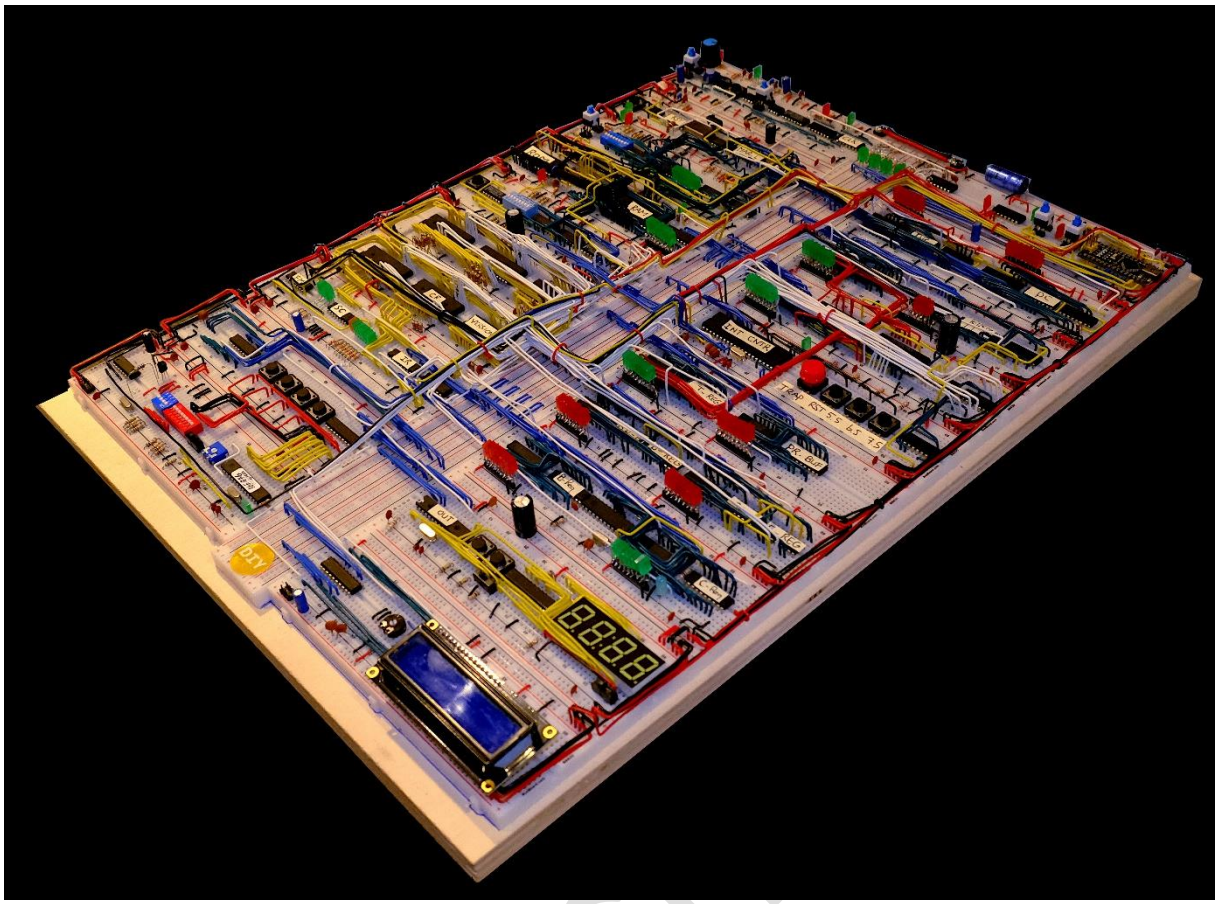
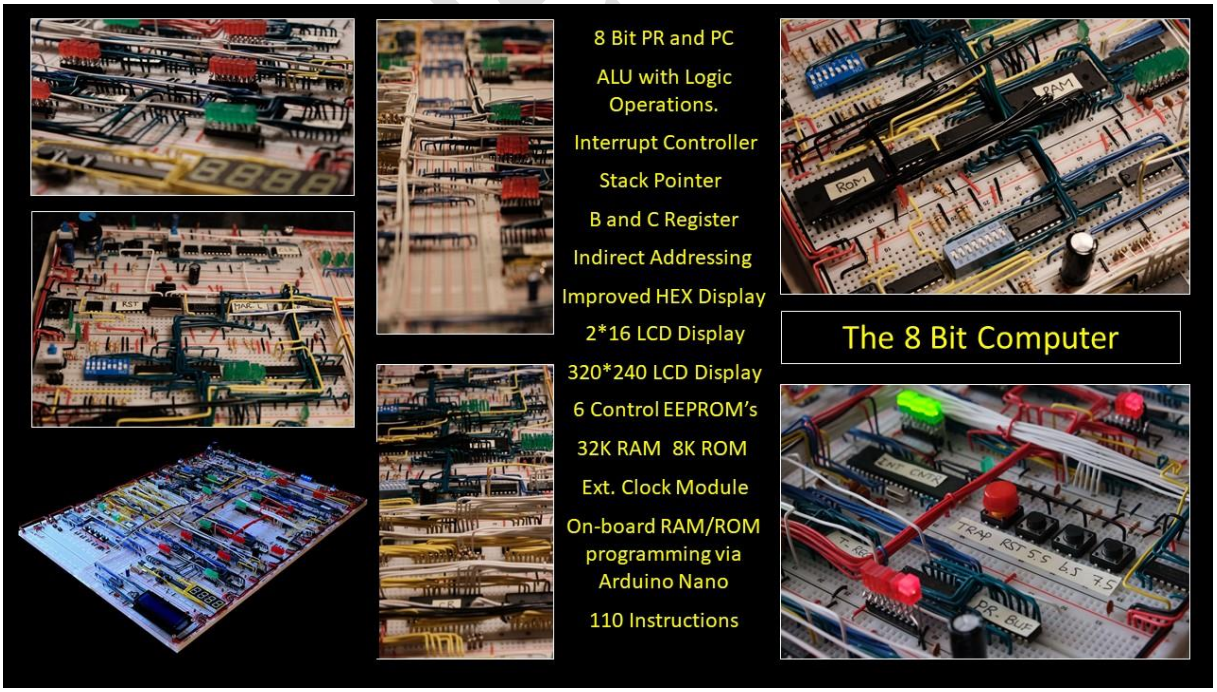
Building The SAP-3+ Computer

By: R.J.Dubbeld
Hoogvliet, The Netherlands
rolf-electronics, used on Reddit and Github
https://github.com/rolf-electronics/The-8-bit-SAP-3+

Rev:	1.0	27 April 2020	
Rev:	2.0	1 May 2020	
Rev	2.2	8 May 2020	
Rev	2.3	9 May 2020	
Rev	2.4	15 May 2020	
Rev	3.0	23 May-2020	
Rev	3.1	Febraury 2021	
Rev	3.2	March 2021	Added descriptions for the HEX display, Analog output, the MNEMONICS display and grammar improvements by Greg H.
Rev	3.3	June 2021	Added the programming via CUSTOMASM



The end result, only missing the 320*240 LCD screen.



My 8 bit computer with all the specifications. (missing analog input and output, the MNEMONICS and PR/PC display and programming in assembly.

Table of Contents

Chapter 1, The Start

Ben Eater has made a very good series on YouTube on building an SAP-1 computer, where SAP stands for Simple as Possible. The idea is to build a computer from first principles with integrated circuits as available in the 1980's. I am going to be deviating from that in building the SAP-3 plus.

The basis of Ben Eater's design is extensively described by Albert Paul Malvino in his book Digital Computer Electronics. A similar build can be found on http://buildyourcomputer.org/aGuideToComputerScience.pdf. This is a good written introduction in SAP-1, explaining from first principles how a simple computer works.

Also the book, An introduction to microprocessors from D.K.Kaushik describes the SAP-1. This books as well as the book from Malvino describes not only the SAP-1 but also the more capable SAP-2 and SAP-3.

The differences between SAP-1, 2 and 3 will be discussed in the next chapters.

The purpose of this document is to describe my design and build of the SAP-3 computer starting from the SAP-1. This also means that nothing will be repeated or explained from the SAP-1 computer since that's all done by Ben Eater on YouTube.

It is a must to fully understand all aspects of SAP-1 before proceeding to SAP-2 or 3. Without the basic knowledge of SAP-1 it's not recommended to build SAP-2 or SAP-3. It can of course be done, but troubleshooting will be far more difficult if not impossible and the learning experience is far less.

I also strongly recommend any builder to read and implement the lessons learned from u/lordmonoxide on the Reddit sub channel of Ben Eater.

This is how I built it, other solutions or directions to build the SAP-3 are always possible.

Is this a project for people without any experience in electronics, well most likely it is not going to be easy for several reasons. Building an SAP-1 with all video's and schematics as provided by Ben Eater and all the info which you can find on the Reddit sub channel, it has shown that it's challenging for absolute beginners. This I base on the type of questions being posted. (not meant to be un-nice)

Upgrading to SAP-2 or 3 means that you have to start designing yourself, which is something totally different than building from schematics made by others. There is not much detailed info to find on the internet, at least not complete builds. You also might be faced with the situation that you can find some sort of design, but cannot find the right components anymore. This would also force you to make design modifications. On top of that when you find something on the internet what's the guarantee that the schematics are correct. How would you know how it's supposed to work, if what you found and build doesn't work. For instance I posted my solution for an upgraded RAM, finding out later that there was a mistake in the drawing. I built it properly since it worked. I made a first idea with some scribbled notes and then started building. Than someone asked how did you build it? So I made a "proper" schematic, unfortunately it had a mistake. (correction is included in this document)

Additionally as long as everything works what you found and built you're ok. But once it doesn't you're in the field of troubleshooting, which is a lot more difficult than just building. On top of that troubleshooting without some equipment is really no fun. But what's the chance that someone without any electronics background has a descent oscilloscope, a multi-meter, and a logic analyser.

Not meant to scare anybody off, just be warned.

Chapter 2, SAP-2

SAP-1 can be summarized as follows:

- 4 bit program counter,
- A and B register,
- ALU with addition and subtraction function,
- 1 HEX display,
- Instruction register,
- 16 byte RAM,
- Maximum of 16 instructions,
- Clock Module up to 300 Hz.

In the book from Malvino there is no summary of the characteristics of SAP-2. So I went through the complete chapter and came up with the following:

- Variable machine cycle,
- Includes Jump instructions, Ben Eaters SAP-1 computer already has this,
- 16 bit program counter, PC,
- 16 bit memory address register, MAR,
- 16 bit W-Bus,
- 2K ROM from 0000[h] to 07ff[h], 8 bit wide (to initialize the computer etc.),
- 62K RAM from 0800[h] to FFFF[h], 8 bit wide,
- Memory Data Register MDR,
- Instruction register 8 bit,
- Temporary 8 bit register connected to the 8 bit ALU,
- 8 bit B and C register,
- Two input ports. One HEX display, one serial,
- Two output ports. One Hex keyboard entry, one serial,
- ALU with arithmetic and logical operations, control by means of control bits,
- Two flags zero and sign. Ben Eaters SAP-1 already has a zero and carry flag,
- 44 Instructions,
- Maximum of 18 T states, which is mainly for the CALL instruction

Chapter 6, Choices I made building SAP-2 discusses all above items in detail.

In my build I did not implement the CALL instruction as described for SAP-2. The use of the Stack Pointer mechanism in SAP-3 is far more efficient and more flexible to save a return address.

Chapter 3, SAP-3

In the book from Malvino there is no summary of the characteristics of SAP-3. So I went through the complete chapter and came up with the following:

- Four additional registers D, E, H, L
- 16 bit Stack Pointer
- More flags, carry and parity
- 33 extra instructions
- Functionality to use register pair DE and HL

In general it can be concluded that the step from SAP-1 to SAP-2 is a lot larger than from SAP-2 to SAP-3. Does the computer work:

https://www.reddit.com/r/beneater/comments/fdix5i/someone_posted_what_kind_of_programs_can_be_made/

This practically uses everything: most of the instructions, PR/PC, ALU, LCD Display, RAM/ROM, the A,B and C register and the stack pointer.

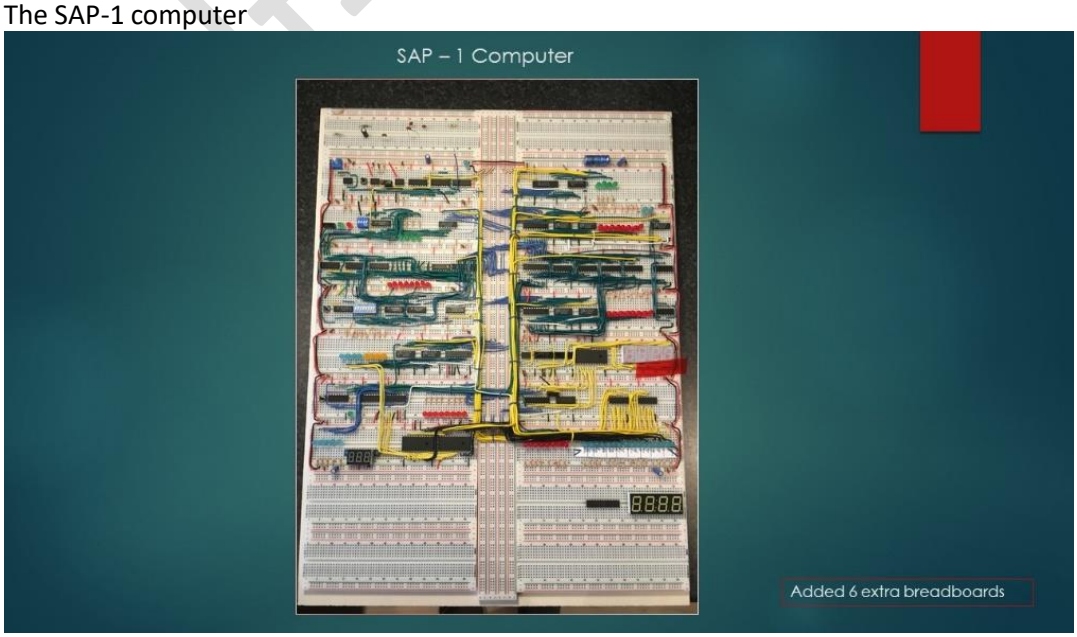
Chapter 4, Add-ons

Additional functionality I built include:

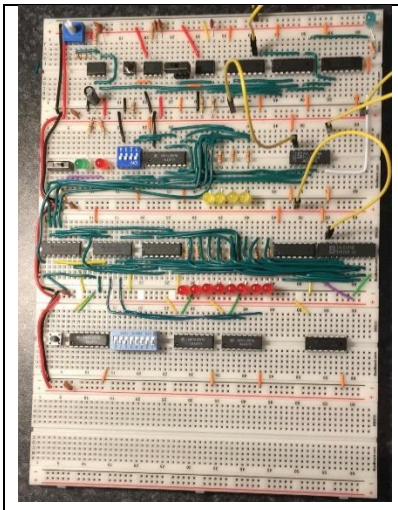
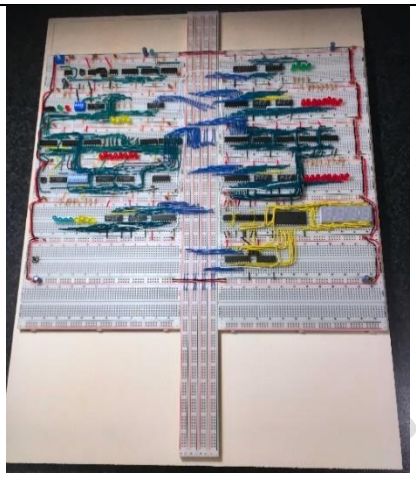
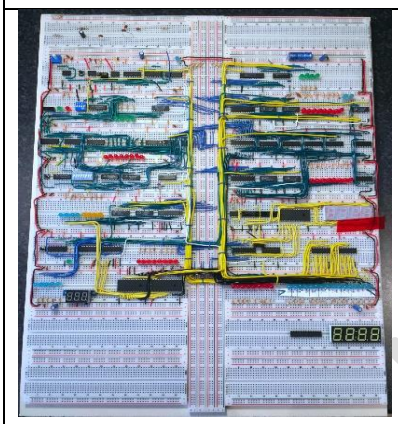
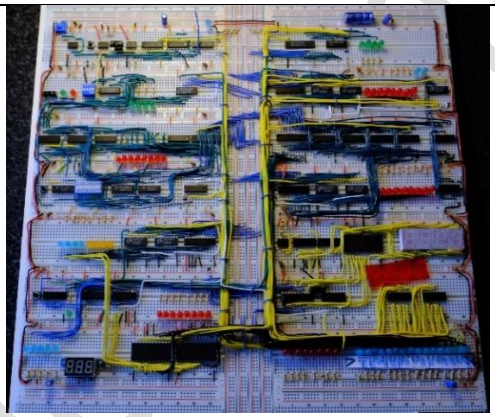
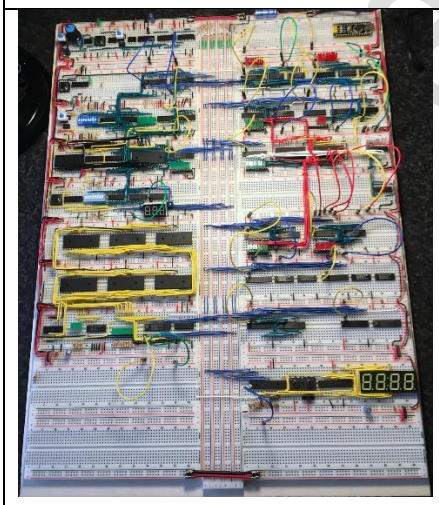
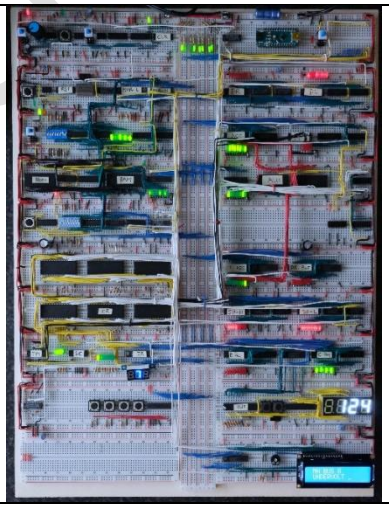
- Use of a device to program the RAM and ROM on-board. Especially for programming the RAM another solution has to be found. With 32K RAM and 8 bits one has to toggle a quarter of a million DIP switch settings to enter a program. Even programming one page, 256 bytes would already be 256*8 ~2000 DIP switch settings. Debugging and altering is even worse.
- An interrupt controller like the 8085
- A 2*16 LCD screen, a 320*240 LCD screen and HEX indication on the 7 segment display
- An analog input and output
- Fixed crystal controlled clock speed to allow for time critical applications.

Details are provided in Chapter 12, Additions to the SAP-3.

The SAP-1 computer



Chapter 5, Evolving from SAP-1 to SAP-3

Pictures speak for themself.

Chapter 6, Choices I made building SAP-2

The following choices were made by me in building the SAP-2. Beside the choices I made there is also more detail on each of the subjects on why and how I did certain things.

(*) Deviates from the Malvino design.

- The baseplate and breadboards

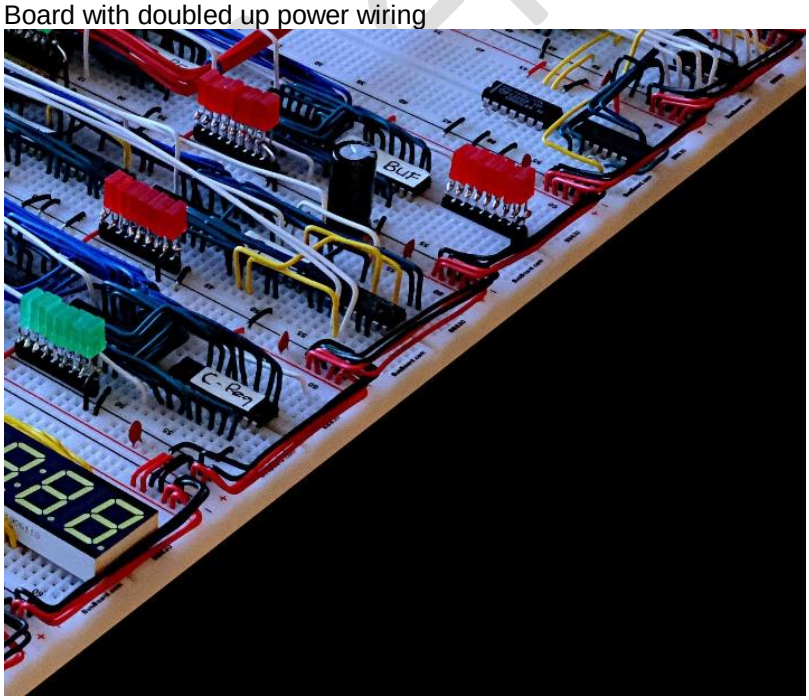
I made a plywood base of 51 (height)* 38 (width)* 15 (thickness) cm. On that I mounted 20 BB830 breadboards according the system as used by Ben Eater. So between two boards there is one power rail. Best is to first draw some marking lines on the plywood on where to align the breadboards. Not doing so could lead up to a serious misalignment on the bottom if you start at the top, or the other way around. And once you mount a breadboard on the plywood they cannot be taken from the plywood anymore without seriously damaging them. This is a one way exercise, when mounting the breadboards also take care that the black power rail goes up. The big advantage of this construction however is that wires between breadboards cannot move it is a very rigid construction. Additionally you can easily store the thing somewhere. Save from cats, dogs children etc.

And yes, on my build I have a small misalignment on the bottom breadboards and my top right breadboard is mounted the wrong way around.

- Power distribution

I doubled up the power wiring, see below picture. This became necessary since the amount of IC's for the SAP-3 is far more than what is required for SAP-1. Without this the IC's most distant away from the feeding point of the board had a Vcc of 4.2 Volt. With the doubled up wiring Vcc went up to 4.8 to 4.9 Volt.

