# Simple as Possible – 1 (SAP-1) Computer

This is a companion paper for the Ben Eater project kit to build the SAP-1 from the book “Digital Computer Electronics” by Albert Paul Malvino (ISBN 0070398615 / 978-0070398610). This book is getting pretty rare, and it can be quite expensive. Even the “new” version is close to $200 as of the time I am writing this.

Prior to reading this document, make sure you have studied the “Foundation of Boolean Logic” paper also in this repository.

The kit is available at this time from <https://eater.net/8bit/kits> and is $299.96. You will need a few tools, and Ben has those documented on his site and in his youtube videos.

Looking at our implementation of the “Simple as Possible – 1” computer, we have these core modules:

* Master Clock
* Program Counter
* Memory Address Register
* Instruction Register
* “A” Register
* “B” Register
* Accumulator (ALU)
* Instruction Decoder (EEPROM)
* Instruction Clock and T-Counter
* Random Access Memory

And these “Helper” modules:

* RAM Programming board
* Output Display Module
* Microcode Control Signal handler

All of these connect to the central “Data Bus”.

We also have a “RESET” circuit to get us out of trouble while testing our computer, and all the “Control Signal” wiring between the components.

Let’s examine each of these in detail.

## Master Clock

Lets start with the electrical signal for our clock. It starts at “0” volts, and raises to the requisite voltage for our circuit to detect. This is usually +5V DC, although in some chips it is quite a bit lower about 3.3V or even less.

In our example we see that it “rises” and “falls” and holds steady for a bit. This particular example is known as a “Square Wave” as that is how it appears on an oscilloscope.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

Square Wave Clock Signal

Here is a signal being generated by a LM555 Timer chip (although not quite square) using a Hantek 2D72, showing the capacitor charge/discharge cycle in green and the resultant triggered clock in yellow:

Graphical user interface

AI-generated content may be incorrect.

For the purposes of our computer, we break this into 2 distinct phases – the “fall and hold” and the “rise and hold”:

|  |  |  |
| --- | --- | --- |
|  |  |  |

We will refer to these phases as our clock “Tick” (fall) and “Tock” (rise). This set of 2 is known as a clock cycle time for an instruction.

This is critical to the operation of the microcode layer as it performs its operations of “Raise Signals” during the tick and “Set Values” during the tock.

This tick-tock clock is expanded by counting the ticks with a separate counter circuit which is called the “tick clock” or “T” clock. It runs from T0 to however many cycles an instruction requires.

## Counters

After the clock creates the heartbeat of our computer, we have a pair of circuits that use those clock cycles to “step through” processes by counting how many cycles have occurred for our sequence of operations, or to step through memory to fetch the next instruction to be executed.

#### Instruction Cycle Counter (T-Clock)

The Instruction Cycle “Tick” counter, or “T-Clock” drives the circuitry in a sequence of “Signal”/”Set” cycles to perform the requisite steps to effectively execute the machine/assembly instruction. This clock runs from “T0” to “T6” on our computer as we have it set to execute no more than 6 steps per instruction. This clock automatically updates itself from the “Tick” of the master clock, and drives the instruction decoding. If we expanded our Decoder to a 3rd EEPROM, we could add a signal for T-Clock reset so each instruction would only use the number of cycles it requires instead of our fixed 6-cycle counter. It uses both a standard 4-bit counter and a 3-to-8 demultiplexer. The output of the demultiplexer is used by the instruction decoder to perform the control signal raise/set cycles.

#### Program Counter

The program counter is setup with a “Counter Enable” signal that causes it to increment by 1 on the next clock “Tock” phase. As we have a 16-byte memory limit, this is implemented as a simple 4-bit binary counter; however it can be externally changed to implement a “Jump” instruction, which sets the new address of the next instruction to be executed. This is done with a standard “Write Enable” signal. Its primary control signal is “Counter Out” which places its value on the data bus.

## Registers

Let’s begin with the definition of what a Register is. In most computers, these are storage bytes, but are outside of the Random Access Memory, and are actually part of the circuitry of the CPU chip itself. They are just scratchpads that are used to hold values until modified or read. There are many standards for naming these – letters (A – L), numbers (0-15), or combinations (FP0 – FP4).

Our registers have the standard “In” and “Out” control signals to either load a byte from the data bus, or place the current value on the bus.

Registers just hold a byte (or more) of data. How that data is interpreted is based on the usage of that register. Many are used for holding “memory addresses” and many for “data values”. Let’s take a look at ours:

#### Memory Address Register

The Memory Address Register (MAR) is the gateway to the Random Access Memory. Whatever the value in this register is, that is the byte that can be read or written in the RAM modules. It has a “Memory In” signal to save the address on the data bus.

