bit is set in the Flags Register   |
| 10-13  |          | Not Used  |

|    |     |   |
|----|-----|---|
| 14 | OUT | Copy the contents of the A Register to the Output register. No operand. |
| 15 | HLT | Stop the Clock and Halt the Computer. No operand.                       |

Even with this very small instruction set, it is possible to write programs that do more sophisticated things. The sample programs include one that multiplies two values in memory, and one that computes and displays the digits in the Fibonacci series. You could write a program for integer division that would fit into memory. In fact, the 16-location memory is more of a limitation than the instruction set. The 4 values that currently have no instructions could be used for various things, including logic operations such as AND, OR, XOR, and NOT. (left as an exercise for the reader)

### The SAP1 Cycle

In order to understand what 'Das BlinkenLights' are telling you, it is necessary to understand how the SAP1 executes instructions. Let's start with the LDA instruction. Here are the microcode ROM values that control what goes on when the LDA instruction is executed:

|        | H | L  | R  | I  | C  | C  | A  | E  | F  | S  | M  | R  | I  | A  | B  | O |
|--------|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
|        |   | T  | O  | O  | O  | E  | O  | O  | I  | U  | I  | I  | I  | J  | I  | I |
| 99, 0, |   | 0, | 0, | 0, | 1, | 0, | 0, | 0, | 0, | 0, | 1, | 0, | 0, | 0, | 0, | 0 |
| 99, 1, |   | 0, | 1, | 0, | 0, | 1, | 0, | 0, | 0, | 0, | 0, | 0, | 1, | 0, | 0, | 0 |
| 01, 4, |   | 0, | 0, | 1, | 0, | 0, | 0, | 0, | 0, | 0, | 1, | 0, | 0, | 0, | 0, | 0 |
| 01, 4, |   | 0, | 1, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 0, | 1, | 0 |

The letters in the first three rows are the names of the control signals:

|     |   |
|-----|---|
| HLT | The Halt signal that stops the computer   |
| RO  | Assert the contents of the Memory Register onto the bus.                          |
| IO  | Assert the contents of the lower 4 bits of the Instruction Register onto the bus. |
| CO  | Assert the contents of the Program Counter onto the bus.                          |
| CE  | Increment the Program Counter.  |
| AO  | Assert the contents of the A Register onto the bus.                               |
| EO  | Assert the contents of the Sum Register onto the bus.                             |
| FI  | Put the bus value into the Flags Register   |
| SU  | Assert the Subtract signal  |
| MI  | Put the bus value into the Memory Address Register                                |
| RI  | Put the bus value into the Memory Register  |
| II  | Put the bus value into the Instruction Register                                   |
| J   | Put the operand value of the instruction register into the Program Counter        |
| AI  | Put the bus value into the A Register   |
| BI  | Put the bus value into the B Register   |
| OI  | Put the bus value into the Output Register  |

Every instruction cycle begins with a 2 clock cycle fetch. These are the two rows that start with '99'. A fetch is required because, after all, the computer doesn't know what instruction will be executed until is

is read in from memory and put into the Instruction Register. The first '99' row has two control bits set: the CO and MI bits. These copy the contents of the Program Counter 'C' into the Memory Address Register 'M' by asserting the Program Counter's value onto the Bus and then reading the Bus value into the Memory Address Register. The second '99' row completes the fetch by asserting the value that comes from memory and lives in the Memory Register onto the bus and copying the Bus value into the Instruction Register, Note that this is an 8-bit data transfer so both the instruction opcode and the operand value are now in the Instruction Register. Also, the 2<sup>nd</sup> fetch cycle increments the Program Counter so that it points at the next memory address. (CE signal)

Following the fetch, the LDA instruction is now executed in 2 clock cycles. The first cycle raises the IO and MI control bits, which copies the operand value from the lower 4 bits of the Instruction Register into the Memory Address Register. The second cycle puts the RAM value at that address that is housed in the Memory Register onto the bus and then copies the bus value into the A register.