Board with doubled up power wiring



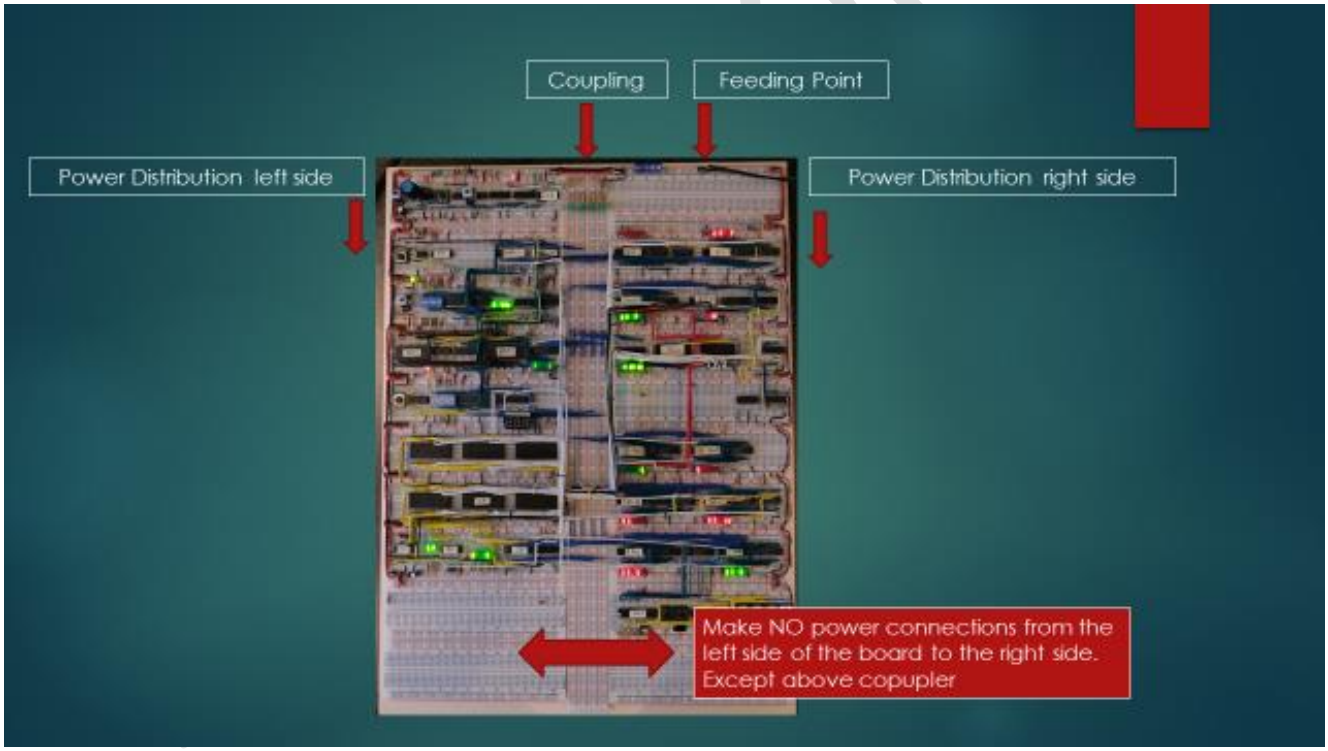
Every power rail has a 0.1 uF capacitor. On the middle of the board left and right I place two 1000 uF capacitors and on the feeding point a 220 uF capacitor.

This is how I made the power distribution over the board. Power is fed from one point and distributed over two strings, one for the boards on the left and one for boards on the right side. One shall not connect the left side of the board with the right side, except the shown coupler. Reason for this is as follows. By making a link on the bottom breadboards between the left and the right power rail a run around loop is created for spurious signals.

Now a lot of posts can be found that power is the most important thing in building SAP computers, I have I note on that remark. Now I agree that decoupling capacitors on each power rail is a good idea as well as some larger capacitors spread over the board. That is even standard practice and advised by the manufacturers of these IC's. And I supply 5.1 volt at the feeding point of the board in order to have 4.8 to 4.9 volt on the IC's on the bottom of the backplane. So I also try to stick to the rules.

But I did some testing with the circuits whereby I gradually reduced the power supply voltage, many IC's kept on working as low as 3.0 volt, some even could go lower. The LED's went blank but on turning the power up again the IC's were still in the same state. So true, power is important but the manufacturers have done a pretty good job in making IC's which are not that sensitive to drops in the power supply voltage. And yes that is not advised according the datasheet. But who uses these IC's at their maximum possible switching range. I think with these breadboard computers nobody.

Power distribution



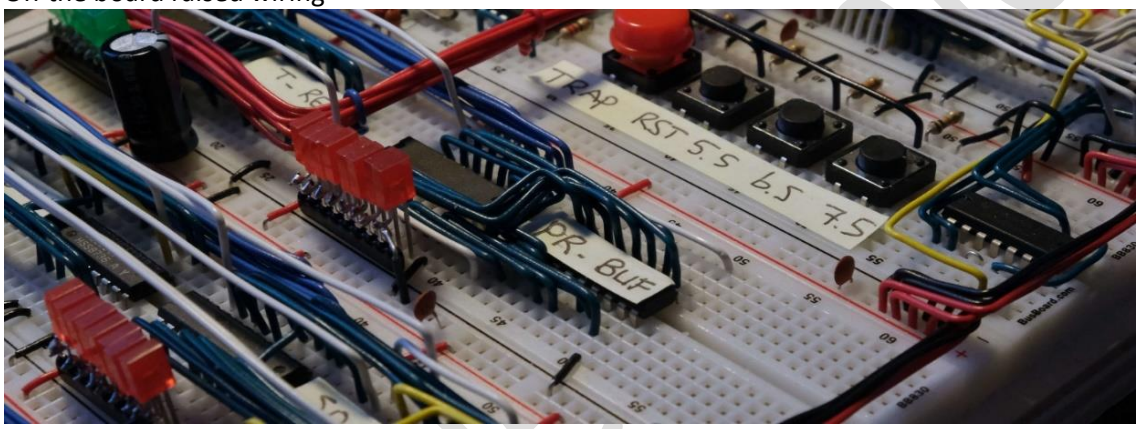
- Wiring Technique.

There are many examples of how to make descent and clean wiring on the breadboard. A good example is given at:

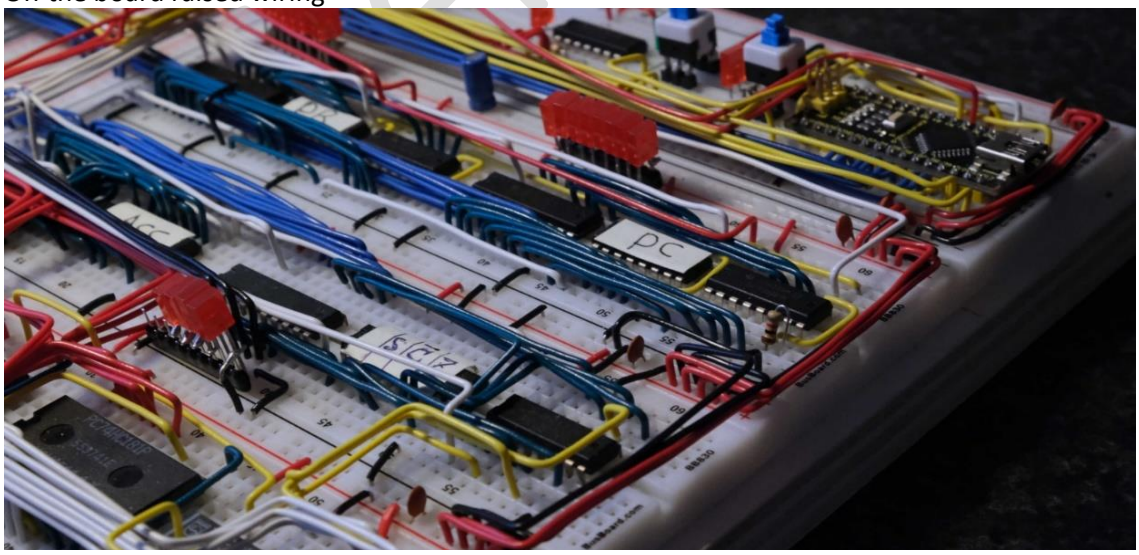
https://www.reddit.com/r/beneater/comments/g9c2do/how_to_make_your_build_clean/

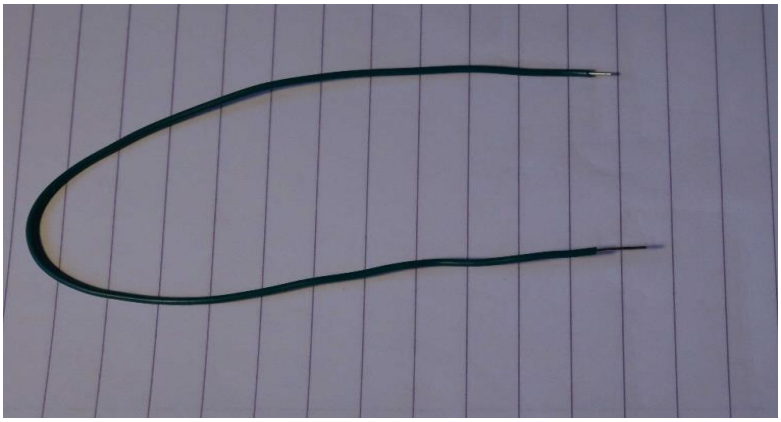
However when changing from a 4 bit addresses to essentially a 16 bit address mode a more compact wiring design is a must. Also I will put two registers on one breadboard, so a more compact wiring technique is required. Due to all this one has to leave the path of flat wiring, it just is too many wires which all need their place. Raising the wires off the board is a possible solution. Below shown are two example pictures of, off the board wiring. But whatever you do neat proper wiring pays off when you need to troubleshoot. Also a good choice of wire colours can help not making mistakes or make troubleshooting easier.

Off the board raised wiring



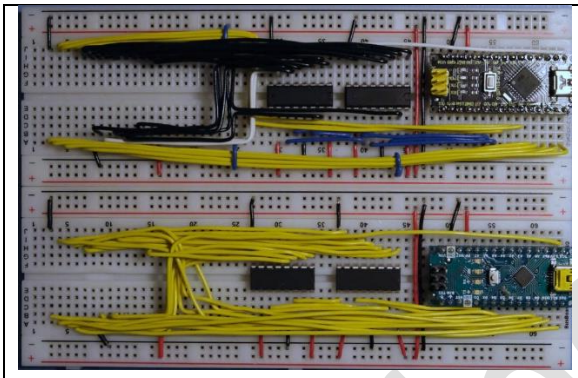
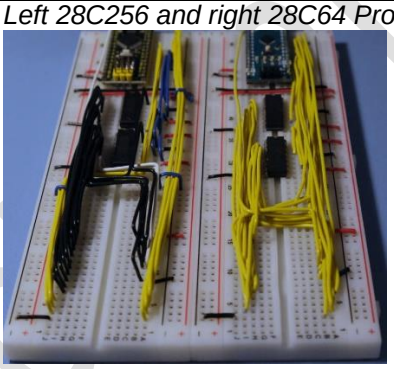
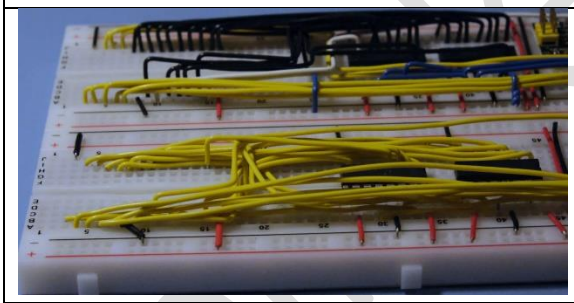
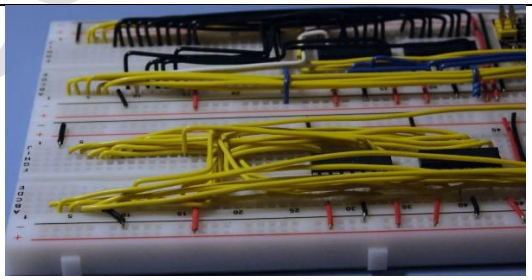
Off the board raised wiring



	Also don't strip off a too short part of the wire insulation. Note the difference of the stripped wire between above and below wire end on below picture. A longer stripped off wire makes a more secure connection.

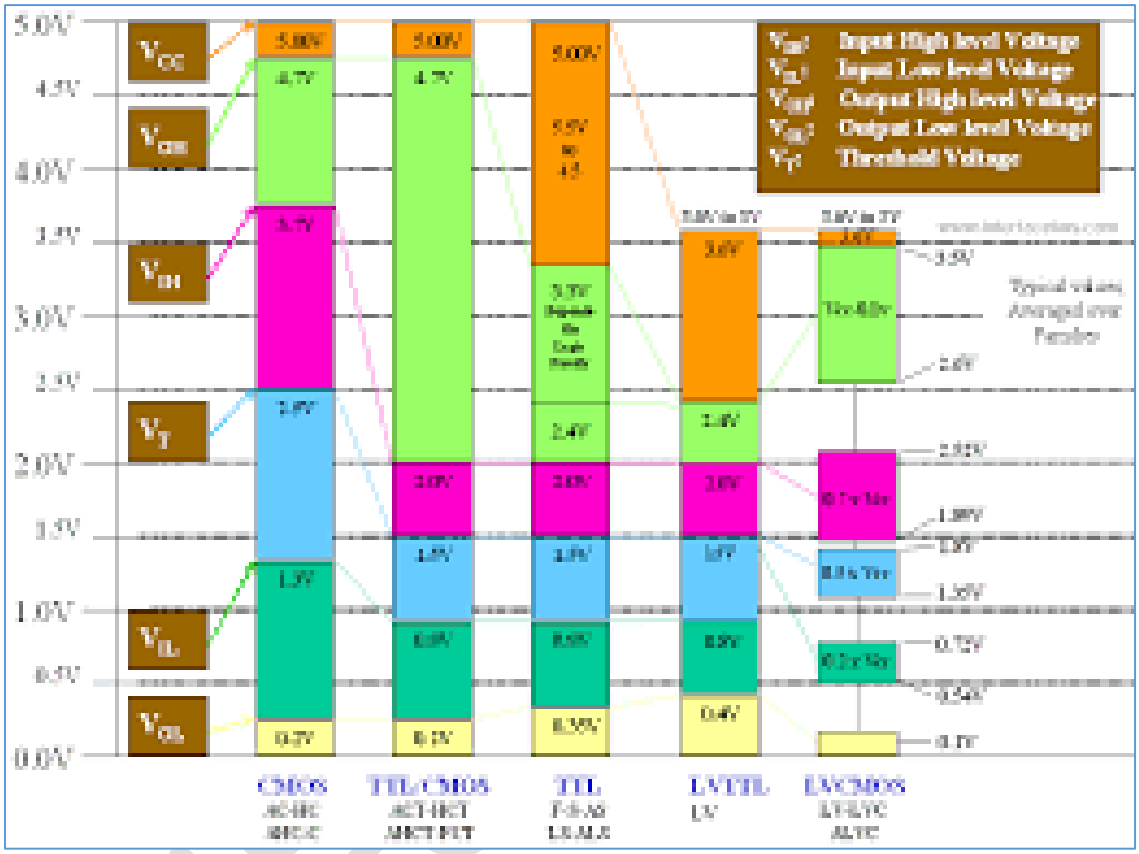
(place better picture)

The advantage of, off the board wiring. Much cleaner, shown are both my EEPROM programmers.

	*Left 28C256 and right 28C64 Programmer* 
	

- TTL Logic Family

The SAP-1 computer from Ben Eater is built with 74LS logic, I built the kit from Ben Eater using these 74LS logic IC'S. Since the amount of IC's would be roughly doubled for the SAP-3, I wanted to reduce power consumption. Initially I changed to 74HCT logic, which wasn't a good idea, but I had many on stock. The better choice for new builds is to go for 74HC logic, for the following reason.



The left most column is typical for 74HC, the column next to it for 74HCT. The 74HCT family has a very high noise tolerance for deviations from high level voltage states. Note that Vt is typically the voltage at which the state is changed. 74HC has a better balance for high and low level noise tolerance and is therefore the better choice.

After having the SAP-2 completely built I could not get it to operate stable, spurious signals made it very unreliable, the cause of this and the solutions are described in Chapter 10, Troubles with my SAP-2 design and Solutions Beside the solutions as described in chapter 10, I also changed from 74HCT to 74HC logic. After all these modifications the computer worked like a charm.

A clearer picture than the one above can be found by googling: TTL logic levels.

- Variable machine cycle, Ring counter versus Step Counter

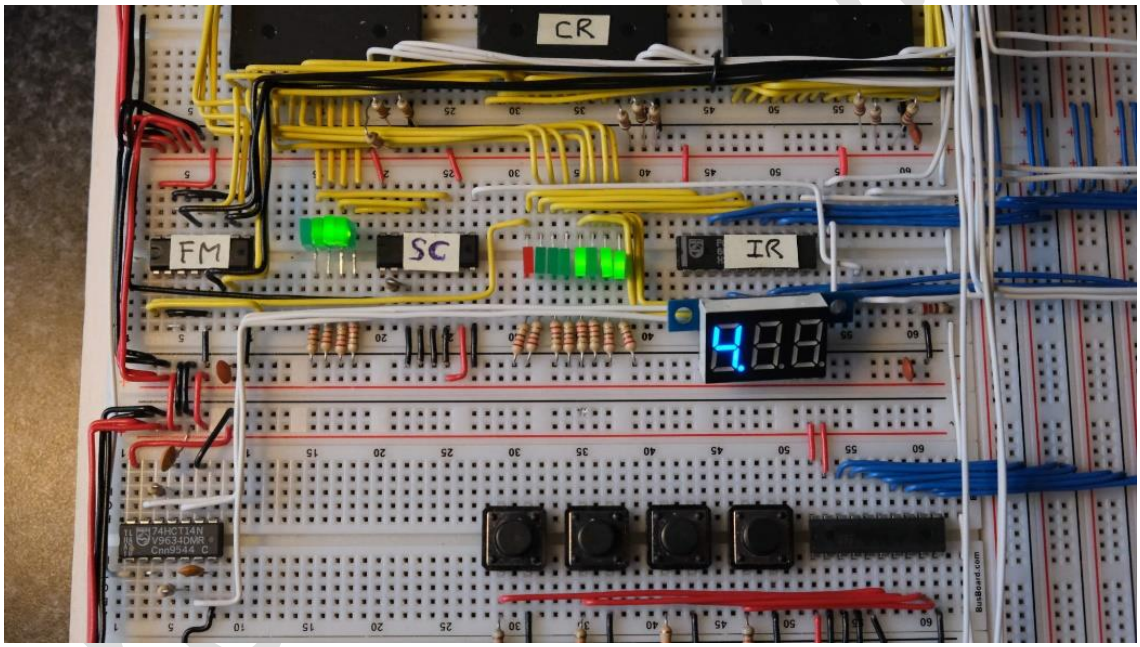
The original SAP-1 design uses a ring counter for the machine states and logic gates for the control logic. A ring counter is different from a step counter that on a 4 bit ring counter only 1 bit is high at the time.

So a ring counter counts: 0001 0010 0100 1000

A step counter counts: 0001 0010 0011 0100 etc

A step counter is more efficient since you can have more steps with the same amount of bits. The advantage of a ring counter is that the signal can directly be used to control the computer. With a step counter a decoding step is required to show in which state the computer is in. Ben Eater in his design built in the 74LS138 demultiplexer to indicate in which state the computer is. Ben Eater built the computer with 5 states, T0 to T4. This can be expanded to T7, by altering the reset which is connected to pin 10 of the 74LS138 to pin 7.

When the control logic is not made from logic gates as described in the book by Malvino but instead an EEPROM is used it's more logical to use a step counter since these can directly be connected to the address lines of the EEPROM.



Picture of the step counter I built. The IC is a 74HC161. I did not build the decoder to reduce IC's so my steps count 0000 0001 0010 0011 0100 etc., indicated by the 4 green LED's. Picture indicates T-state = T7

Additionally I included a step counter reset, this has the following advantage. Most instructions require between 6 and 8 T states, the big exception is the CALL statement which uses 15 T states. Not implementing a step counter reset would mean that almost every instruction would carry out on average 8 no operation steps. This is very ineffective and slows down the computer considerably.

- Page Register and Program Counter

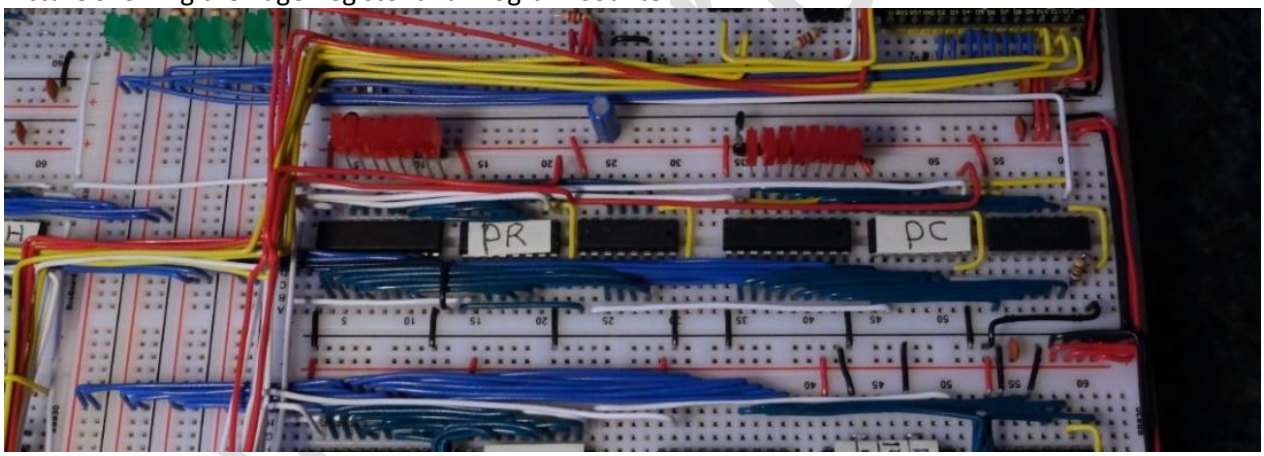
The SAP-1 and the computer from Ben use a 4 bit program counter allowing 16 bytes to be addressed. Although nice for educational purposes 16 bytes severely limits the possibilities of the machine.

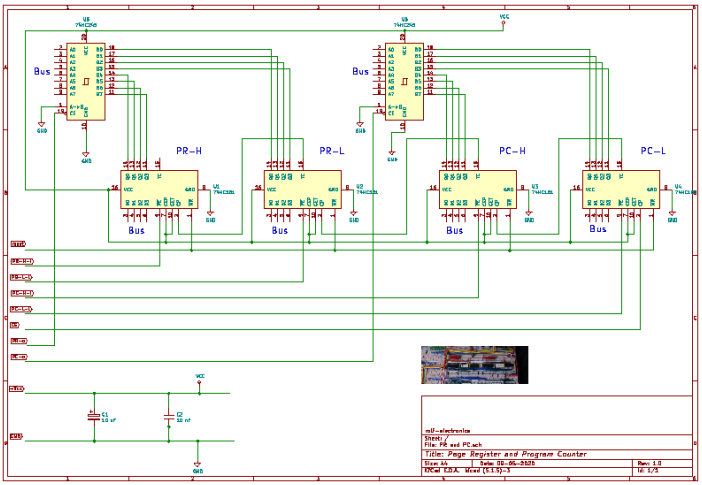
SAP-2 and SAP-3 use a 16 bit program counter allowing 64K (65.536) bytes to be addressed

In my design I use a PR and PC standing for Page Register and Program Counter. Each page has 256 program counter steps and in total there are 256 pages. This is called a paging or a segment register. The program counter indicates an address relative to the page as indicated by the page register. Both the PR and PC are 8 bits wide.

Now this sounds the same as having 1 program counter with 16 bits opposed to two counters with 8 bits, but there is a difference. Since I have an 8 bit bus I cannot send all 16 bits at once. So when addressing memory I cannot send all 16 bits at once, first the PR must be sent and in the next state the PC value. So first the page is addressed and then the relative value from the program counter. This gives the possibility of relative addressing, meaning when I stay on the same page there is no reason to send the PR to the MAR. I have not implemented this yet, but it would double up the speed for instruction requiring access to the memory. For the PR and PC 74HC161 IC's are used.