#### Instruction Register

The Instruction Register holds the byte of data that is the instruction to be “decoded” into the sequence of circuit Signal/Set to effectively perform the machine instruction. As our instructions are 4-bit Operation Code and 4-bit Value or Address, this register has control signals for “In” and “Out”.

#### Output Register

This registers value is displayed on the 7-Segment LCD panels as either a 0 – 255 value, or as a signed integer (-128 to +127 ). As it is just a display, it only has the “Output In” signal.

#### “A” Register

The primary data register in our computer. This has both “In” and “Out” signals.

#### “B” Register

This is the secondary data register, and its value is automatically added or subtracted from the “A” register by the ALU. As we are limited to 16 signals, we only implement the “In” signal on this register.

## Arithmetic/Logical Unit

The heart of any digital computer is the Arithmetic and Logical unit. It performs the addition of registers, or the binary operations on data values (AND, OR, NOT, XOR) and holds the result. Its core is the Full-Adder circuit which is discussed in the lab module building this circuit. Our ALU has control signals for placing its value on the Data Bus (Sum Out), for Subtraction (SUB), and for setting result flags for ZERO and CARRY from the result of the last operation.

## Instruction Decoder

The instruction decoder uses the value in the instruction registers high nybble as the operation code to be performed, the “flags” (Zero and Carry), and the “T-Clock” counter to drive the microcode steps to be executed in sequence.

# Our Machine Instruction Codes

When Assembly/Machine instructions are “decoded” to be performed, the bits that are the actual instruction are commonly called the “Operation Code” and the value bits are called the “Operands”.

Our computer uses only 4 bits for the Operation Code, so we have 16 possible instructions. This chart shows the mnemonic, the binary code value, and its definition.

## Operation Code List

NOP 0b0000 – No Operation

LDA 0b0001 – Load A register from memory

ADD 0b0010 – Add the B register to the value in the A register.

SUB 0b0011 – Subtract the value in the B register from the value in the A register.

STA 0b0100 – Store the value in the A register in memory at the defined address

LDI 0b0101 – Load the 4 lower bits of this instruction directly into the A register

JMP 0b0110 – Jump to the address in the lower 4 bits

JC 0b0111 – Jump if the carry flag is set to the address in the lower 4 bits

JZ 0b1000 – Jump if the zero flag is set to the address in the lower 4 bits

IC9 0b1001 - not yet defined

ICA 0b1010 - not yet defined

ICB 0b1011 - not yet defined

ICC 0b1100 - not yet defined

ICD 0b1101 - not yet defined

OUT 0b1110 – Output the value in the A register to the output display

HLT 0b1111 – Halt the computer clock

# Introduction to Microcode

When the circuitry is being used to actually implement the operation code, it is referred to as “microcode”. It is a sequence of events that raise various signals to different circuits, and then trigger those signals to move data, perform I/O, or control external hardware. Our SAP-1 computer has 16 control signals as we are using a pair of EEPROM chips to drive our processor, and each chip outputs one byte of data

## Microcode Control Signal List

These are the “control signals” that are used by the circuitry of our SAP-1 computer:

HLT - Halt clock

MI - Memory address register in

RI - RAM data in

RO - RAM data out

IO - Instruction register out

II - Instruction register in

AI - A register in

AO - A register out

EO - ALU out

SU - ALU subtract mode

BI - B register in

OI - Output register in

CE - Program counter enable

CO - Program counter out

J - Jump (program counter in)

FI - Flags in (set the ZERO and CARRY OUT flags)

As every instruction execution starts with the same process (Fetch Instruction) we will learn it by examining the components of the computer required and how they interact.

An “Instruction Fetch Cycle” gets the next instruction to be executed from memory by using the “Program Counter” register, the “Memory Address Register”, the “Instruction Register” and the data bus.

The “Program Counter” must ALWAYS contain the address of the next instruction to be executed, so keeping it updated is part of fetching an instruction.

The process is:

* Get Program Counter value
* Place into Memory Address Register
* Get data byte from that memory address
* Put data byte into the Instruction Register
* Update the Program Counter value

In our implementation of the “Simple as Possible – 1” computer, this only requires 2 instruction cycle times. And this will be the microcode for the first 2 Tick Clock cycles of EVERY instruction.

Let’s look at our “Control Signals” to see what we need to do for this. We simply “OR” together all the control signals we need for each microcode step.

First, getting the value from the program counter. That is done by raising the “Output Enable” signal on the transceiver attached to the PC register. In our microcode we call this “Program Counter Out (CO)”. When this signal is raised, it immediately places the value of the register onto the data bus. And we also need the “Memory Address Register In (MI)” signal to be raised. At that point, we “tock” the clock and the value is moved from the data bus into the Memory Address Register (MAR).

Our T0 microcode signal is: CO | MI

Second, getting the value from RAM into the Instruction Register and updating the Program Counter.