Note that although there are 4 different 'jump' instructions (J, JC, JZ, JN) there is only a single 'J' control line. There is additional 'instantaneous' logic that handles the conditional jumps.

You can see the entire control ROM values in the data statements at the end of the SAP1 source code. It is very instructive to single-step the SAP1 computer through a number of different instructions and see how the control bits route the data where it is needed.

## The Assembler

The Assembler reads a program source file ('.asm') and outputs a RAM image file ('.bin') Note that the '.bin' output file is actually readable text, but when it is read into SAP1, it is converted into real binary numbers.

Here is a sample assembly language source file:

```
; Tiny test program for the SAP1 computer. Add 2 numbers in memory,  
; put the result into the output register, and halt.
```

```
    .org 0  
    LDA X  
    ADD Y  
    OUT  
    HLT  
    .org 14  
X:   28  
Y:   24
```

This little program contains examples of pretty much the entire repertoire of the assembler:

- Comments: any text preceded by a semicolon is a comment and will be ignored.
- Directives: any word preceded by a period (.) is an assembler directive. Currently there is only one directive, '.org'; it tells the assembler to set the current address to the operand of the directive.

- Labels: Any word that starts in column 1 and ends with a colon is a label. A label has a value that is the current assembler address where the label is located. Labels may be referred to before they are defined, but they must be defined somewhere in the code.
- Instruction Mnemonics: One of the 12 defined instruction mnemonics. Cannot start in column 1, and must be capitalized.
- Instruction operands: follow the Instruction Mnemonics with at least 1 intervening white space character. Can be either a decimal literal or a Label reference.

The first two comment lines and the following blank line are ignored. The `.org 0` says to start the assembly at address 0. (This is the default anyway.) The 4 instructions in the program follow. They say Load the contents of the Memory location with the Label 'X' into the A register; then Add the contents of the Memory location with the Label 'Y' into the B register. This causes the Sum register to have the sum of A and B, which then gets automatically loaded back into the A Register. The OUT instruction copies the value in the A Register into the Output Register, which causes the sum to be displayed in the simulated output display. This should be decimal 42. The HLT instruction then halts the computer. The Labels 'X' and 'Y' are equivalent to the memory addresses of the two values to be added. So when the computer executes `LDA X`, it means "load the value of the memory location of the label X". The value of 'X' resolves to 14 decimal or 1110 binary. The value loaded at address 'X' is 28 decimal.

To use the Assembler, start the 'Assembler.bas' program. It will present you a list of the available assembly source files in the 'Programs' directory. Use the UP and DOWN keyboard arrow keys to navigate to the program you want to process and then press the ENTER key to assemble the code and write the '.bin' output file in the Programs directory. Any problems with the source will cause an error message to be printed.

If the assembly is successful, then the Assembler will print out an 'annotated' version of the source code onto the screen. This shows the original source code plus the resulting binary image. Here is the one for the tiny test program:

```
; Tiny test program for the SAP1 computer. Add 2 numbers in memory,
put the result into the ouput register, and halt.
```

```

.org 0
LDA X      0000 00011110
ADD Y      0001 00101111
OUT         0010 11100000
HLT        0011 11110000
.org 14
X:  28      1110 00011100
Y:  24      1111 00001110
```

The two banks of binary numbers are the memory address and the memory contents at that address. Note that only instruction lines that consume memory space have the numeric fields to their right. However, if we look at the '.bin' file that is output by the Assembler, we will see that all 16 memory locations are present (just their values, not the addresses), with the locations not used set to zero:

```
00011110
00101111
11100000
11110000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00011100
00001110
```

### Sample Programs Included

- Test2 -- the tiny test program described above
- CountUpDown -- Counts forever up and down over the entire unsigned 8-bit range (0 to 255)
- Multiply -- Multiplies two integers X and Y and displays the product
- Fibonacci -- Computes the values in the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13...
- GCD -- Computes the Greatest Common Divisor of 2 integers and displays it

-- End of Instructions --