Picture showing the Page Register and Program Counter





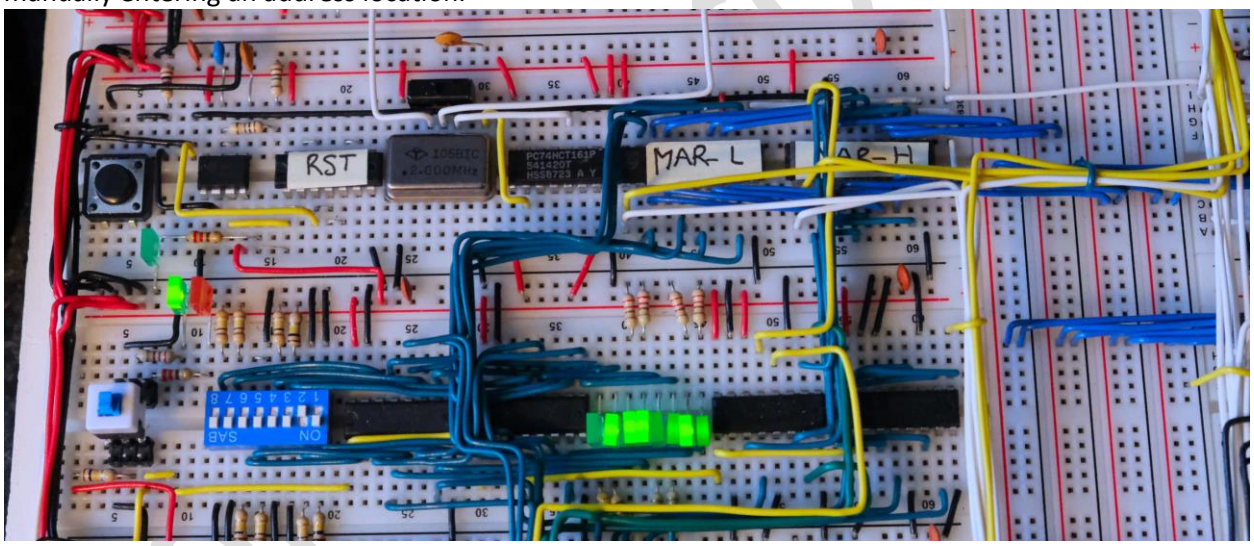
- 16 bit memory address register, MAR

I built a 16 bit MAR with a split in MAR-H and MAR-L. MAR-H being the higher byte of the address and MAR-L the lower byte. Thus PR addresses MAR-H and PC addresses MAR-L. Both MAR-H and MAR-L use a 74HC377 IC. On below picture the MAR-H and MAR-L are shown.

Furthermore there are DIP-switches for manually entering an address in program mode. These are only foreseen for the MAR-L address. My address mapping is such that the RAM section starts from address 0x8000. The address multiplexers are wired in such a manner that only for page 0x80 addresses can be entered manually. I found it totally irrelevant to make also DIP switches for MAR-H, I never had any intention to toggle in that many program steps. Entering one page, 256 bytes, with DIP switches is really sufficient. The Green LED's show the MAR-L status. The pushbutton on the right is the PROG/RUN switch with the two LED's, red for program mode and green for run mode.

Since the addition of the Arduino for programming the RAM the PROG/RUN switch and the DIP switches for manually entering a program are hardly used anymore. One could decide to skip the manual programming mode completely saving space on the breadboards, IC's and wiring, in my case the complete lower breadboard of below picture.

Picture on the top right shows the Memory Address Register, MAR, and the DIP switches for manually entering an address location.



- 8 bit W-bus

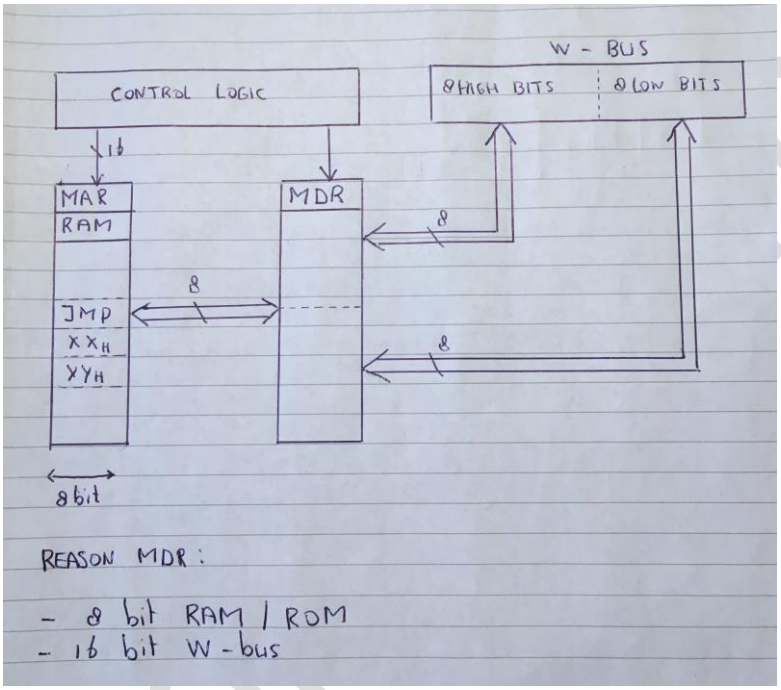
When I started SAP-1 very quickly I made a plywood board to mount all the breadboards. I anticipated on more breadboards for SAP-2 and SAP-3, but only when I started to look into it in detail I found that in SAP-2 and 3 the bus is 16 bit. As stated earlier once mounted on the backplane the breadboards cannot be removed anymore without seriously damaging them. So I had an 8 bit bus, as can be seen on previous picture. Ok throw away the board and start over again.

Hell NO, I am Dutch and I'm not going to through away 20 breadboards !

And why should you have a 16 bit bus, all registers, the memory and ALU are 8 bits. Only the PC and MAR are 16 bit. And in using relative addressing there would hardly be any difference.

For the rest the bus is standard and made exactly as per Ben Eater design.

- No Memory data register, MDR



Sketch of the MDR, as explained in the book from Malvino.

Data flow

With an 8 bit wide RAM/ROM only 8 bits can be stored and that's also the format of all registers. So by using the lower 8 bits of the bus data can be transferred. Only if the registers and ALU were 16 bits an MDR would be required to make the translation from 16 to 8 bit. But making registers and memory 16 bit is no longer an 8 bit computer.

Address data

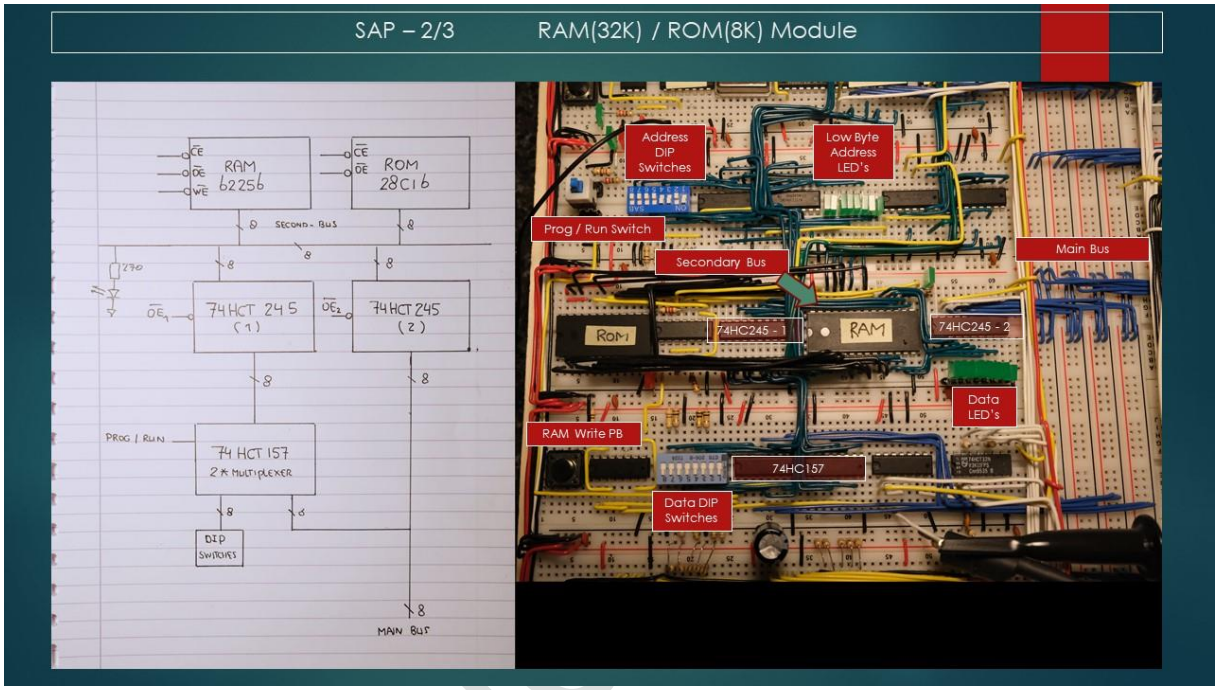
For a JMP instruction with 2 bytes for the address an 8 bit RAM/ROM and a 16 bit bus and PC which can only be loaded in one time with the complete address also an MDR would be required. But since I have the PR and PC separate both for register out and register load signals, an MDR is not required in my design.

- ROM and RAM memory

My memory address is mapped as follows:

- 0x0000 to 0x7FFF, is for ROM,
- 0x8000 to 0XFFFF, is for RAM.

The memory data is 8 bit wide. With IC's which were readily available, I build 8K ROM and 32K RAM with respectively an 28C64 and a 62256.



Lay-out and principle sketch of my memory section

Also here a DIP switch can be seen for manually entering data, once I built the Arduino interface to the memory this has hardly been used anymore.

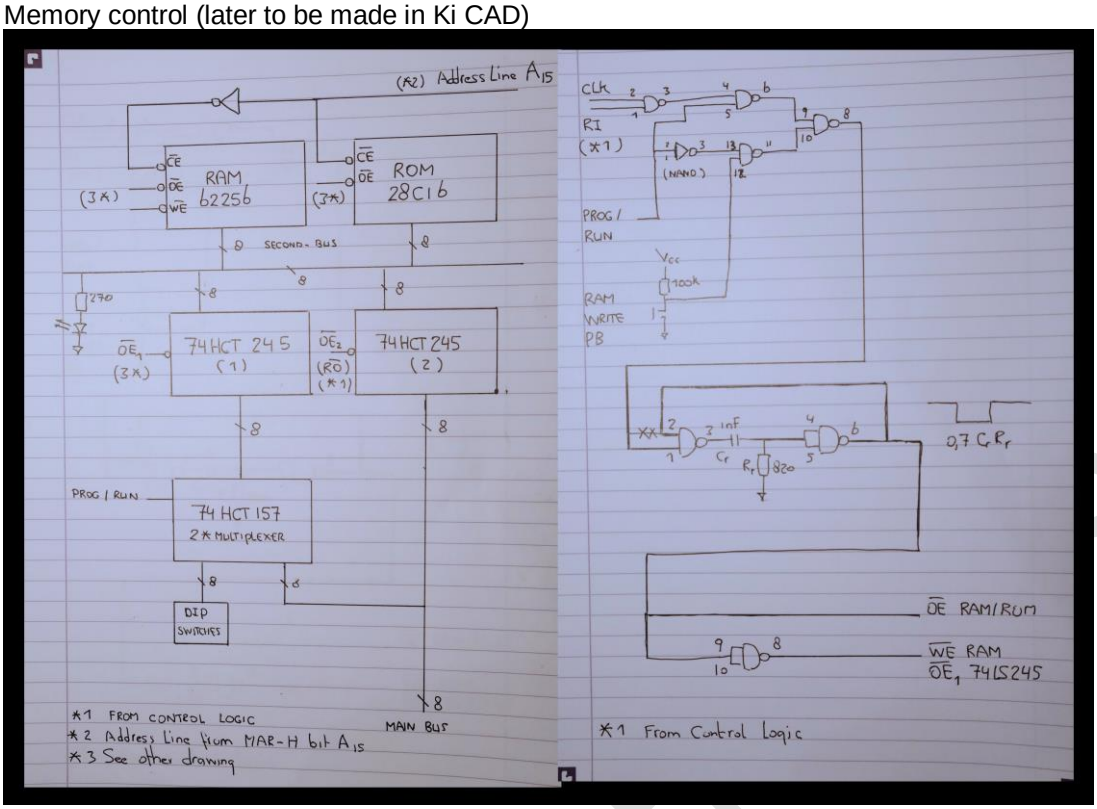
Unlike the 74LS189 as used by Ben Eater the 62256 has common I/O. If the data LED's should indicate the data output for the selected address, than the 62256 must have its output enabled. This however would conflict with the data signal from the multiplexer. Leading to a situation were 2 outputs drive a common line, which is an unwanted situation. This is the reason for the extra created bus, indicated on the drawings as "second bus". A 74HC245 separates the output from the 65256 from the output from the multiplexer.

If data needs to be written into the RAM, either manually or by means of RI, the following happens:

- the OE1 signal is set to low, now the data from the multiplexer is on the second bus,
- the OE of the 65256 is made high, setting the I/O to input,
- the WE of the 65256 is pulsed to store data on the secondary bus in RAM.

The circuit to control this is drawn on the next page.

Memory control (later to be made in Ki CAD)



(there is a mistake in above drawing, OE RAM/ROM and the signals connected to pin 8 of the NAND should be reversed)

- Registers

There are two kind of registers; special purpose and general. Normally special purpose is not available for the programmer and general purpose is available to the programmer. On this some sort of rule there are exceptions like MCU's having access to special purpose registers is quite common.

Special purpose register:

- the 8 bit instruction register,
- the temporary register.

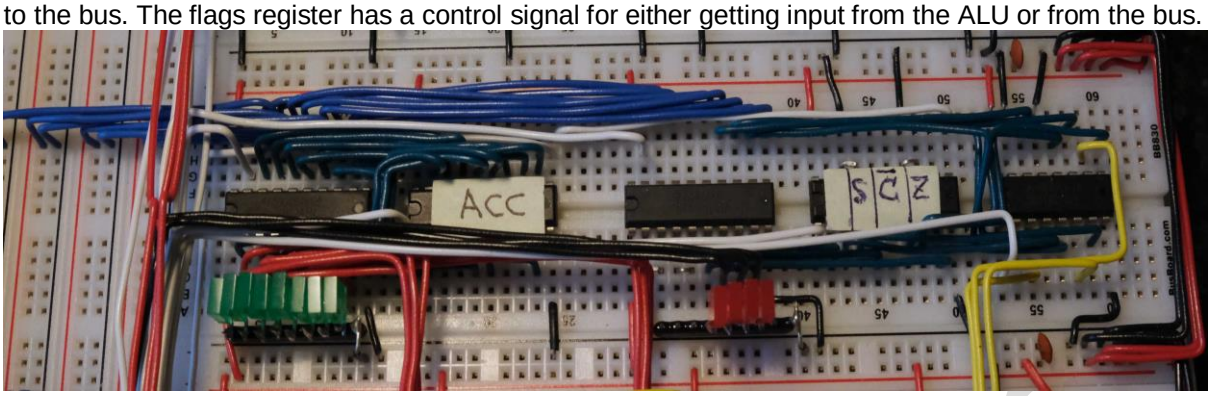
General purpose register

- the flag register,
- the 8 bit A, B and C register.

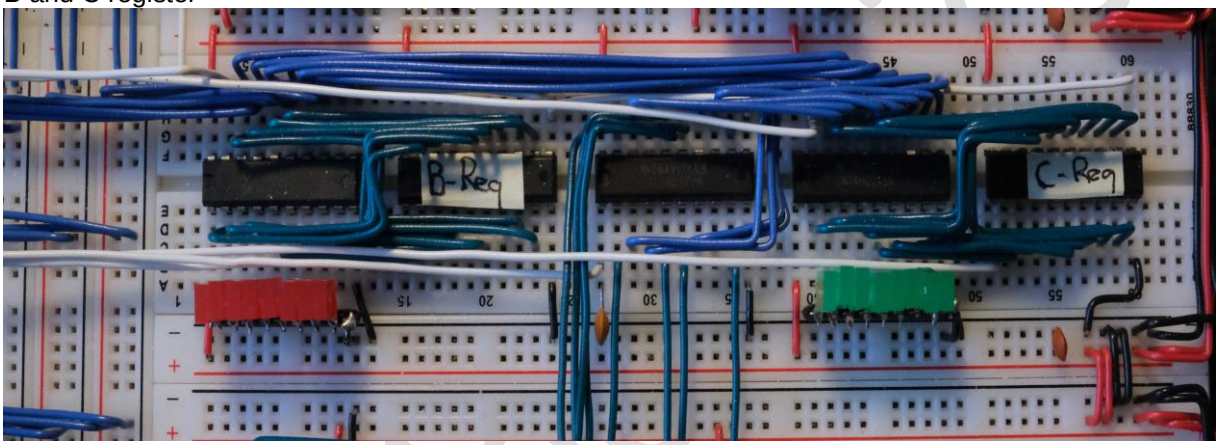
For space consideration the A and flag register are built on one breadboard. Also the B and C register are built on one breadboard. The instruction register is combined with the step counter and flag masking register, which purpose will be explained later.

Additionally the flag register is connected to the bus via a 74HC245, this makes it possible to store the PSW. PSW is the Program Status Word. All register use the 74HC377.

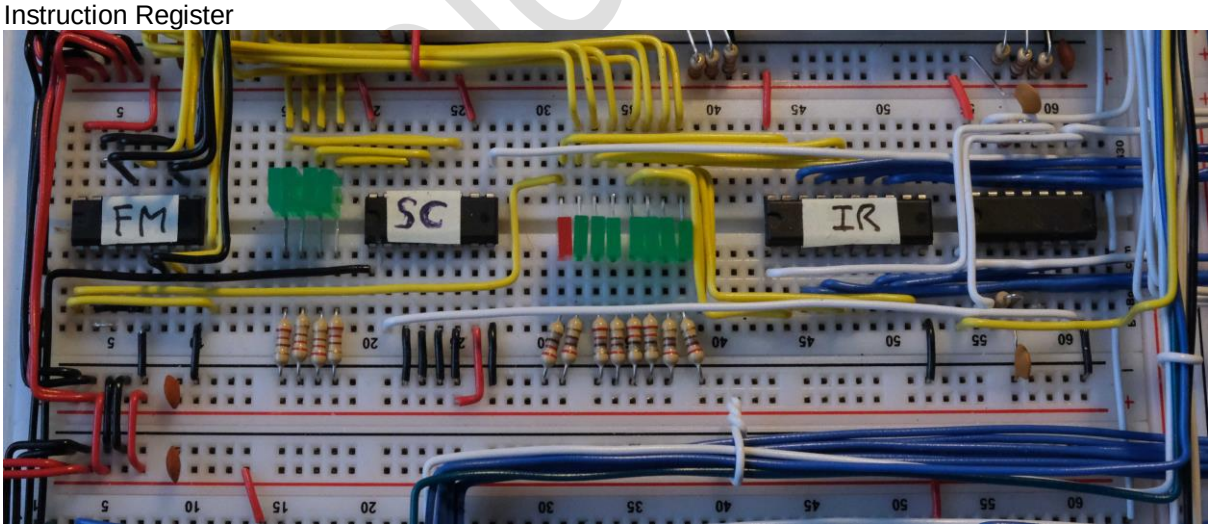
A and flag register. The IC left to the IC marked “S Z C”is the 74LS245 for connecting the flag register to the bus. The flags register has a control signal for either getting input from the ALU or from the bus.



B and C register



Instruction Register



The purpose of the FM IC will be explained later.

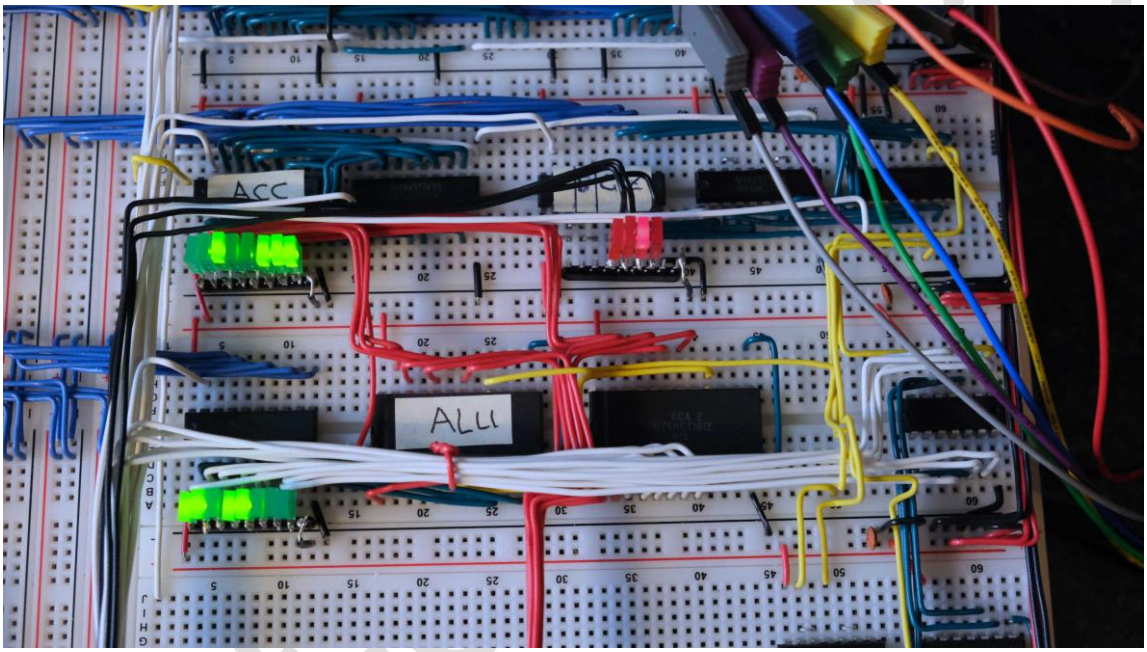
- 8 bit ALU with 74HCT181

For the ALU I chose to use a 74HC181, kind of an obvious choice since there is not much to choose from. Each IC handles 4 bits so you need two, and the carry bit connects the lower 4 bits (nibble) with the upper 4 bits.