This involves 3 control signals – “Ram Out (RO)”, “Instruction Register In (II)”, and “Counter Enable (CE)”. These 3 signals are raised during our T1 “tick”. The “Ram Out” signal immediately places the value from the pre-set address in the MAR onto the data bus. The “Instruction In” will store the value on the bus in the Instruction Register for decoding. The “Counter Enable” signal on the Program Counter will cause that counter to increment on the T1 clock “tock”.

Our T1 microcode signal is: RO | II | CE

From this point, the “T” clock will continue to process the decoded machine instruction until it gets to the “done” point. Machine instructions take varying numbers of “T” clock cycle times, and these are documented in the Assembly Language manuals of computer chips and systems.

Just for fun, now that we did nothing, let’s add the HALT to our NOP and run those 2 in our computer. Remember that all instructions start with the same “instruction fetch cycle”, so we already know:

Our T0 microcode signal is: CO | MI

Our T1 microcode signal is: RO | II | CE

Our HLT instruction just adds one more microcode step, with one signal:

Our T2 microcode signal is: HLT

So, HALT requires 3 cycles and is quite easy. If we ran this, our Program Counter would have the value of 2, the Memory Address Register would have the value of 1, the Instruction Register would have the “HLT” instruction code, and our clock would be stopped.

Let’s try a real example – load a value from memory into the “A” register, and display it on the output display unit.

The “Assembly/Machine” code for this is 3 lines:

LDA 2

OUT

HLT

This is Load the “A” register from the value at memory address 2; place the value in register “A” into the output display; then halt the clock.

Let’s decode the “LDA 2”. This is “Load the A register from memory address 2”. So that means our instruction contains the Operation Code “0001” and Operand (data address) “0010”. We will need to get that address out of the instruction and back into the memory address register to then get the value from that address and put it into register A. Although it sounds like a lot, it is only 2 microcode steps.

Fetch

Our T0 microcode signal is: CO | MI

Our T1 microcode signal is: RO | II | CE

Get the address from the instruction back into the memory address register – so “instruction out (IO)” and “Memory Address In (MI)”:

Our T2 microcode signal is: IO | MI

Then, from memory into the “A” register – RAM Out (RO), A In (AI):

Our T3 microcode signal is: RO | AI

So our “Load A” instruction uses 4 microcode cycles – the 2 Instruction Fetch, and 2 to perform the actual instruction.

Moving on to the “OUT” instruction, it needs to get the value out of the “A” register, and put it into the Output Display register. That will be all of one microcode step after the Fetch

Fetch

Our T0 microcode signal is: CO | MI

Our T1 microcode signal is: RO | II | CE

Now we need “A OUT (AO)” and “OUTPUT DISPLAY IN (OI)”:

Our T2 microcode signal is: AO | OI

And we already know how to do “HALT”

Our T3 microcode signal is: HLT

Run this, and the display will show “240”, which in hex is “F0”. That is the “HLT” instruction code. Yes, we used an instruction (HLT) as a data value to be displayed. Bytes are just Bytes. They are only instructions when processed by the Instruction Decoder.

This should scare you. The Instruction Register does not care if you accidentally put what you think is data in it. It will decode it and do stuff. Data Registers do not care if you put addresses in them. They will happily add them together whether the result is what you expected or not.

All data is just that. Raw Data. Until something interprets it, it has no “meaning”.

There is a very old and very politically incorrect joke about data and information.

*362436 is just raw data. It carries no meaning, and puts no image in your mind.*

*362,436 – we just added a comma, but now you think of it as a value. It starts to mean something.*

*36-24-36 – now, that is information!*

And most of you reading this are probably too young to get that reference. And I’m too old to care if you do understand it and it offends you. It is a simple example that really explains the difference in the terms data and information.

## Microcode Instructions

Let’s take each of our assembly/machine language instructions, and learn the microcode steps for them.

### NOP

The “No Operation” instruction merely performs the Instruction Fetch cycle; this is frequently used as a delay step, or used to “reserve” a few bytes of memory for later modifications.

As such, it is only the 2 steps:

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

### LDA

The “Load A Register” instruction uses the operand as the address of a value to be loaded into the A register. As such, it does the Instruction Fetch, then uses its operand to load the MAR, then move the value from RAM to the A register:

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

IO|MI – Instruction Register Out; Memory Address Register In

RO|AI – RAM Out; A register In

### ADD

The “Add” instruction uses the ALU to perform the addition via its full-adder circuit. It uses the operand as the address of the value to be added to the current value of the “A” Register. This value is placed into the “B” Register, the ALU automatically adds these values, we get the output of the ALU, and place that back into the “A” register. We also set the Carry and Zero flags based on the result of the Addition.