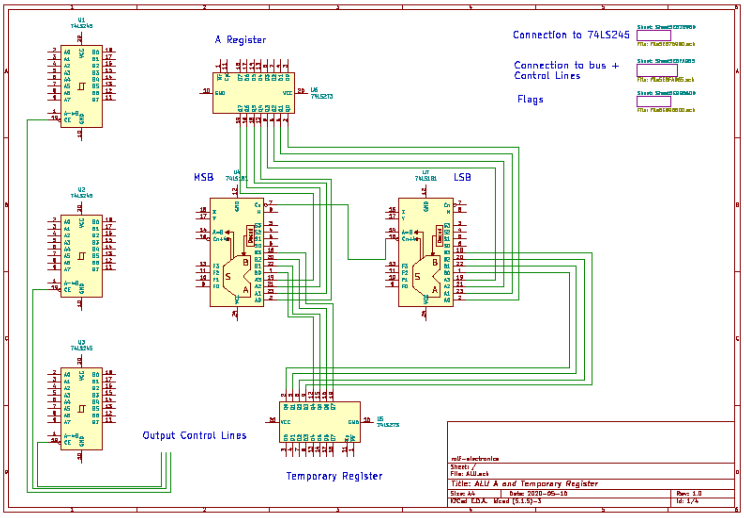
The 74HC181 has 6 control signals:

- 4 operation mode select. Selecting things like addition, subtract, AND, OR etc,
- 1 mode selection. Selecting arithmetic or logic operation,
- A carry in, which you can see as a control signal.

The carry flag to the control EEPROM's comes from the ALU with the higher 4 bits. The sign is the MSB and the zero-flag I made identical as done in the built from Ben Eater with some additional logic circuits.



Picture of the ALU, the green LED's on the bottom left are the ALU output



Actual drawing is 4 pages, see KiCAD sheet reference.

- Two strange beasts, number 1: Flag masking

On the page with the instruction register or the picture were I explained the built on the step counter there is an IC labelled FM. Now this is not an FM radio, it stands for flag masking. For what purpose did I built this.

One of the charms of the 8 bit computer is that everything fits nicely on a breadboard. And Ben Eater supplied a program suitable to program an array of EEPROM's 28C16/64/256. So my plan was to stay with that program and use an Arduino Nano.

Now as you might have noticed in one of the video's Ben gets a warning, saying something like

"Low memory, stability errors might occur". Which is computer language for "I cannot do this". Ben's solution was to use the program area of the Arduino Nano by means of a different array definition being: const uint8_t array_name PROGMEM. Problem solved.

Now Ben had two flag, 8 T states, 16 instructions and a 16 bit control word. And for n flags he need 2^n copies of the database.

In order to build SAP-3 I needed 4 flags, 16 T states, basically 256 instructions since the instruction register is now 8 bits and with 6 control EEPROM's I would have 48 bits on the control word.

And then it all goes wrong. The array I had to build to program the EEPROM became way too big, could not be handled by an Arduino Nano, also switching to an Arduino UNO or Mega did not help. They all could not handle the extreme large array. So I had to think out a solution, which I called FM.

Now it gets complicated to explain, and that's why we engineers use DRAWINGS. But I will give it a try.

I route the flags first through a multiplexer, a 74HC157. The select line of the multiplexer is connected to the MSB of the instruction register. The A inputs of the 74HC157 are connected to ground the B inputs to the flags. The outputs go to the EEPROM address lines. The result is the following.

My first 127 instructions were the MSB=0 select S=0 (the A inputs) and as such the EEPROM address lines are zero always by definition. The first 127 instructions are defined as non-flag related. So in my design for the non-flag related instructions the flag lines to the EEPROM are zero always.

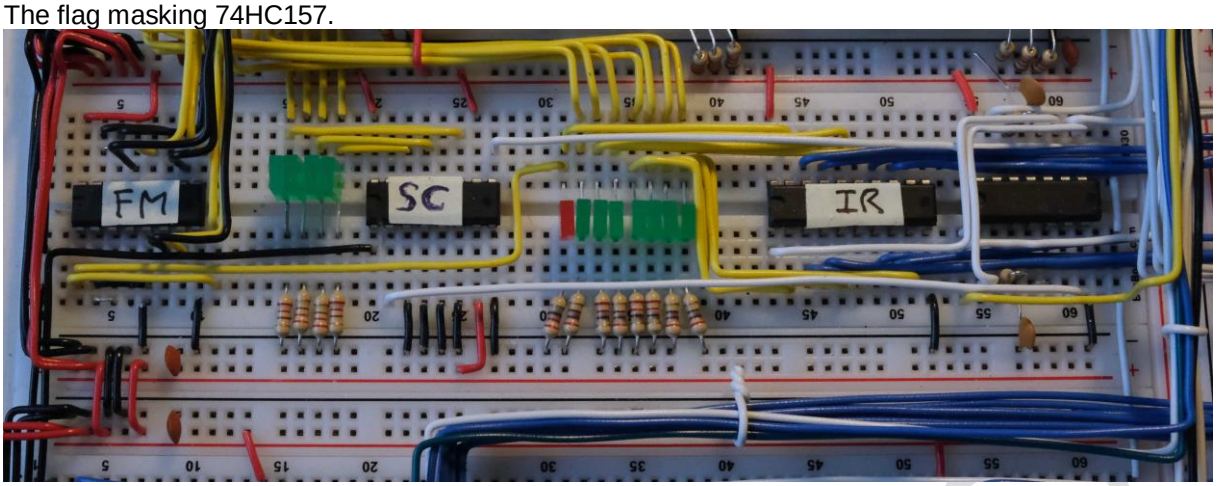
This avoids me making multiple copies of the data array as Ben does, this helped a lot.

My flag related instructions are from 128 upwards, these are a handful and for them I have to make multiple copies, but that's much easier to handle.

Additionally I skipped the idea of swappable EEPROM, before I program an EEPROM I have to explicitly set which one. So each step in the microinstructions is only 8 bits. And with that it fits nicely in an Arduino Nano.

Hopefully my writing is clear, it's always better to use drawings since that's the engineering language.

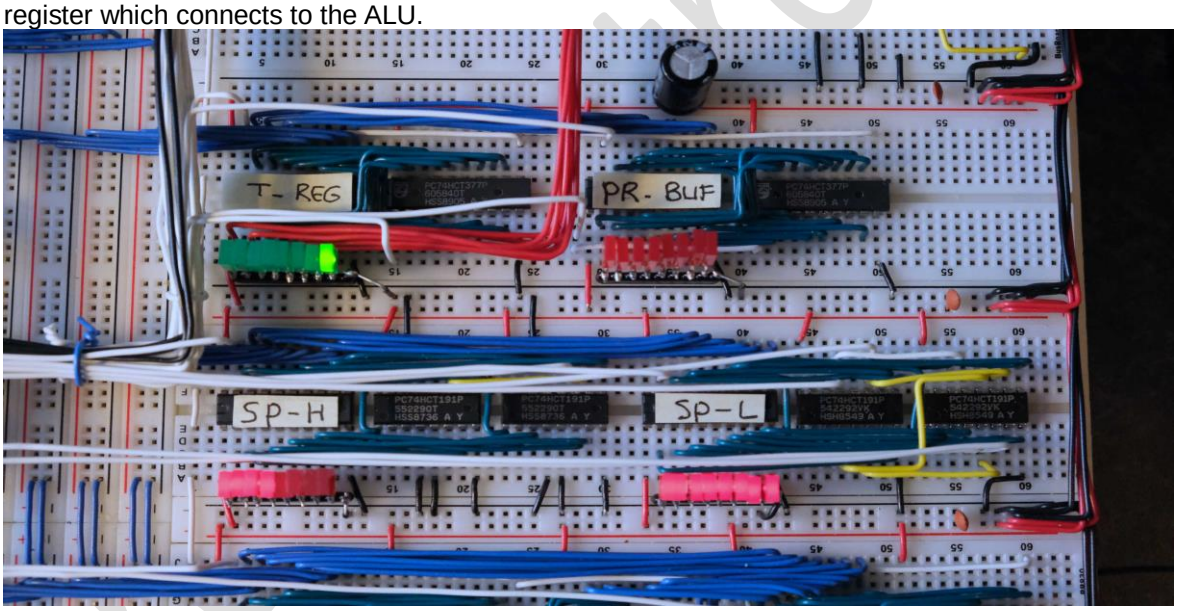
The flag masking 74HC157.



Note the LED's in-between the step-counter and instruction register. This is the current instruction in the IR. The red LED is the MSB, when it lights up it's a flag related instruction and vice-versa.

- Two strange beasts, number 2: PR buffer

The PR buffer is just another register build from an 74HC377. I installed it next to the temporary register which connects to the ALU.



Each instruction starts with fetching the next instruction which is pointed to by the PR and PC. Per instruction I need to increase the program counter with 3, since each instruction is three bytes.

At T0 PR goes out to the bus, and gets loaded into MAR-H.
At T1 PC goes out to the bus, and gets loaded into MAR-L, MAR now points at the next instruction
At T2 the instruction is outputted from RAM or ROM and loaded into the instruction register.

The instruction register is now loaded with the next instruction, this is always the same for each and every instruction. Except one which has to do with the interrupt controller.

Below as example are the micro instructions for the LDA command.

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
ins_0,	ins_1,	ins_2,	pco&mal,	ro&prbi&CE,	pco&mal,	ro&mal,	prbo&mah,	ro&ai&CE,	scr,

ins_0 = pro&mah

ins_1 = pco&mal

ins_2 = ro&ii&CE

T0, T1 and T2 are for fetching the instruction. I defined the microinstructions in the array with ins_0, ins_1 and ins_2 for simplicity. Now a typical LDA instruction would have three bytes stored as follows:

- the instruction
- the page register to jump to
- the pc value to jump to.

The purpose for the PR buffer register becomes clear now. Once the instruction is loaded at T2 the program counter is increased and now points to the PR from the LDA instruction. In T4 the program counter gets out to the bus and is loaded into the MAR-L.

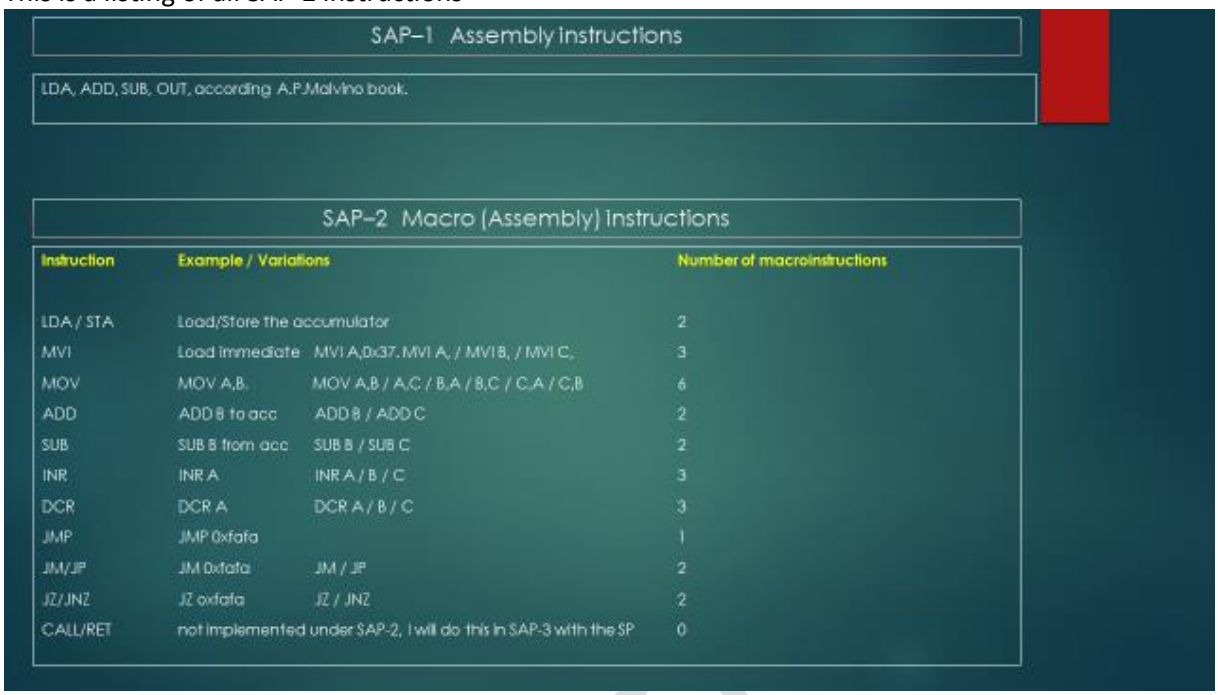
But you cannot say now RO&PR-load, meaning you want to load the PR of the LDA instruction into PR, if you would do that than the PR is suddenly pointing to a completely new instruction on another page which has nothing to do with the next step in the original instruction getting the PC value. And that's the function of the PR-buffer.

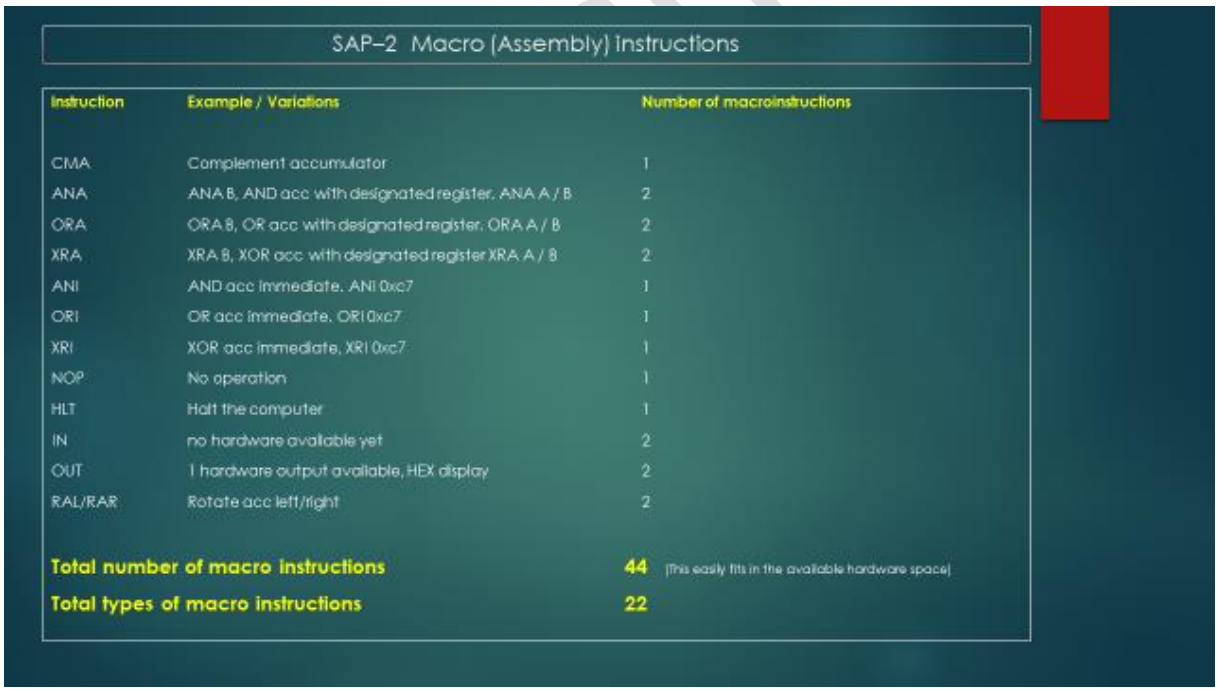
Therefore:

- program counter is increased pointing to PR from the LDA instruction
- PR from LDA instruction is outputted from RAM/ROM to the bus and stored in the PR-buffer
- program counter is increased pointing to PC value from the LDA instruction
- PC is outputted from RAM/ROM to the bus and stored in MAR-L
- PR-buffer is outputted to the bus and stored in the MAR-H
- MAR points now to the address which needs to be loaded in the A register
- RAM/ROM value is outputted to the bus and loaded in the A register.

- 44 Instructions

This is a listing of all SAP-2 instructions





- The CALL instruction 18 T states in SAP-2

To start with I am not going to explain why you need CALL instructions and how they work together with a stack, this is all nicely explained by Ben Eater in the follow video.

https://www.youtube.com/watch?v=xBjQVxVxOxc (30 min video)

Now most instruction from SAP-2 are pretty straight forward and can easily be derived from SAP-1, this however is not the case with the CALL and RET instruction. The CALL instruction is also the reason why you need a 5 bit step counter in SAP-2, since you need 18T states to execute this instruction.

Next are three power point slides trying to explain how I think the CALL instruction works in SAP-2, this is not explained in detail in neither the book from Malvino and also not in the book from D.K.Kaushik. This is what I think, I might be right since Malvino does state that it takes 18T states, which I also count. For executing the CALL command in SAP-2 special hardware must be built to load the MAR-H and MAR-L.

Like I said earlier I decided not to build this and implement the CALL instruction as explained for SAP-3 using a stack pointer, this will be explained later under SAP-3 extensions. In SAP-3 when using the stack pointer for the CALL instruction this can be made with 15T states, which fits nicely in a 4 bit counter. 18T states as per SAP-2 would require a 5 bit counter and also would require an extra address line to the control EEPROM's, and that for one instruction. Additionally on SAP-2 nesting subroutines is only 1 level. On SAP-3 you can have many more.

But the basic idea under SAP-2 is as follows:

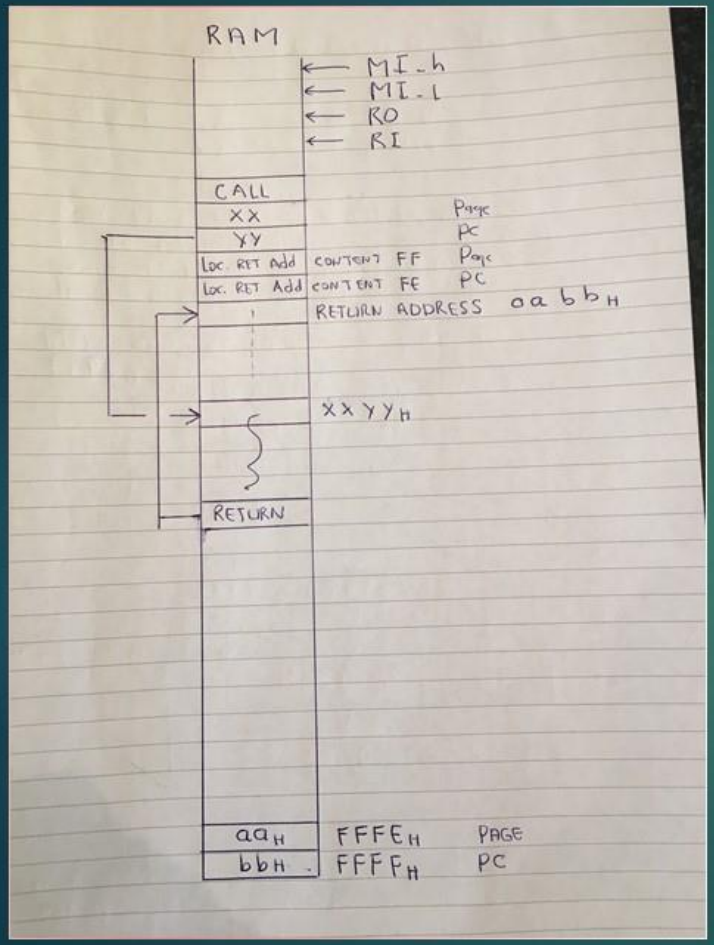
- The A and temporary register are used to store to address to be jumped to as defined in the CALL instruction, in my example xx and YY.
- When that is done the PC points to the return address
- Next the return address is saved at memory location 0xffff and 0xfffe,
- Only this location is foreseen for the CALL instruction, so you can have 1 level of nested CALL, A CALL function in which another CALL is used will not work (*)
- After the return address is saved the values in the A and temporary register are loaded into the PC.

The return function works identical, only then the values stored in memory location 0xffff and 0xfffe are loaded into the PC.

The next pages provide this in detail with as last the setup of the microinstructions as I think it is supposed to work.

10 LDA, 5	100 CALL 200 (this is not possible in SAP-2)
20 CALL 100	110
30 etc	120

Ring Counter Design for the CALL instruction in SAP-2



CALL INSTRUCTION SAP - 2

Step 0 fetch Instruction

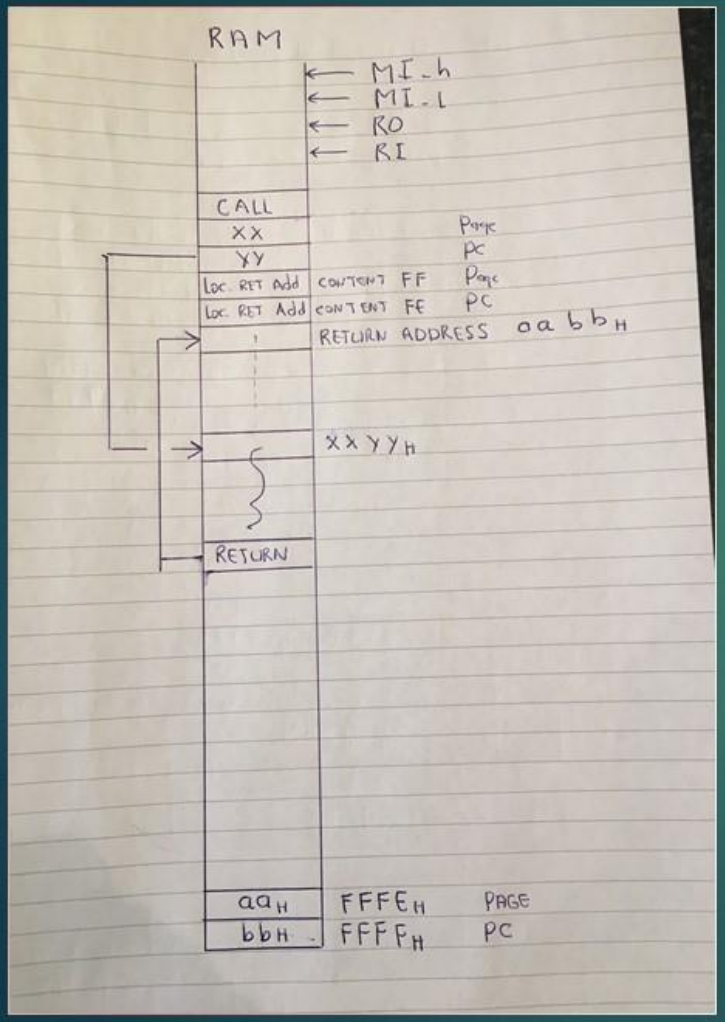
Step 1 XX → A-register
YY → Temp-register

Now PR + PC (OR 16 bit PC) points to the return address ⇒ cannot change anymore before they are stored otherwise the return address is lost

MAR must be updated to point to $FFEE_H$ → How to do this without using PR + PC

This solution adds a lot of hardware and more wiring. Secondly not used in SAP-3 ⇒ SP

Ring Counter Design and Micro Instructions for the CALL instruction in SAP-2



Step 1 Fetch Instruction		3T
Step 2 xx → A-register		4T
yy → Temp-register		
Step 3 FE → B-register		4T
FF → C-Register	PR + PC now indicate the return address (PR + PC cannot be changed anymore)	
Step 4 C-register_out → MI-h B-register_out → MI-l	MAR now points to FFFE[h]	2T
Step 5 PR_out → RAM-in		1T
Step 6 C-register_out → MI-h C-register_out → MI-l	(optimization possible, can be taken out) MAR now points to FFFF[h]	2T
Step 7 PC_out → RAM-in		1T
Return address now loaded at FFFE and FFFF[h]		
Step 8 A-register_out → PR_in		2T
T_register_out → PC_IN	Jump address loaded in PR + PC	
Total number of T states		**19**
Number of T states counting from T0		**18**

Ring Counter Design and Micro Instructions for the CALL instruction in SAP-2

Step 1	PR_o | MI_h , PC_o | MI_l , RO | II | PC_e	T0 T1 T2	Fetch instruction	
Step 2	PC_o | MI_l , RO | AI | PC_e	T3 T4	xx to A-register	For step 2 and 3 PR does not need to be
	PC_o | MI_l , RO | TI | PC_e	T5 T6	yy to T-register	updated, all required data is stored sequential
Step 3				
	PC_o | MI_l , RO | CI | PC_e	T7 T8	FF in C-register	
	PC_o | MI_l , RO | BI | PC_e	T9 T10	FE in B-register.	After T10 PR and PC indicate the return address
Step 4	CO | MI_h , BO | MI_l	T11 T12		MAR points to FFFE
Step 5	PR_0 | RI	T13		Store PR
Step 6	CO | MI_h , CO | MI_l	T14 T15		MAR points to FFFF
Step 7	PC_0 | RI	T16		Store PC After T16 return address is loaded
Step 8	AO | PR_I , TO | PC_I	T17 T18		Load jump address in PR and PC

Chapter 7, Additional minor Improvements SAP-2

No LED's for the control bits and no inverters, this saves a lot of wiring with 48 control bits

Moving the control Logic to the middle of the board. This will reduce control wiring lengths compared to having them mounted on the bottom of the backplane.

Looking from top left to bottom left the modules are ranked as follows:

- Clock module
- ROM RAM memory
- Control register
- FM / Step counter / Instruction register

In hindsight to reduce the control wiring even further I think the following sequence is even better:

- Clock Module
- Control register
- ROM / RAM memory
- FM / Step counter / Instruction register

I decided not to have reset signals to the registers this would save wiring.

Resetting the registers could be done on start-up via the ROM. There is one sneak, this works perfectly fine for the A, B and C register and also for the HEX display.

However:

When the build was complete and I started testing, the computer was not working correctly. The step counter as well as PR and PC still had a connection to the rest button, the instruction register however not. So at start the IR would show a random instruction, if this instruction is not defined than control signals will be undefined and weird things happen. Having reset the PR and PC does not help if the first things defined in the IR is not: PR|MAR-H and next PC|MAR-L and consequently RO|II.

The solution was to write on the EEPROM for each possible instruction the first above three steps.

Problem solved!

Von Neumann Cycle, first get the instruction. I did not think about this when I deleted the reset signal on the instruction register. And I had to find a solution since an 74HC377 has no reset and I did not wanted to go back to two 74HC161's. This would use more breadboard space.

Better quality of the clock signal

A modification I still want to do is installing HEX Schmitt triggers on the clock signal and separate the clock from the left side of the board from the right side. Purpose of this is to improve the quality of the clock signal. This is one of the things required in order to have the computer run on higher clock speeds.

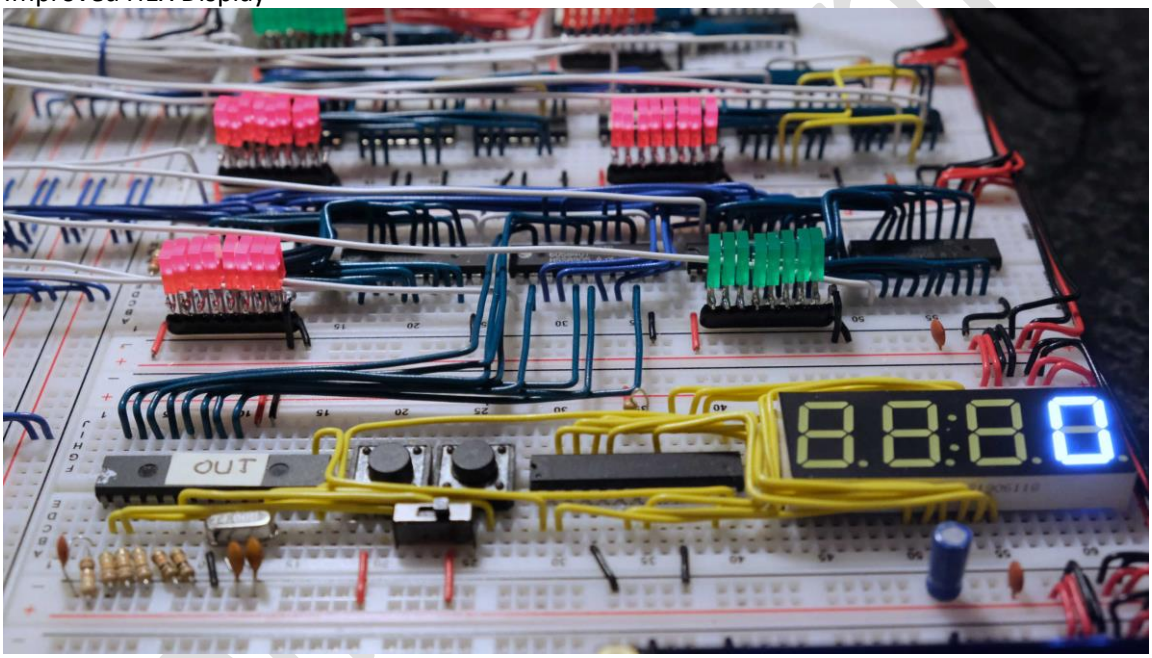
Improve brightness HEX display

The original display as built by Ben Eater is faint it helps placing the red foil on top of it, but still it's not the brightest display. Main reason for that is that an EEPROM is not supposed to drive a 7 segment display. This can very clearly be seen on the data output LED's of my memory module. When RAM is selected the LED's burn a lot brighter than when the ROM (EEPROM) is selected.

I also wanted to build the complete display on one breadboard. I used a MAX 7219CNG in combination with an 16F874 MCU. Beside the switch for one and two complement I also added two pushbuttons for brightness control. It makes a lot of difference driving the LED's with an MAX7219CNG, especially since I also used a new 7 segment display a super bright version. This makes the display very easily readable also during daytime. Initially I had the max current to the 7 segment display adjusted too high and the display was really (really) bright. In doing so I burned a couple of segments on the display. After adjusting this at the MAX7219CNG, which means changing a resistor, it worked like a charm. And obviously I had to buy a new display.

Disadvantage is the use of a microcontroller, making it less easy to reproduce for others. I have an EASYPIC V7 MCU development boards and with that it's a walk in the park.

Improved HEX Display



The IC marked "out"is the 16F874. The IC next to the 7 segment display is the MAX7219CNG.

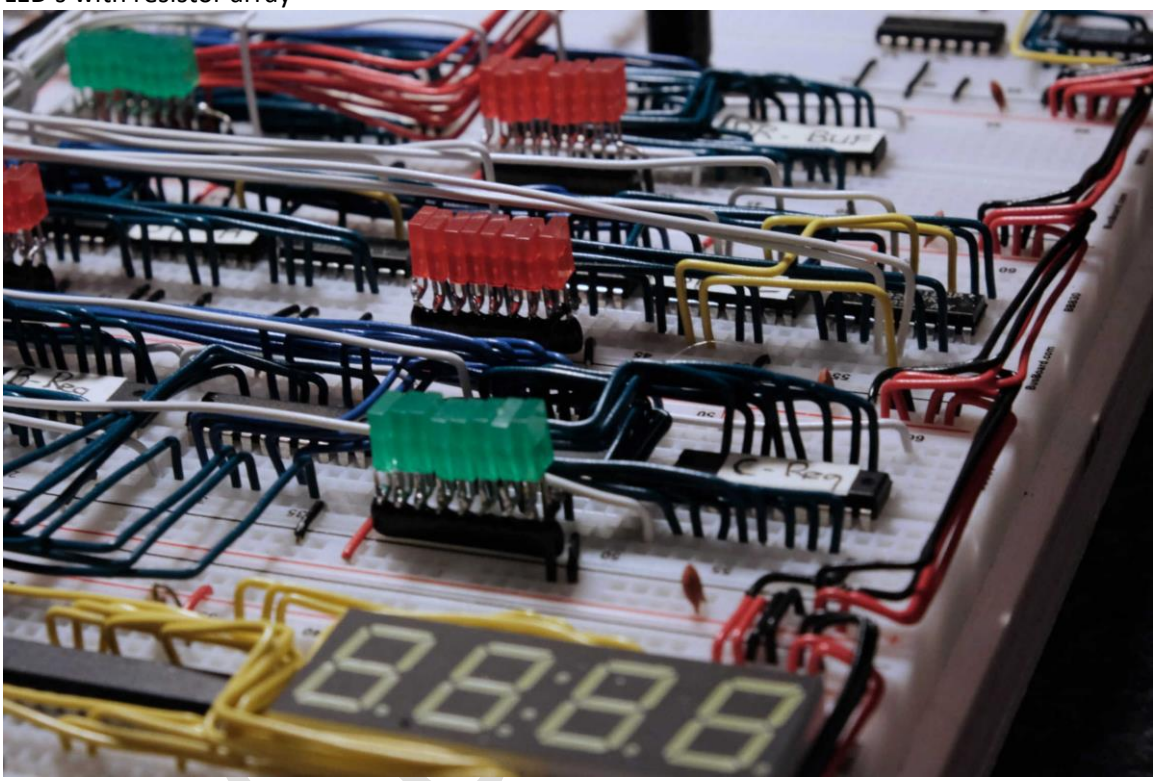
Currently I still have an 74HC377 connects the bus with the 16F874. Reason for that is that I only had a 4MHz crystal on stock, making it too slow at higher clock speeds. In future I might change this, although there is no real good reason. It works very well.

Use of less and a different setup of the LED's

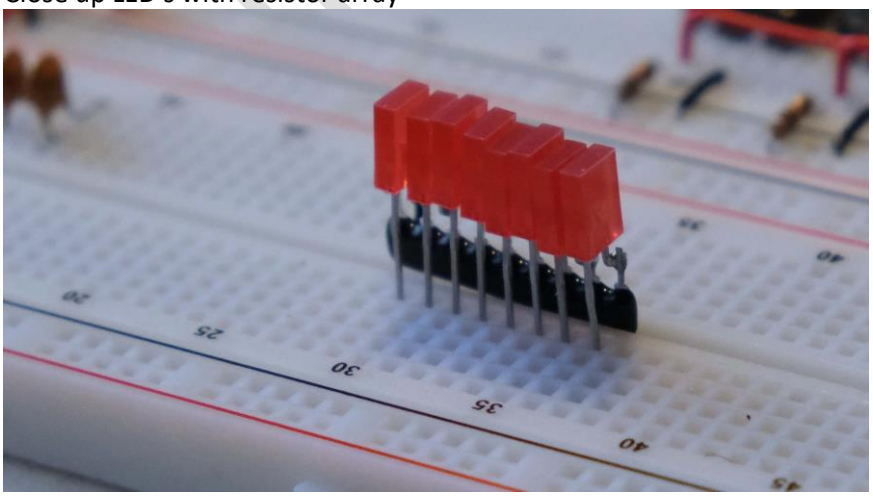
I did several posts on my 8 bit computer build, this one was popular. The original LED's as used by Ben Eater uses the round 8mm LED's with separate wiring. This takes up a lot of space on the breadboard and also more wiring. They also do not nicely line up with the breadboard.

So I took 2*5 mm square LED's, ordered some 8*270 ohm resistor arrays, placed the resistor array upside down and soldered the LED's to the array. Add one additional ground wire and done. And they line up perfectly with the breadboard,

LED's with resistor array



Close up LED's with resistor array



For some reason you see people not using current limiting resistors on LED's, I did four posts on the 8 bit Reddit subchannel on why you should install them.

https://www.reddit.com/r/beneater/comments/gcojww/led_directly_connected_to_an_output_i_hooked_up_a/

https://www.reddit.com/r/beneater/comments/gct4bk/more_experimenting_with_leds_someone_asked_whats/

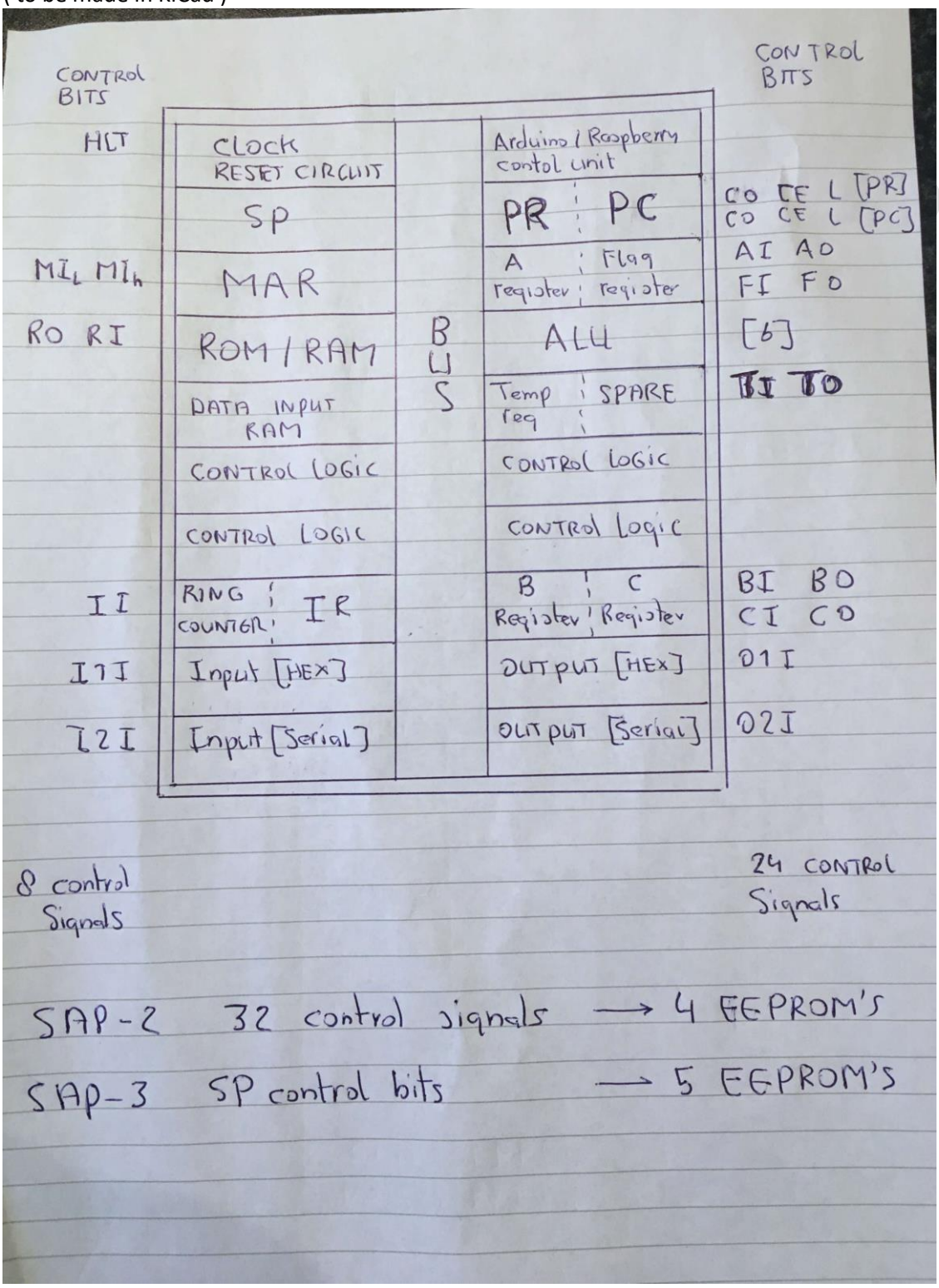
https://www.reddit.com/r/beneater/comments/gcw42q/last_one_with_leds_for_today_if_i_could_do_the/

https://www.reddit.com/r/beneater/comments/gcxgpu/one_more_and_than_i_am_done_with_these_leds_i/

Chapter 8, Starting to build

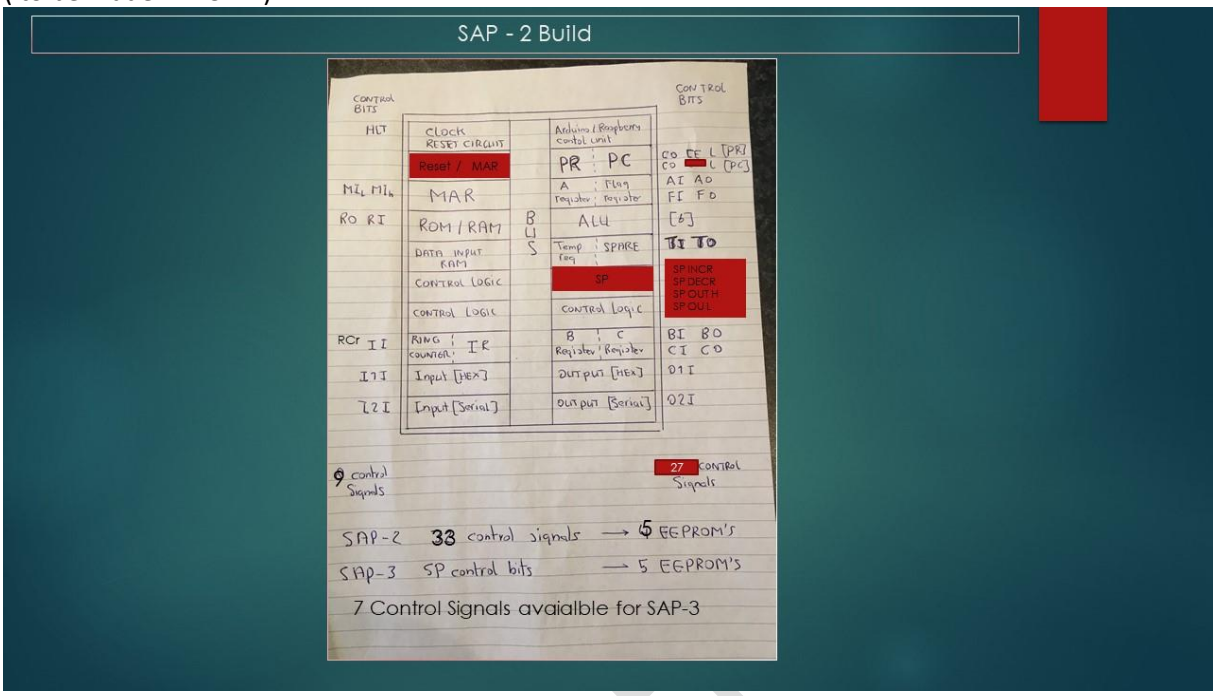
First I made a layout of how I thought the modules had to go on the backplane, nice word for a big chunk of plywood. It looked something like this.

(to be made in KiCad)



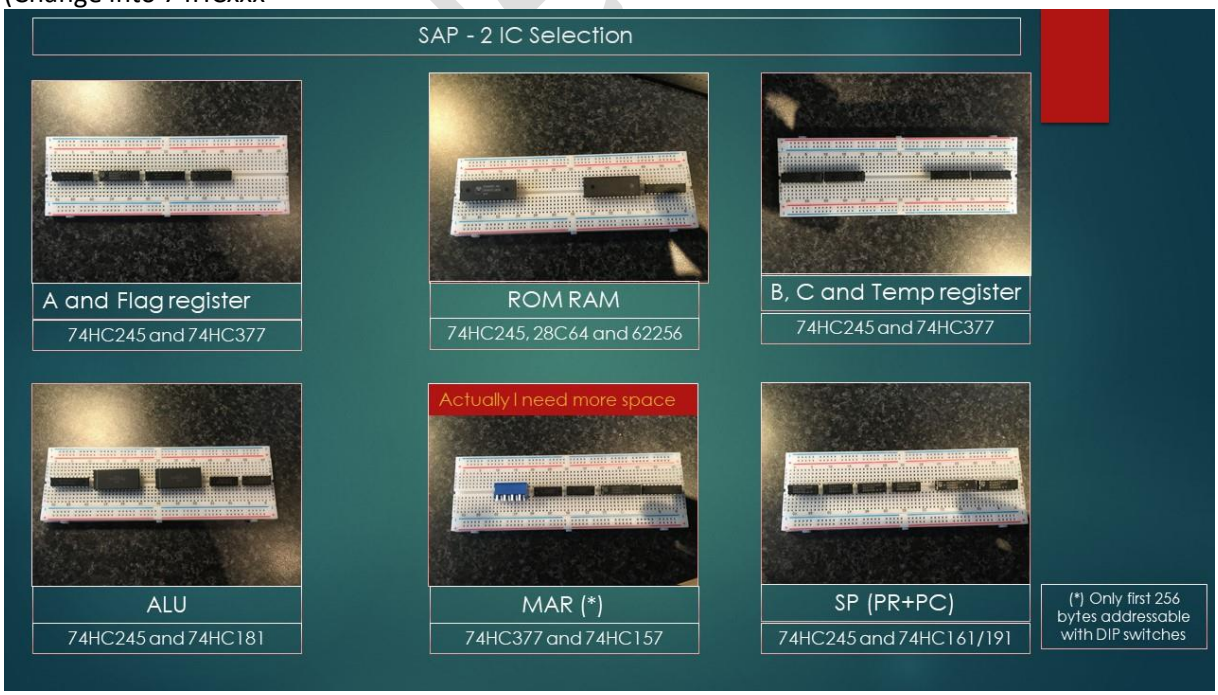
I re-arranged the stack pointer and had more control signals and after some proper counting, I ended up with this.

(to be made in KCAD)

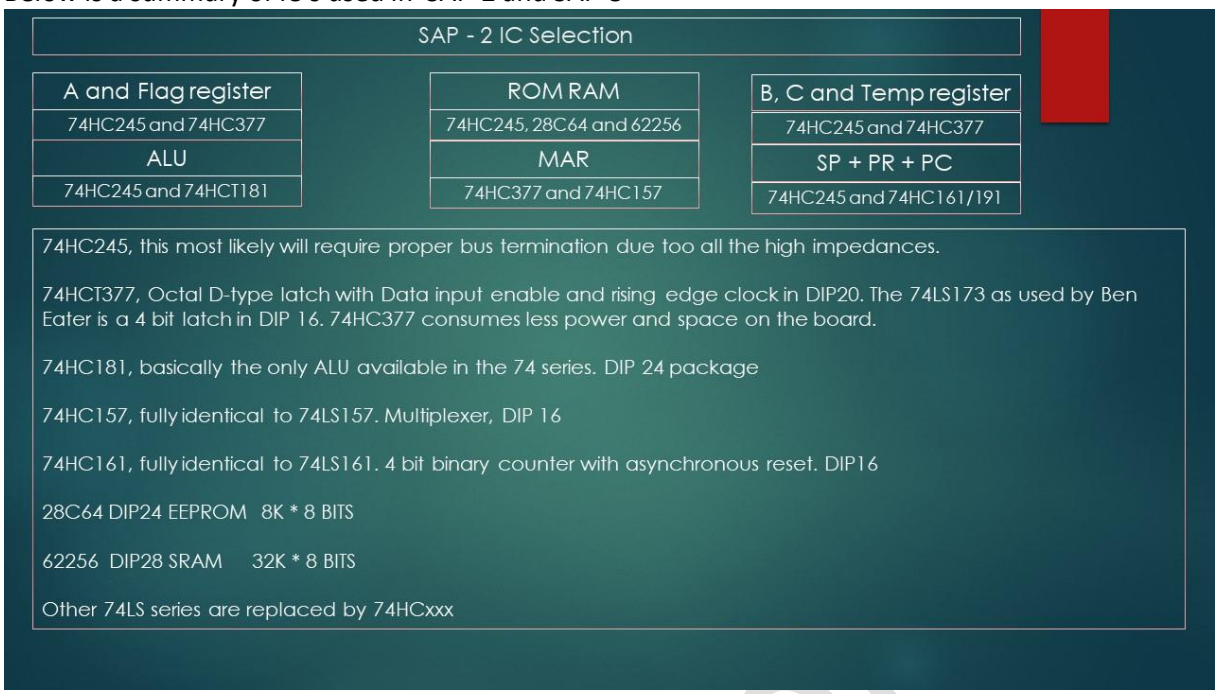


Then I did some trial fitting of how it would all go on breadboards. I made a mistake with the MAR for a 16 bit address you need 16/4 = 4 74HC157's, and not 2.