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

IO|MI – Instruction Register Out; Memory Address Register In

RO|BI – RAM Out; B register In

EO|AI|FI – ALU (Sum) Out; A Register In, Set the Carry and Zero flags

### SUB

The “Subtract” instruction also uses the ALU to perform what appears to be a subtraction. However, the ALU only contains a full-adder circuit, so uses the grammar school trick of performing subtraction by adding the negative of the value. Remember kids – “Ten Minus Five” is the same as “Ten Plus Negative Five”. Setting the “SU” bit uses an inverter chip to flip the value, and raises the “Carry In” to make it the “Two’s Complement” in binary. So our code is nearly identical to the ADD above, just with the “SU” signal added:

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

IO|MI – Instruction Register Out; Memory Address Register In

RO|BI – RAM Out; B register In

EO|AI|SU|FI – ALU (Sum) Out; A Register In, Set the Subtract signal, Set the Carry and Zero flags

### STA

The “Store A register at address” instruction uses the operand as the address where the value in the A register is to be stored. As such, it gets the address from the instruction, then gets the output of A and places it into RAM.

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

IO|MI – Instruction Register Out; Memory Address Register In

AO|RI – A Register Out; RAM In

### LDI

The “Load Immediate Value in A Register” uses the operand itself as the 4-bit value to be loaded into the A register. We use the normal “Instruction Register Out”, but instead of loading it into the MAR, we just put it in A:

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

IO|AI – Instruction Register Out; A Register In

### JMP

The “Jump” instruction and its variants (Jump on Carry, Jump on Zero) update the Program Counter so the address of the “next” instruction to be fetched is changed. This uses the operand as the 4-bit value to be loaded into the Program Counter.

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

IO|J – Instruction Register Out; “J” (Program Counter In)

### JC and JZ

The “Jump on Carry” and “Jump On Zero” rely on the settings of these flags. If set, then the jump (IO|J) is performed; if not set, then the instruction is basically a “NOP”.

### OUT

The “Output” instruction places a value into the register driving the 7-Segment display unit. It takes whatever value is in the A register and puts it into the Output Register

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

AO|OI – A Register Out; Output Register In

### HLT

Our “Halt” instruction stops the master clock.

CO|MI - Program Counter Out; Memory Address Register In

RO|II - RAM Out; Instruction Register In

HLT – Stop the Master Clock

## Storing Microcode Signals in an EEPROM

This is the signal coding for the EEPROMs to drive the microcode execution steps. Note that every instruction starts with the 2 step “Instruction Fetch”. The code to write the EEPROMs adds the correct signals for the Jump on Carry and Jump on Zero based on the flag values. Based on the flag values for the Zero and Carry flag, certain sets will have step T2 set to “IO | J” which takes the 4-bit operand from the Instruction Register and places it into the Program Counter. There are 4 sets of these instructions created in the EEPROMS for neither flag set, Zero set, Carry set, and both set. You will see these in the code as Z0C0, Z1C0, Z0C1, and Z1C1.

CO|MI, RO|II|CE, 0, 0, 0, 0, 0, 0 // 0000 - NOP

CO|MI, RO|II|CE, IO|MI, RO|AI, 0, 0,0,0 // 0001 - LDA

CO|MI, RO|II|CE, IO|MI, RO|BI, EO|AI|FI, 0, 0, 0 // 0010 - ADD

CO|MI, RO|II|CE, IO|MI, RO|BI, EO|AI|SU|FI, 0, 0, 0 // 0011 - SUB

CO|MI, RO|II|CE, IO|MI, AO|RI, 0, 0, 0, 0 // 0100 - STA

CO|MI, RO|II|CE, IO|AI, 0, 0, 0, 0, 0 // 0101 - LDI

CO|MI, RO|II|CE, IO|J, 0, 0, 0, 0, 0 // 0110 - JMP

CO|MI, RO|II|CE, 0, 0, 0, 0, 0, 0 // 0111 - JC

CO|MI, RO|II|CE, 0, 0, 0, 0, 0, 0 // 1000 - JZ

CO|MI, RO|II|CE, 0, 0, 0, 0, 0, 0 // 1001

CO|MI, RO|II|CE, 0, 0, 0, 0, 0, 0 // 1010

CO|MI, RO|II|CE, 0, 0, 0, 0, 0, 0 // 1011

CO|MI, RO|II|CE, 0, 0, 0, 0, 0, 0 // 1100

CO|MI, RO|II|CE, 0, 0, 0, 0, 0, 0 // 1101

CO|MI, RO|II|CE, AO|OI, 0, 0, 0, 0, 0 // 1110 - OUT

CO|MI, RO|II|CE, HLT, 0, 0, 0, 0, 0 // 1111 – HLT

So I hope this gives you a view into the internal workings of a computer and how the circuits are triggered to perform their magic.