(Change into 74HCxxx



Below is a summary of IC's used in SAP-2 and SAP-3



Chapter 17, List of used Components lists the components used, type and quantity.

AND THEN IT WAS FINALLY TIME TO ACTUALLY START BUILDING.

Chapter 9, Tips while building the computer, Testing

Beside the lessons learned from u/lordmonoxide on Reddit sub channel the 8 bit on how to build the 8 bit breadboard computer and some other tips which can be found in this document in chapter 5, there is one other important thing while building the SAP-2 and 3. And that is TESTING it cannot be said enough to test as much and extensively as possible. Once you connect everything together like connecting the modules to the bus and connecting the control signals from the EEPROM's and then you really start testing, well it makes debugging a lot harder. The next paragraphs deal with testing.

Below a picture of my work bench, and it really helps a lot to have this once you need to troubleshoot.

Testing

To start, unfortunately of all technical disciplines electronics is one of the least forgiving. A civil design will still work even if you miss it by 5mm (or more, sorry civil engineers). In electronics 1 mistake => zippo result. You have two options, get another hobby or get used to this and accept the consequence. The consequence is: troubleshooting is part of the job, no way around it.

Next I strongly suggest that you read and fully understand the article below:
http://web.mit.edu/6.101/www/reference/TheArtofDebuggingCircuits.pdf

Below I took from RC2014.co.uk (with permission), I edited it to remove what's not applicable but I think this can be very useful.

Most problems can be diagnosed by looking at it. !

Without power applied

Take a look at every connection. Correctly wired and properly plugged in the board.

Are the modules individually properly tested, more on that later.

Are the correct value resistors and capacitors fitted. Check the individual assembly guides for visual confirmation.

Are the correct ICs in the correct place. Note that 74LSxx and 74HCTxx chips are interchangeable but not with 74HCxx.

Is the orientation of the ICs correct. Are the IC pins all seated cleanly in the breadboard. Make sure none of the pins are bent underneath the chip.

If powering with 5v from a lab power supply set the proper max current, should not be too low.

With power applied

Is power properly being applied?

Is nothing getting way too hot or is smoke part of your build. Turn off the power directly

Trouble shooting with a multimeter

There are two types of test that are easy to do with a multimeter: continuity test and voltage test.

Continuity test

Set your multimeter to the continuity or beep setting. If you're not familiar with how to do this, consult the documentation that came with it, or use the internet. This is a very useful mode to see if you have any doubt that something might not be properly connected. Power must be switched off for this.

Check the power supply

Measure Vcc on all chips on the board. This should be above 4.7 Volt.

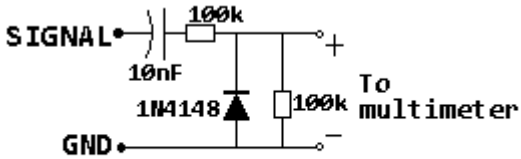
Signal Tracer Circuit

On the RC2014-Z80 Google Group, a simple signal tracer circuit can be found. I don't know what equipment you have but I'll assume you have a multimeter that can read DC voltage.

A signal tracer can be used to check if signals travels from A to B. An LED with current limiting resistor can also be used but less suitable for higher frequency signals. There is no longer much difference between pulsing and permanently on.

One of the checks you want to do is to check if there is a clock. Unfortunately multi-meters are not really suitable to check clock signals. So you need to construct an AC probe to read higher frequency clock signals, once your build is stable and you want to increase the clock frequency. Luckily, this is easy and cheap to do.

You take two 100k resistors, a 10 nF capacitor, a 1N4148 diode and to construct the following signal tracer:



SIMPLE SIGNAL TRACER

The component values are not critical at all. If you have 10k resistors, use 10k resistors. If you have a 100nF capacitor, use 100 nF.

Connect the 'signal' end to the CLK signal, or any other high frequency signal, and GND to GND. Set your multimeter to DC volts. The meter will read more than 1 V if there is a changing signal on the CLK and much less (it should show something close to 0V) if there is a static signal (either 5V or 0V).

These tests above will give you more information where the problem lies. Of course, if you have an oscilloscope, use that to actually look at the signals instead of using the probe.

Logic Probe

Logic probes are also a very useful tool for tracking signals. The better logic probes also have a pulse signal indication

Testing each module

It cannot be said enough test each module very very careful before you connect everything together to the bus and control signals.

Don't proceed to the next module before the module you are working on is 100% correct. So an approach like "I did not build it according the design from Ben Eater but somehow it works" will bring you nowhere. I have seen things like "well I placed a 10 uF capacitor on RI and now it works". NO, this is incorrect. If you don't know exactly what you change on a schematic than don't do it.

Remember if you have an error: THE CIRCUIT BEHAVES EXACTLY AS YOU BUILD IT, this however can be different from what you expect how it should behave. This is also a good analysing tool, which modification do I need to make in the schematic to see it behave as it does.

Take for instance the program counter

Make a temporary bus with nothing else connected to it except bus status LED's. Try writing values to the bus and ty to load values from the bus. See if the CE signal works properly. Try this at higher frequencies. If you don't have a logic analyser place a digital camera above the computer and play it on your PC to see if it works correct.

Once you have tested all the modules, they can be connected to the bus. Use jumpers for all control signals and make them inactive. And then step by step start testing.

Bring a value from PC on the bus, try to write it in RAM. Try to write it in the A-register. Use several values like 0x55 or 0Xf0. Try signal flow in all possible directions. Try to build a counter with the A register, the temporary register and the HEX display.

And again work the problem if you have one, don't proceed to the next step.

Then replay the 28+14 calculation as shown by Ben Eater, by manipulating the control wires with jumpers.

And for SAP-2 this is a serious amount of testing. Only then as last step connect the control wires.

AND THAT WAS FOR ME WHERE THE TROUBLES STARTED.

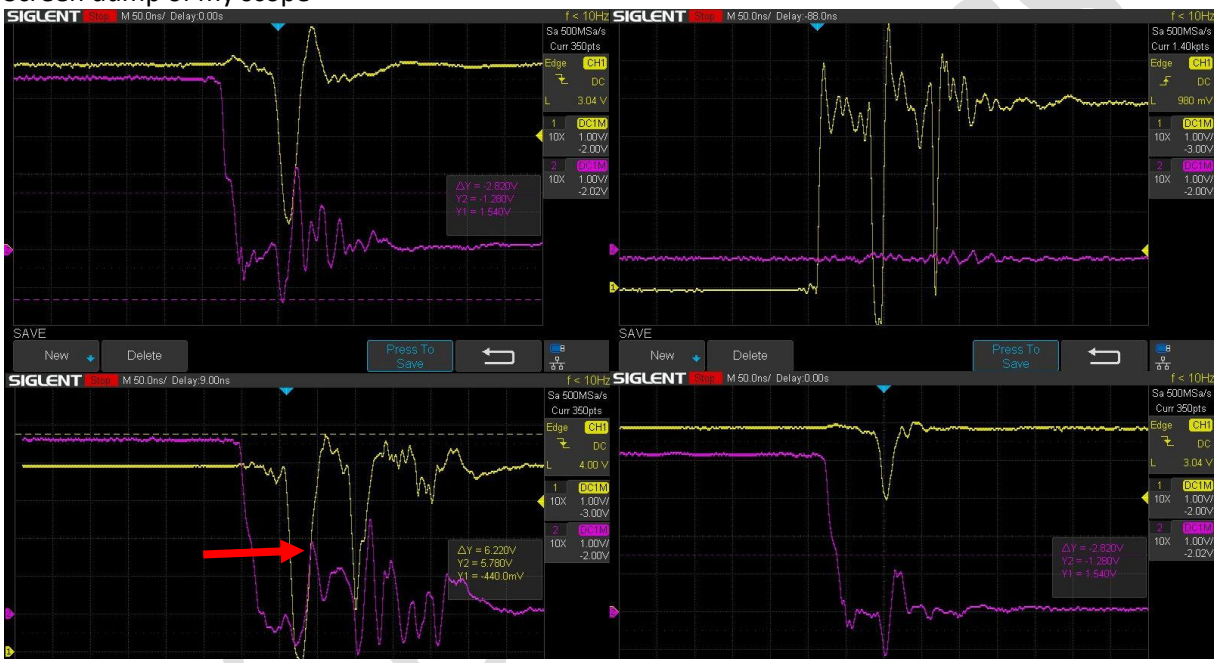
Chapter 10, Troubles with my SAP-2 design and Solutions

I did all the testing as previously described. Then I connected the control signals, entered a simple program and executed this in manual mode step by step. Troubles; random behaviour and total not stable.

Entering two values in RAM, and entered a simple program to add the two values. Well 5 + 5 showed up as 7, 88, 25 etc and sometimes 10.

After some analysing I found the cause of these problems, below scope screenshot was the root cause.

Screen dump of my scope



What shows above screen dumps from my scope.

The 8 bit computer is controlled by means of the data outputs from the EEPROM, nothing new. But when their address changes the data outputs show glitches, they don't stay stable while you would expect that in many cases. This happens on ALL data output signals of the EEPROM'S. The glitches are short but long enough to cause problems. That's however according the datasheet, when the address changes the output is not defined for some time, while OE is low. This however causes many issues since these lines control the computer. Top pictures and bottom-left picture are screen dumps from my scope of such glitches.

As example the glitches would activate all 74HC245 to write to the bus. I simulated this by making all OE signals of the 74HC245 IC's low with jumpers. First off all this obviously causes total confusion on the bus, but as side effect the power consumption of the computer increased from 0.7 amp on average to over 2.1 amp.

I have installed pull-up resistors on active low signals and pull down resistor on active high signals on the data output lines of the EEPROM's.

The result is shown on the picture on the bottom on the right. Adding the pull-up and pull-down resistors helped a lot.

This is a more detailed explanation on the glitches. The most clear picture what is happening is the bottom left picture.

Now the yellow line is the control line, II, of the instruction register and the purple line the clock. Note the horizontal scale on my scope was set to 50ns/div. The means that the time span between the horizontal lines is 50 nSec.

I marked the point to note with a red arrow.

Due to these glitches the control line of the instruction register goes low, the yellow line. But a short moment later there is also a spurious clock signal. This leads to whatever is that moment on the bus will be clocked into the instruction register.

So suddenly you have a new random instruction clocked into the instruction register.

Similar things can happen for other registers. Or extra pulse on the step or program counter, and suddenly you jump from T0 to T2 or the program counter counts a step too fast.

The reason why the clock goes low-high is caused by two things in my opinion. First of all, all register output control signals are pulled low by these glitches since they appear on all the data output lines of the EEPROM. So they all want to write to the bus, this obviously creates total confusion on the bus but there is also another side effect. When this happens the current drawn by the complete board goes up from an average 0.7 Ampere to over 2.1 Ampere. I measured this by replacing all control lines of the tri-state bus IC's with jumpers and tied them all low. This leads to a severe increase in current drawn by the board and causes a short dip in Vcc, also on the clock IC's.

On top of that the HLT is also connected to the EEPROM's and is connected to the clock module. This causes more problems and results in unwanted pulses on the clock line, seen as extra clock cycles.

The glitches are 50 nsec and sometimes you get double glitches, but the complete thing happens within let's say 200 nsec. 1/5 of a millionth of a second.

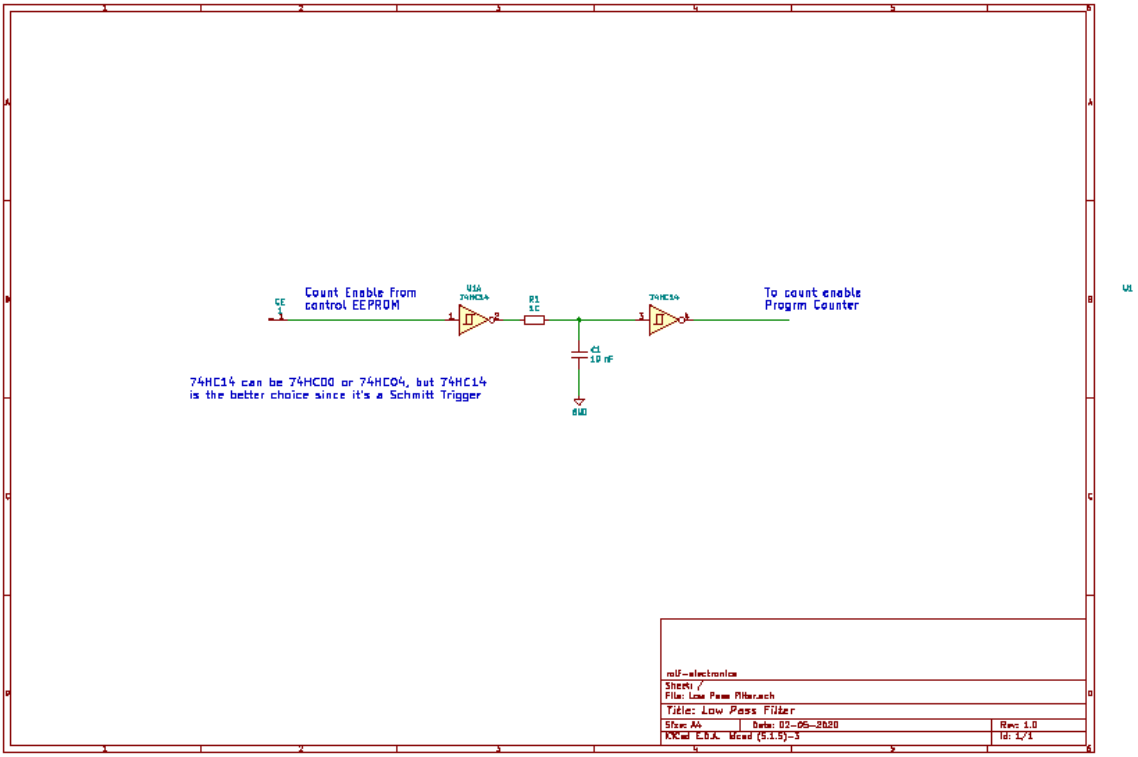
Without oscilloscope you would be utterly clueless of what's going on.

One of the last expansions I will try to implement is altering the clock control circuit, to avoid these glitches in total.

Low Pass filters

The installation of pull-up and pull down resistor helped a lot but still the design was not 100% stable. After more testing and measuring with the scope I installed low pass filters on the following control lines:

- Count Enable of the Program Counter
- Instruction In of the instruction register
- Count Enable of the step counter
- And later on the control signals of the stack pointer



I will load the .pdf of the KiCAD on my github account, the symbols cannot be made larger.

https://github.com/rolf-electronics/The-8-bit-SAP-3/tree/KiCAD-Drawiings

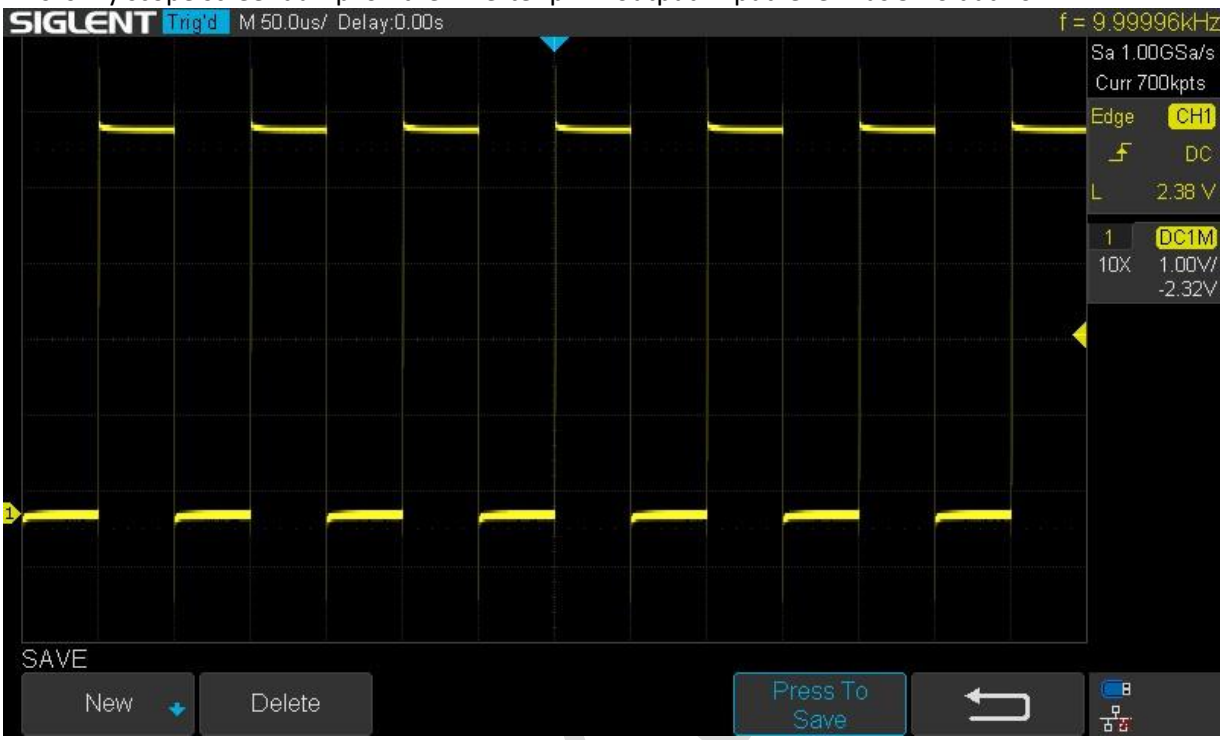
Above is an example how to build a low pass filter, in my build I used the 74HC14 which are Schmitt trigger HEX inverters. They give a more clean output signal.

There is a very interesting video on You Tube from Dave from EEVblog on bad clock signals sent to a 74HC161 counter.
https://www.youtube.com/watch?v=Ht48vv0rQYk&t=277s

After installing the pull-up and pull down resistor, adding the low pass filters and changed from 74HCTxxx logic to 74HCxx the computer worked like a charm.

This shows the effect of adding a low pass filter in a control line

This is my scope screendump on the inverter pin 4 output. Input level was 5 Volt at 10 KHz



This is my scope screendump on the inverter pin 4 output. Input level was 5 Volt at 1 MHz. Note that the signal gets distorted and the scope measures 1.3 MHz

Chapter 11, Choices I made in building SAP-3

Stack Pointer

A stack is a portion of the memory and it's a sequenced storage, the stack pointer is used to store or retrieve data from the stack, since it points to the current location on the stack.

In contrary to the step counter and the PR and PC the stack pointer should be able to count up and down. Since it must be able to execute PUSH and POP instructions. PUSH is bringing something to the stack and POP is getting something from the stack. This something can be a return address but also the content of any of a general purpose register.

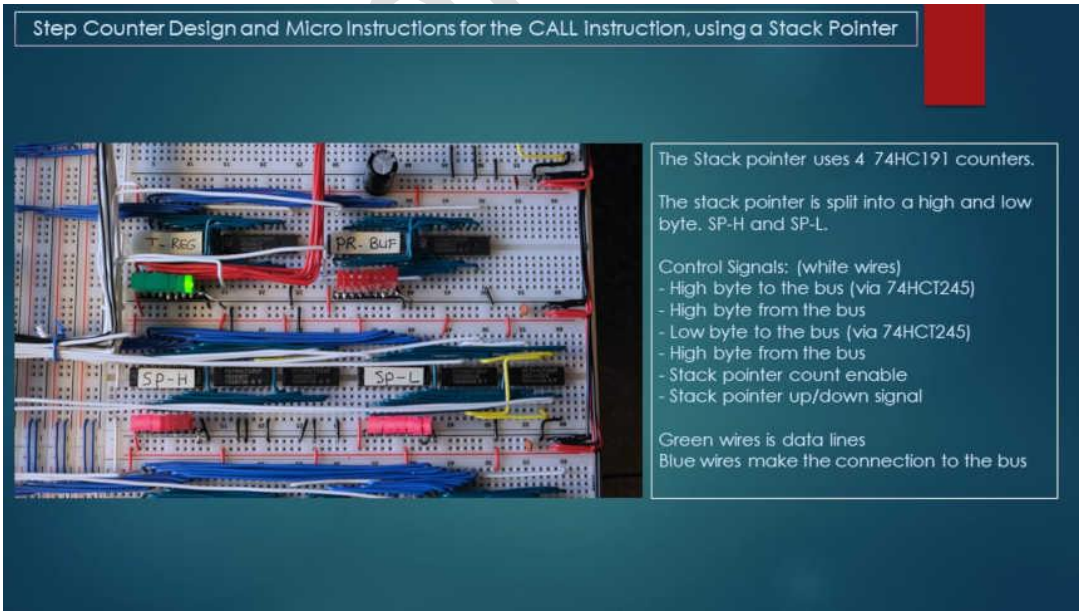
An instruction like PUSH_A, places something on the stack and then decreases the value of the stack pointer. An instruction like POP_A, increases the value of the stack pointer and gets a value from the stack

As said this increasing and decreasing of the value of the stack pointer can only be done by a device which can count up and down, for this purpose I used the 74HC191, and since it's 4 bits I needed four of them. Due to the fact that I have an 8 bit bus I split the stack pointer in a SP-H and an SP-L, identical to PR and PC.

Now assume you want to PUSH something on the stack. The sequence I have chosen is to write to the stack and then decrease the value of the stack pointer, the other way around is also perfectly possible. The following micro instruction steps must be taken.

Wanting to write something to the stack means that the value from SP-H and SP-L must be loaded into MAR-H and MAR-L. Once that's done the MAR points to the bottom of the stack. A stack always works top down. You start at a certain address and while storing data you decrease the value of the stack pointer. You could build a stack pointer which works the other way around, but it's not common practice.

So we were at the point where the MAR was loaded with the bottom of the stack. Now you can use the RI command to write for instance the value of the A register to the stack. And as last action you have to decrease the value of the stack pointer



Picture of the stack pointer

THE instruction with the most micro instructions, the CALL instruction

Now how did I build the CALL instruction in SAP-3, again this is what I dreamed up none of the books referred to provides detailed info on this.

I will break this down into 8 steps.

Let's say the instruction is CALL xx yy, whereby xx is the page and yy the pc value to jump to.

1/ Fetch the instruction and load xx into the A register and yy to temporary register. PR and PC now indicate the return address.

2/ Load MAR-H with SP-H and MAR-L with SP-L.

3/ Decrease the stack pointer value.

4/ Now load the value of PR into RAM, read stack. The return address page is now pushed on the stack.

5/ Load MAR-H with SP-H and MAR-L with SP-L. This can be optimised by deleting load MAR-H with SP-H, this is already done in step 2. This would limit the stack to 128 return addresses, but that is more than enough. I have to see the first program built on a breadboard computer with a program using a nested level of 127. Ain't gonna happen.

6/ Decrease the stack pointer value.

7/ Now load the value of program counter into RAM, read stack. The return address program counter value is now pushed on the stack. The complete return address is now saved on the stack.

8/ Final step load the page register with the content of the A register and load the program counter with the content of the temporary register. This actually executes the jump to address xx yy.

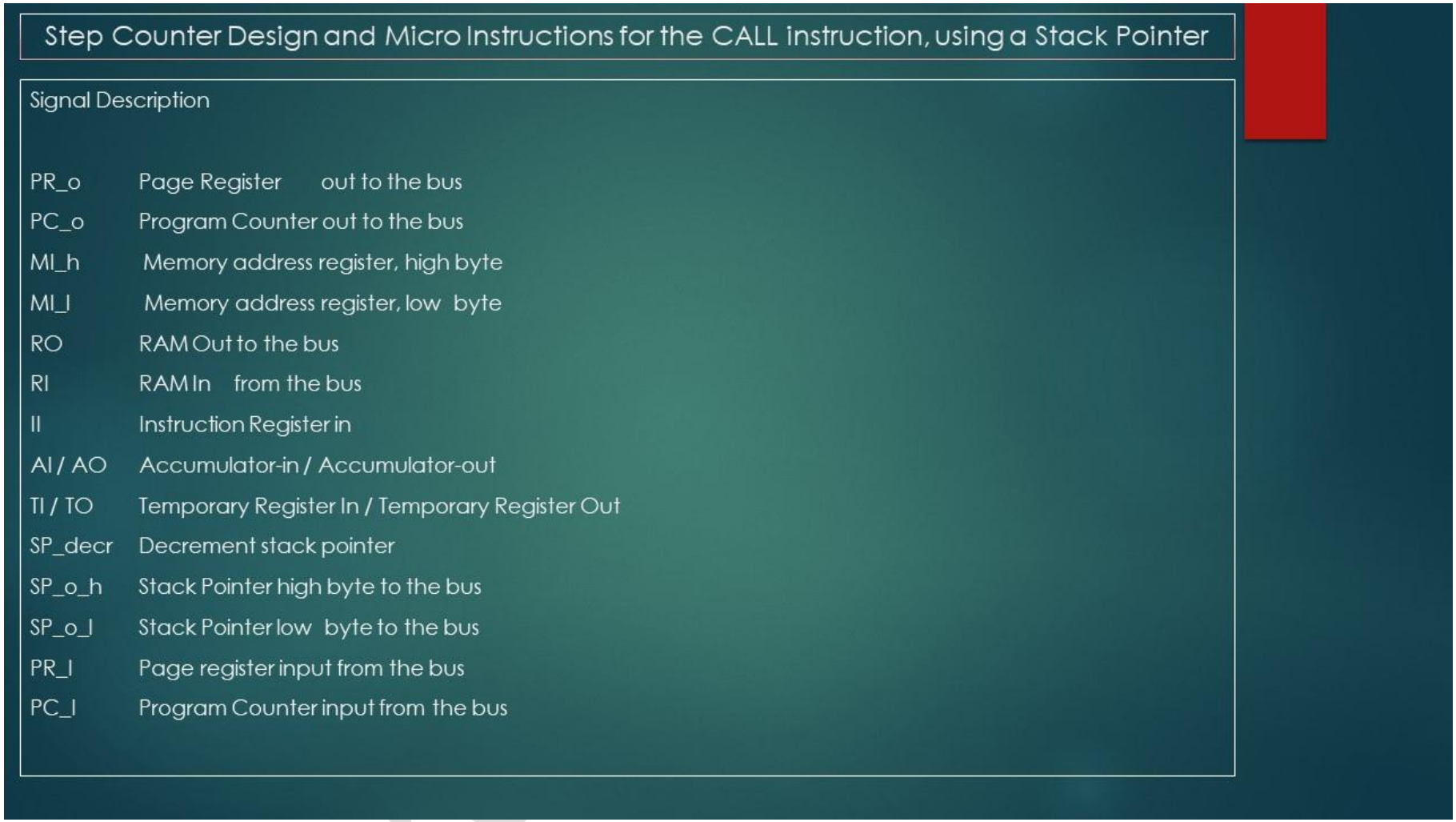
For this instruction I needed 15T states as can be seen on the next slides, T0 to T14, which brought the necessity of a 4 bit step counter. But it is 3 states more efficient than the CALL implementation in SAP-2. And basically nested CALL can be executed 128 levels deep. After this page three pages are shown on the CALL instruction with one page detailing the micro instructions.

The RET, return instruction

The return instruction is a lot simpler since no return address has to be saved on the stack.

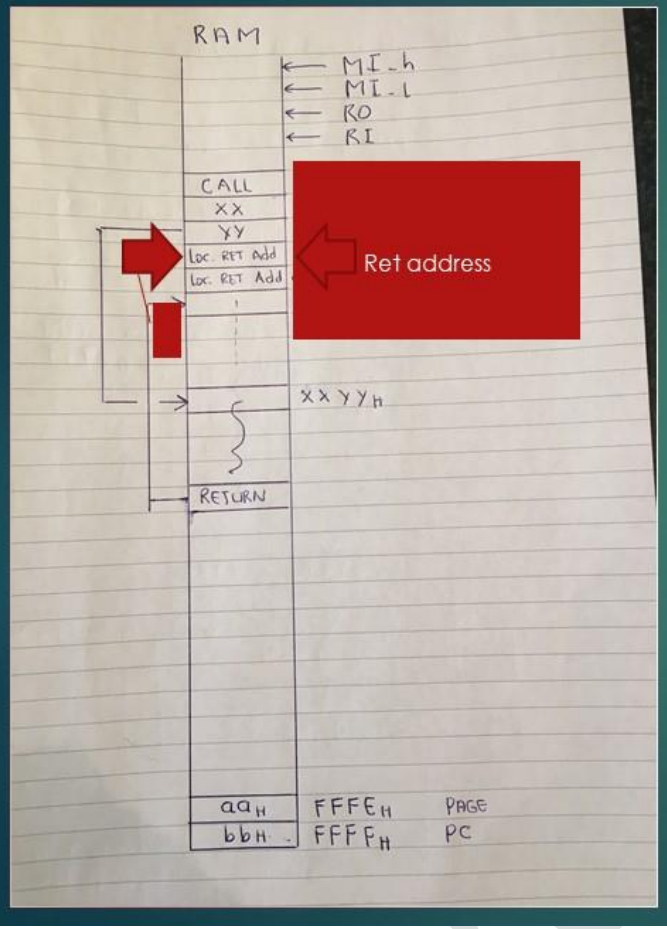
(to be detailed later)

Abbreviations used



General functionality CALL instruction

Ring Counter Design and Micro Instructions for the CALL instruction USING a Stack Pointer



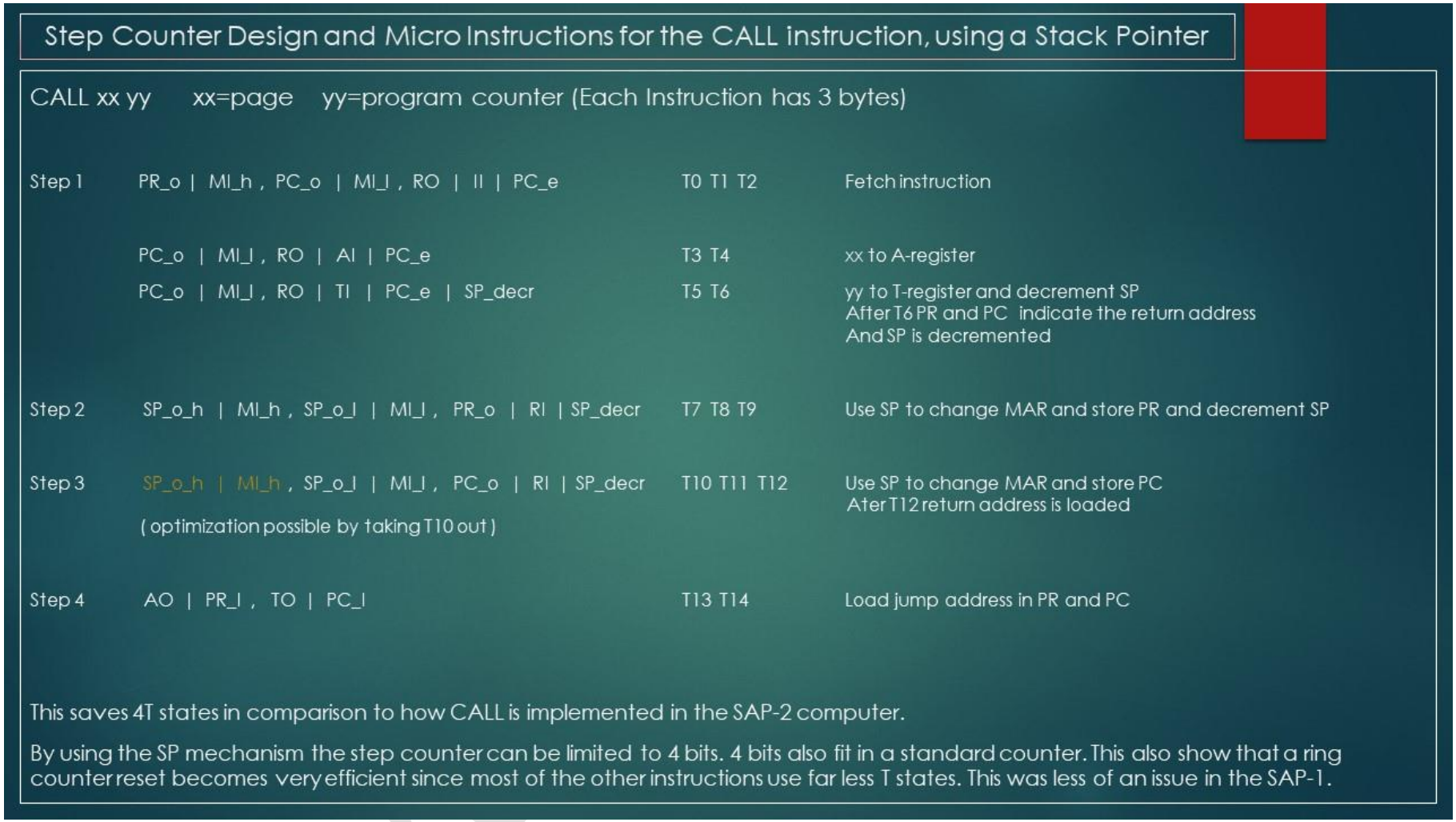
Step 1 Fetch Instruction and xx to A-reg and yy to T-reg And decrement SP with dedicated control signal (*) After step 1 PR and PC point to the return address	7T
Step 2 Use SP to store byte **SP_o_h to MI_h**, SP_o_l to MI_l PR_o | RI | SP_decr	3T
Step 3 Use SP to store byte (can be optimed to 2 T states) **SP_o_h to MI_h**, SP_o_l to MI_l PC_o | RI	3T
Step 4 A-reg to PR T-reg to PC	2T
Total number of T states	**15**
Number of T states counting from T0	**14 (13)**

This fits in a 4 bit ring counter

(*)

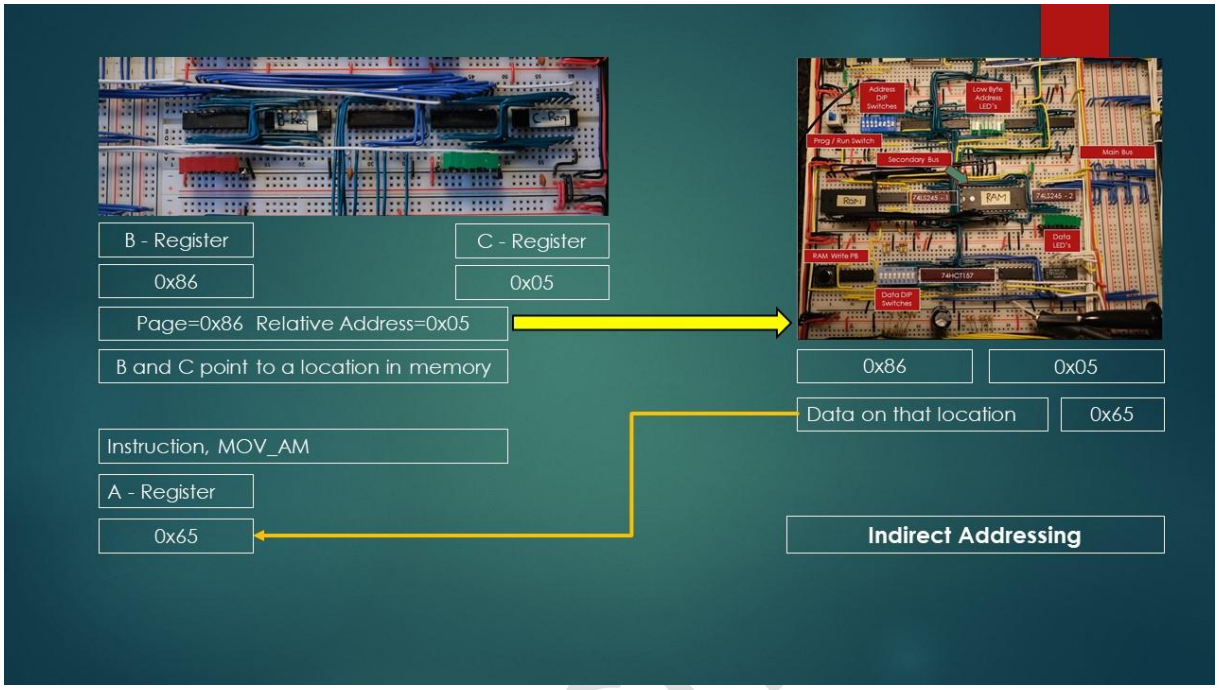
Using a dedicated control signal to decrement the SP, avoids doing this in the ALU, and saves T states.

The micro instruction



Indirect addressing

For the B and C register I made a couple of separate instructions in order to enable indirect addressing. Below picture illustrates how indirect addressing works. I did not make any hardware modifications to implement this, it only requires definition of a number of micro instructions



The B and C register holds data which act as a pointer to a location in memory. The micro instructions of MOV-AM loads the content of respectively the B and C register into MAR-H and MAR-L. And next the content of RAM/ROM is loaded into the A register.

You can use this feature to define an array and with the B and C register manipulate data in the array. As example somewhere in memory, lets say at start location 0x8780 you load “This is the 8 bit computer/”. You can then use indirect addressing to get one character after the next to display this on a screen. The “/” act as termination so that the loop knows that the last character has been reached.

My instruction set for the SAP-3

Following is my instruction set definition, 110 instructions. Now this sounds way over the top and an incredible amount of work. Well once you get the hang of it in writing micro instruction you can copy a lot from previous instructions for quickly making a new instruction. Obviously you can only do this if you fully understand the computer, how to program the EEPROM and what each instruction exactly is supposed to do. But that's the advantage of designing and building things yourself, that you know.

The computer has an 8 bit data bus.
It has an 8 bit page register(PR)and an 8 bit program counter(PC), there is a carry-over from PC to PR
RAM and ROM are 8 bits data. Currently 32K RAM and 2K ROM
Instruction register is 8 bits
There are three flags. Zero, Carry and parity

Each instruction takes three bytes
First byte: opcode
Second byte: mostly 0(zero) or the page register referred to
Third byte: often 0(zero) or the program counter referred to or an immediate value
Range Page Register: 0 to 0xFF. Range Program Counter: 0 to 0xFF
Instructions are set up in blocks of 16 opcodes, last 4 bits range from 0000 to 1111

All instructions with opcode MSB=0 are non flag related.
All instructions with opcode MSB=1 are flag related

Example on below instruction set
LDA, 0x00, 0x0f,
This should deliver three sequential bytes being:
0b00000000 (opcode)
0b00000000 (first operand)
0b00001111 (second operand)

In the examples were 0's are shown, they are manditory for the hardware to be there

My sap-3 plus instruction set

Block 1 Accumulator Instructions					
LDA	0b00000000	LDA	A,	B,	Load accumulator with content memory location page=A pc=B
STA	0b00000001	STA	A,	B,	Store content accumulator to memory location page=A pc=B
MVI_A	0b00000010	MVI_A	0	a,	Load A register with a, range 0 to 0xFF
MVI_B	0b00000011	MVI_B	0	a,	Load B register with a, range 0 to 0xFF
MVI_C	0b00000100	MVI_C	0	a,	Load C register with a, range 0 to 0xFF
MOV_AB	0b00000101	MOV_AB	0	0	Move content B register to the A register
MOV_AC	0b00000110	MOV_AC	0	0	Move content C register to the A register
MOV_BA	0b00000111	MOV_BA	0	0	Move content A register to the B register
MOV_BC	0b00001000	MOV_BC	0	0	Move content C register to the B register
MOV_CA	0b00001001	MOV_CA	0	0	Move content A register to the C register
MOV_CB	0b00001010	MOV_CB	0	0	Move content B register to the C register
ADD_B	0b00001011	ADD_B	0	0	Add content B to the A register, flags checked
ADD_C	0b00001100	ADD_C	0	0	Add content C to the A register, flags checked
SUB_B	0b00001101	SUB_B	0	0	Subtract content B from the A register (A-B), if result does not go through zero, carry flag set. if through zero carry flag reset
SUB_C	0b00001110	SUB_C	0	0	Subtract content C from the A register (A-C), if result does not go through zero, carry flag set. if through zero carry flag reset
	0b00001111				This means that here is space for an additional instruction

Block 2 Accumulator Instructions					
INR_A	0b00010000	INR_A	0	a,	Increment A register with a, range 0 to 0xFF. For instance a=1
INR_B	0b00010001	INR_B	0	a,	Increment B register with a, range 0 to 0xFF
INR_C	0b00010010	INR_C	0	a,	Increment C register with a, range 0 to 0xFF
DCR_A	0b00010011	DCR_A	0	a,	Decrement A register with a, range 0 to 0xFF
DCR_B	0b00010100	DCR_B	0	a,	Decrement B register with a, range 0 to 0xFF
DCR_C	0b00010101	DCR_C	0	a,	Decrement C register with a, range 0 to 0xFF
LDB	0b00010110	LDB,	A,	B,	Identical to LDA, but now B register

STB	0b00010111	STB,	A,	B,	Identical to STA, but now B register
LDC	0b00011000	LDC,	A,	B,	Identical to LDA, but now C register
STC	0b00011001	STC,	A,	B,	Identical to STA, but now C register
ADD_M	0b00011010	ADD_M	A,	B,	Add content memory location A,B to the A register. Without carry
SUB_M	0b00011011	SUB_M	A,	B,	Subtract content memory location A,B from the A register (A-M). Without carry
ST_SPH	0b00011100	ST_SPH,	A,	B,	Store SPH on loacation Page=A PC=B
ST_SPL	0b00011101	ST_SPL,	A,	B,	Store SPL on loacation Page=A PC=B
LD_SPH	0b00011110	LD_SPH,	A,	B,	load SPH from loacation Page=A PC=B
LD_SPL	0b00011111	LD_SPL,	A,	B,	Load SPL from loacation Page=A PC=B

Block 3 Logic instructions					
CMA	0b00100000	CMA	0	0	Complement the accumulator
ANA_B	0b00100001	ANA_B	0	0	AND accumulator with content B register
ANA_C	0b00100010	ANA_C	0	0	AND accumulator with content C register
ORA_B	0b00100011	ORA_B	0	0	OR accumulator with content B register
ORA_C	0b00100100	ORA_C	0	0	OR accumulator with content C register
XRA_B	0b00100101	XRA_B	0	0	XOR accumulator with content B register
XRA_C	0b00100110	XRA_C	0	0	XOR accumulator with content C register
ANI	0b00100111	ANI	0	0b11110000	AND A register with 0b11110000
ORI	0b00101000	ORI	0	0b11001100	OR A register with 0b11001100
XRI	0b00101001	XRI,	0	0b10101010	XOR A register with 0b10101010
RAL1	0b00101010	RAL,	0	0	Rotate ACC left LSB=1
RAL0	0b00101011	RAR,	0	0	Rotate ACC left LSB=0
	0b00101100				This means that here is space for an additional instruction
	0b00101101				This means that here is space for an additional instruction
	0b00101110				This means that here is space for an additional instruction
	0b00101111				This means that here is space for an additional instruction

Block 4 Input Output instructions					
OUT_A	0b00110000	OUT_A	0	0	Output the content of the A-register to the HEX display
OUT_B	0b00110001	OUT_B	0	0	Output the content of the B-register to the HEX display
OUT_C	0b00110010	OUT_C	0	0	Output the content of the C-register to the HEX display
OUT_M	0b00110011	OUT_M	A,	B,	Output the content of memory location page=A pc=B to the Hex display
LCD_C	0b00110100	LCD_C	0	a,	Control charachter a to LCD Display. Range 0 to 0xFF
LCD_D	0b00110101	LCD_D	0	b,	Data charachter b to LCD Display. Range 0 to 0xFF
OUTLC_A	0b00110110	OUTLC_A,	0	0	Content A register as command to LCD display
OUTLD_A	0b00110111	OUTLD_A,	0	0	Content A register as data to LCD display
OUTLD_B	0b00111000	OUTLD_B,	0	0	Content B register as data to LCD display
OUTLD_C	0b00111001	OUTLD_C,	0	0	Content C register as data to LCD display
IN_A	0b00111010	IN_A,	0	0	Input from 4 pushbuttons to A-register. Not debounched
IN_B	0b00111011	IN_B,	0	0	Input from 4 pushbuttons to A-register. Non debounched
IN_C	0b00111100	IN_C,	0	0	Input from 4 pushbuttons to A-register. Non debounched
	0b00111101				This means that here is space for an additional instruction
	0b00111110				This means that here is space for an additional instruction
	0b00111111				This means that here is space for an additional instruction

Block 5 Various instructions, related to indirect adrresing					
JMP	0b01000000	JMP,	A,	B,	Jump to adress location with Page register=A and Program Counter=B. Should always be multiples of three
JMP_R	0b01000001	JMP_R,	0	B,	Jump to adress location with Program Counter=B. PR is not chnaged, relative jump on a page
NOP	0b01000010	NOP,	0	0	No operation
HLT	0b01000011	HLT,	0	0	Halt the computer
LXI_BC	0b01000100	LXI_BC	A,	B,	Load Register Pair BC. Page=B-register=A PC=C-register=B
INX	0b01000101	INX	0	a,	Increment C register with a. No change B register, 255 indirect adreses is sufficient
DCX	0b01000110	DCX	0	a,	Decrement C register with a. No change B register, 255 indirect adreses is sufficient

MOV_AM	0b01000111	MOV_AM	0	0	Move content adress pointed at by BC-register pair to A register
MOV_MA	0b01001000	MOV_MA	0	0	Move content A-register to adress point at by BC-register pair
MVI_M	0b01001001	MVI_M	0	a,	Move immediate value a to location pointed at by BC register
LDD	0b01001010	LDD,	A,	B,	Load D(SPH) with content memory location page=A pc=B
STD	0b01001011	STD,	A,	B,	Store content D(SPH) to memory location page=A pc=B
LDE	0b01001100	LDE,	A,	B,	Load E(SPH) with content memory location page=A pc=B
STE	0b01001101	STE,	A,	B,	Store content E(SPH) to memory location page=A pc=B
	0b01001110				This means that here is space for an additional instruction
	0b01001111				This means that here is space for an additional instruction

Block 6 Stack pointer instructions					
LXI_SP	0b01010000	LXI_SP,	A,	B,	Load Stack pointer with PR=A and PC=B
PUSH_A	0b01010001	PUSH_A,	0	0	Push the content of the A-register to the stack
PUSH_B	0b01010010	PUSH_B,	0	0	Push the content of the B-register to the stack
PUSH_C	0b01010011	PUSH_C,	0	0	Push the content of the C-register to the stack
PUSH_PSW	0b01010100	PUSH_PSW,	0	0	Push the PSW to the stack
POP_A	0b01010101	POP_A,	0.	0	Pop the content of the stack to the A-register to the stack
POP_B	0b01010110	POP_B,	0	0	Pop the content of the stack to the B-register to the stack
POP_C	0b01010111	POP_C,	0	0	Pop the content of the stack to the C-register to the stack
POP_PSW	0b01011000	POP_PSW,	0	0	Pop the content of the stack to the PSW
CALL	0b01011001	CALL,	A,	B,	CALL to adress with PR=A and PC=B. Save return adress on the stack
RET	0b01011010	RET,	0	0	Get return adress from the stack and executethe jump to that adress
	0b01011011				This means that here is space for an additional instruction
SPU	0b01011100	SPU,	0	0	Increment Stack pointer by 1
SPD	0b01011101	SPD,	0	0	Decrement Stacl pointer by 1
	0b01011110				This means that here is space for an additional instruction
Unvalid	0b01011111	PUSH_PRPC	0	0	CANNOT be used in normal programs !!!!!

Block 7 Instructions related to the 320*240 LCD screen					
LLCD_C	0b01100000	LCD_C,	0	a,	content a to large LCD as command
LLCD_P	0b01100001	LCD_P,	0	a,	content a to large LCD as parameter
LLCD_AC	0b01100010	LCD_AC,	0	0	send content A register as command
LLCD_AP	0b01100011	LCD_AP,	0	0	send content A register as parameter
LLCD_BP	0b01100100	LCD_BP,	0	0	send content B register as parameter
LLCD_CP	0b01100101	LCD_CP,	0	0	send content C register as parameter
	0b01100110				This means that here is space for an additional instruction
MOV_AD	0b01100111	MOV_AD,	0	0	Move content D register to the A register
MOV_AE	0b01101000	MOV_AE,	0	0	Move content E register to the A register
MOV_BD	0b01101001	MOV_BD,	0	0	Move content D register to the B register
MOV_BE	0b01101010	MOV_BE,	0	0	Move content E register to the B register
MOV_CD	0b01101011	MOV_CD,	0	0	Move content D register to the C register
MOV_CE	0b01101100	MOV_CE,	0	0	Move content E register to the C register
MVI_D	0b01101101	MVI_D,	0	a,	Load D register with a, range 0 to 0xFF
MVI_E	0b01101110	MVI_E,	0	a,	Load E register with a, range 0 to 0xFF
	0b01101111				This means that here is space for an additional instruction

Block 8 Interrupt related instructions					
EI	0b01110000	EI,	0	0	Enable Interrupt
DI	0b01110001	DI,	0	0	Disable Interrupt
SIM	0b01110010	IM,	0	a,	Set Interrupt mask with value a RST75 RST65 RST55
SIM_A	0b01110011	IM_A, 0,	0	0	Set interrupt Mask with value in the A-register, RST75 RST65 RST55
	0b01110100				This means that here is space for an additional instruction
MOV_DA	0b01110101	MOV_DA,	0	0	Move content A register to the D register
MOV_EA	0b01110110	MOV_EA,	0	0	Move content A register to the E register
MOV_DB	0b01110111	MOV_DB,	0	0	Move content B register to the D register
MOV_EB	0b01111000	MOV_EB,	0	0	Move content B register to the E register

MOV_DC	0b01111001	MOV_DC,	0	0	Move content C register to the D register
MOV_EC	0b01111010	MOV_EC,	0	0	Move content C register to the E register
	0b01111011				This means that here is space for an additional instruction
	0b01111100				This means that here is space for an additional instruction
	0b01111101				This means that here is space for an additional instruction
	0b01111110				This means that here is space for an additional instruction
	0b01111111				This means that here is space for an additional instruction

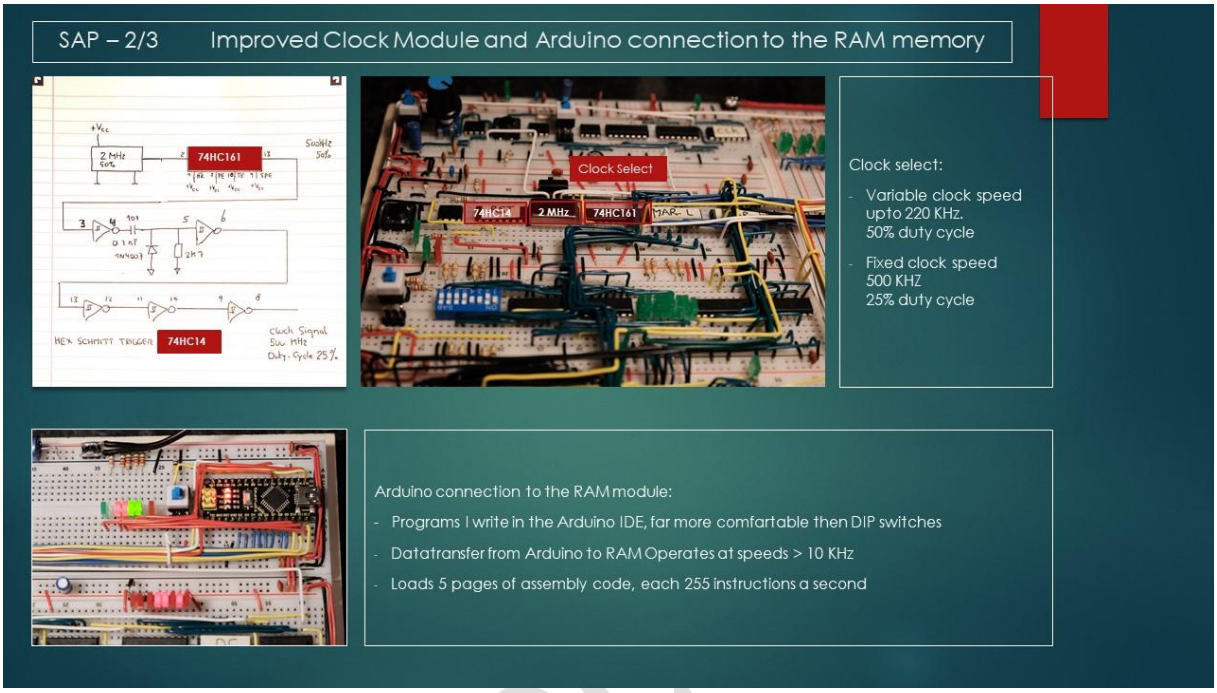
Block 9 Jump related instructions					
JZ	0b10000000	JZ,	A,	B,	Jump if zero flag set to PR=A PC=B
JC	0b10000001	JC,	A,	B,	Jump if carry flag set to PR=A PC=B
JP	0b10000010	JP,	A,	B,	Jump if positive to PR=A PC=B
JNZ	0b10000011	JNZ	A,	B,	Jump if zero flag not set to PR=A PC=B
JNC	0b10000100	JNC	A,	B,	Jump if carry flag not set to PR=A PC=B
JM	0b10000101	JP	A,	B,	Jump if negative to PR=A PC=B
	0b10000110				This means that here is space for an additional instruction
ADDC_B	0b10000111	ADDC_B	0	0	no flag set Add content B register to the A register, with carry
ADDC_C	0b10001000	ADDC_C	0	0	no flag set Add content C register to the A register, with carry
SUBB_B	0b10001001	SUBB_B	0	0	no flag set Subtract content B from the A register with borrow
SUBB_C	0b10001010	SUBB_C	0	0	no flag set Subtract content C from the A register with borrow
ADDC_M	0b10001011	ADDC_M	A,	B,	no flag set Add content memory location A,B to the A register. With carry
SUBB_M	0b10001100	SUBB_M	A,	B,	no flag set Subtract content memory location A,B from the A register (A-M). With carry
	0b10001101				This means that here is space for an additional instruction
	0b10001110				This means that here is space for an additional instruction
	0b10001111				This means that here is space for an additional instruction

Block 10 CALL and RETURN flag related instructions					
CZ	0b10010000	CZ,	A,	B,	Conditional Call if zero flag set to PR=A PC=B
CC	0b10010001	CC,	A,	B,	Conditional Call if carry flag set to PR=A PC=B
CP	0b10010010	CP,	A,	B,	Conditional Call if positive to PR=A PC=B
CNZ	0b10010011	CNZ,	A,	B,	Conditional Call if zero flag not set to PR=A PC=B
CNC	0b10010100	CNC,	A,	B,	Conditional Call if carry flag not set to PR=A PC=B
CM	0b10010101	CM,	A,	B,	Conditional Call if negative to PR=A PC=B
	0b10010110				This means that here is space for an additional instruction
	0b10010111				This means that here is space for an additional instruction
RZ	0b10011000	RZ,	0	0	Conditional Return if zero flag set, Get return address from the stack and execute the jump to that address
RC	0b10011001	RC,	0	0	Conditional Return if carry flag set, Get return address from the stack and execute the jump to that address
RP	0b10011010	RP,	0	0	Conditional Return if positive, Get return address from the stack and execute the jump to that address
RNZ	0b10011011	RNZ,	0	0	Conditional Return if zero flag not set, Get return address from the stack and execute the jump to that address
RNC	0b10011100	RNC,	0	0	Conditional Return if carry flag not set, Get return address from the stack and execute the jump to that address
RM	0b10011101	RM,	0	0	Conditional Return if negative, Get return address from the stack and execute the jump to that address
	0b10011110				This means that here is space for an additional instruction
	0b10011111				This means that here is space for an additional instruction

Chapter 12, Additions to the SAP-3, making it an SAP-3 plus

This chapter describes the additional functionality I built on top of the SAP-3.

Fixed crystal controlled clock speed possibility to allow for time critical applications.



Once you need fixed time intervals in your computer, like a fixed 1 second delay for an application or a 10 msec delay to control a display. Then it's not very practical to do that with the standard clock circuit. Try setting the potmeter every time at the exact value. Solution, I took a 2MHz crystal oscillator and divided the signal with an 74HC161.

I however have one problem in my build, above 220 KHZ the computer goes ballistic, with the AWG I found that I could increase the clock frequency to 500 KHz if I reduced the duty cycle to 25%. On above hand drawn schematic that's the function RC circuit with diode, the diode eliminates negative undershoot. The additional Schmitt triggers are required to clean up the signal, two could be removed , but for now I just linked them.

In the mean time I found the reason for the limit of 220 KHz.

An on-board RAM /ROM programmer

Why you would kind of need this becomes quickly quite obvious. Toggling in and debugging the 16 bytes of the SAP-1 is doable. If you extend your RAM to 256 bytes then this already gets unpleasant. Having 32K RAM, this gets undoable. So one has to find another way how to program the RAM.

Having installed all the resistors earlier on the data output lines of the EEPROM's now comes in handy. If you connect all the CE lines of the EEPROM's together and tie them high then virtually all EEPROM's are no longer there. All EEPROM's are disabled. The resistors avoid that the computer starts doing weird stuff, since all control lines are tied to an inactive level.

Now with another device you can start manipulating the control lines and as such control the functionality of the computer. You could place something on the bus and give as short pulse on MAR-H and next you do the same with MAR-L. In this way you can change the MAR to whatever address you need. The same you can do with RI, place a value on the bus and pulse RI and the value is written into RAM. And the ideal device to use for this functionality is an Arduino Nano. Fits nicely on the breadboard and connects to your PC.

Later I realized that in this way I could also program the EEPROM for the ROM memory. In doing so it avoids that I had to take it from the breadboard for re-programming. So now I have an on-board ROM programming feature.

With this I can program the RAM and ROM in the Arduino IDE. Not the most ideal solution but way better than toggling DIP-switches.

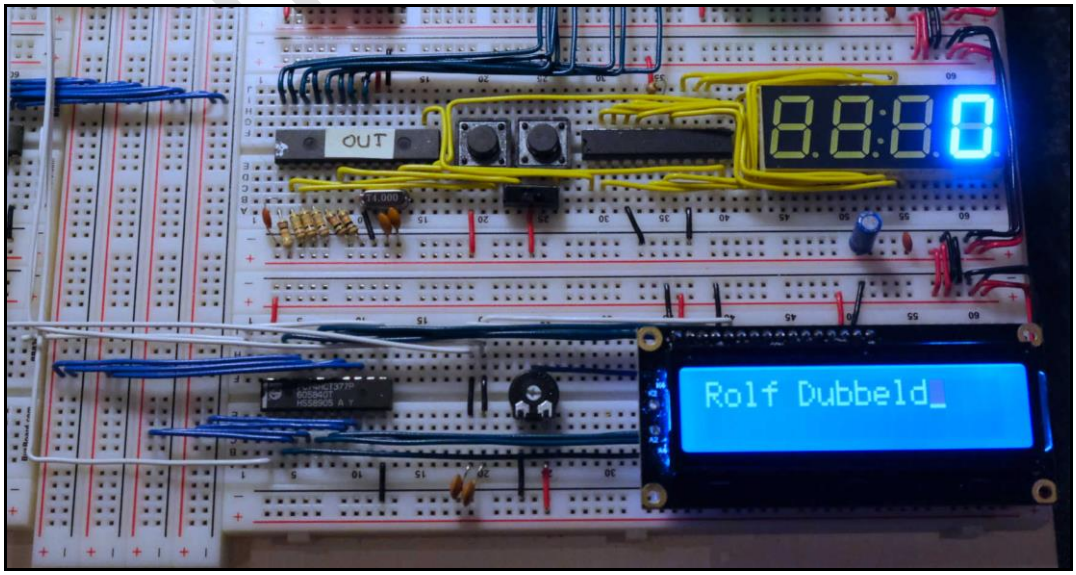
I will place a KiCAD drawing on my Github account on the wiring and post the code for the Arduino Nano also there.

It's kind of obvious how it works. You push the RAM/ROM program button, the Arduino makes the CE line of the EEPROM's high and control can be taken over by the Arduino Nano. I use an array in the Arduino IDE, or multiple array's, to make a progam and write it into RAM/ROM. Very much the same as Ben Eater programs an EEPROM.

Below is a link to a video showing onboard ROM updating:

https://www.reddit.com/r/beneater/comments/g6tfq5/my_latest_addition_onboard_eeprom_flahing_of_the/

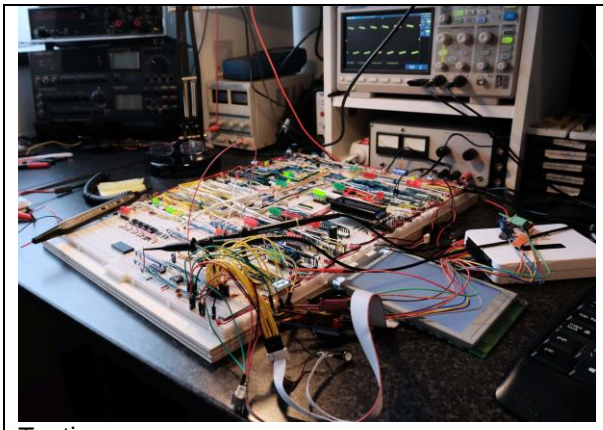
The 2*16 LCD screen



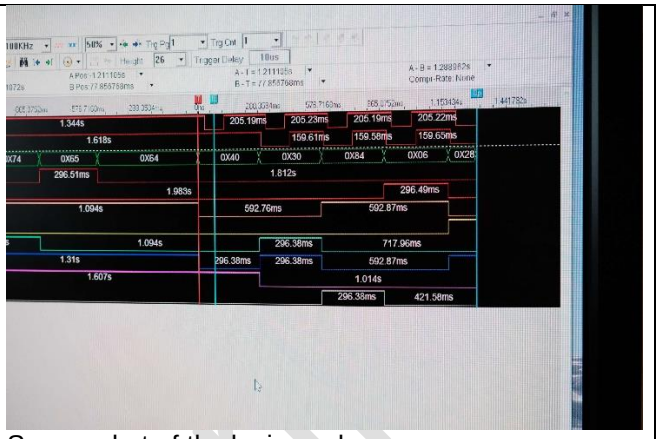
Basically I followed Ben Eaters Video for the LCD display on the 6502 and build it in the 8 bit.

The 320*240 LCD screen

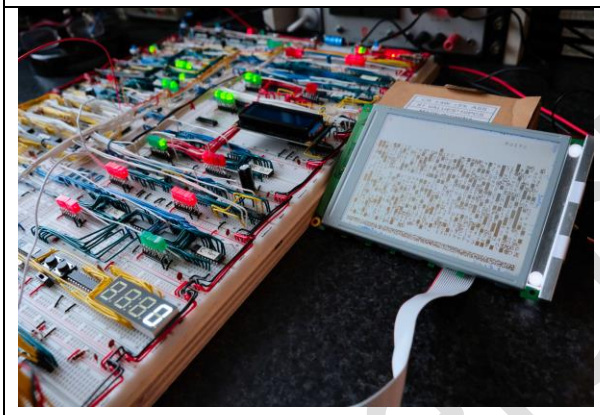
Before I go into detail how I build this, first below photo impression. I had to do some serious debugging to get this to work, needed my scope, logic analyser and logic probe. Well in the end it worked. I found the display's, three of them on, Dutch Ebay (Marktplaats) for 15 Euro. So I thought that's interesting to connect to the 8 bit computer.



Testing



Screen shot of the logic analyser



Clearing the graphic and text address area

Working

https://www.reddit.com/r/beneater/comments/flhpe4/connected_a_320240_lcd_screen_to_the_8_bit/

The display itself is a 320*240 display, and from what I could trace were manufactured by EPSON, type number is a AG320240A. This display is controlled by a SED1330, control looks a lot like the control on the 2*16 LCD controller with the exception that the SED1330 is more complicated.

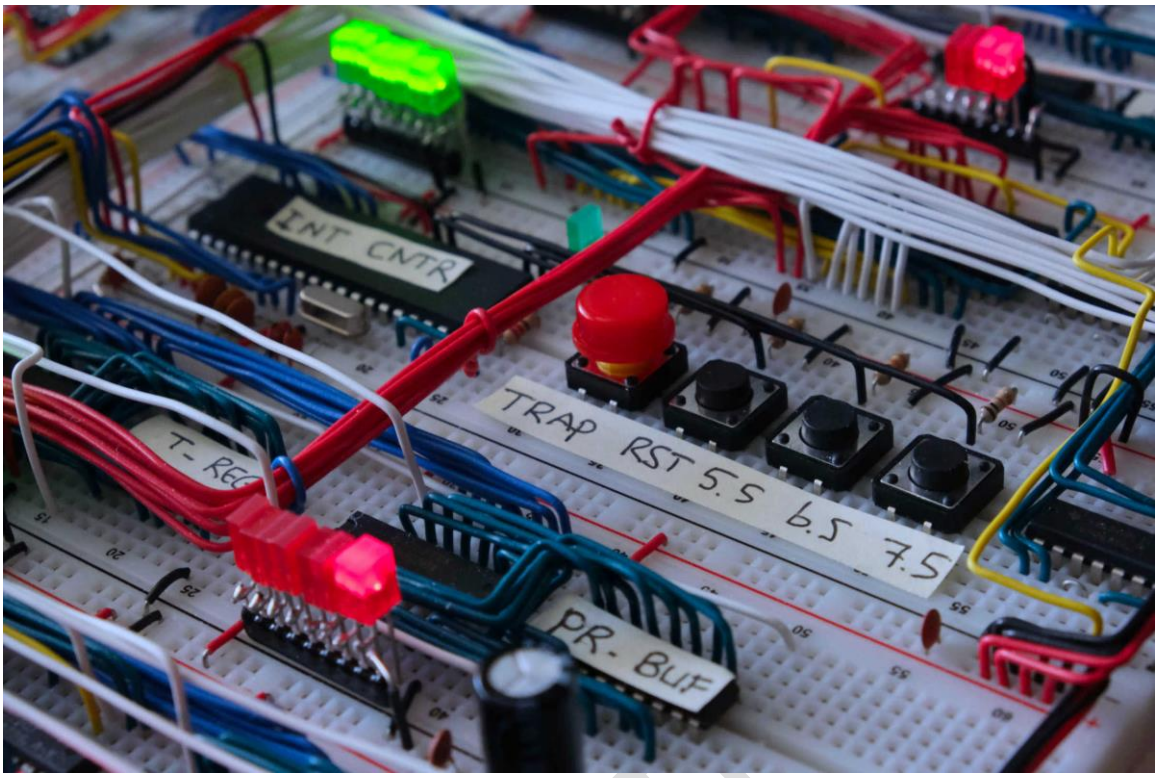
To give an idea:

- Initialization is almost 90 instructions,
- Clearing the graphical and text address area another 30 instructions. There is no instruction like on the HD44780, used for the 2*16 LCD screen, to clear the display. You have to write some code for that.
- Writing the above message 150 instructions, this however was done inefficiently. I want to write a subroutine to do this more efficient.

And obviously I had to write some instruction to control the display, like:

- LLCD_C *value*, write *value* as command to the large LCD
- LLCD_P *value*, write *value* as parameter to the large LCD

An interrupt controller like the 8085



Picture of the Interrupt controler

The final “pièce de résistance”, building an interrupt controller like it’s implemented in the 8085. I first tried using an Arduino Nano since they fit nicely on the breadboards, but I ended up in finding out that I did not have enough I/O. I could have used a shift register to create more I/O but that would consume too much space. I also needed to place the four buttons for the interrupt control lines somewhere. And on the far right is an IC for building the zero flag of the ALU. That left me little options, I have quite a range of MCU’s and I have the EasyPIC v4 and V7 MCU development kit. So it was kind of logical to use an MCU. I know the 16F877A quit well, having it used on other electronic projects. It’s a quite versatile device.

What does an interrupt controller do?

Imagine you have the following program:

10	instruction
20	etc
30	Scan emergency button and if pressed; do something really urgent
40	etc
…	etc
500	Conditional CALL “do something else”
510	etc
…	etc
600	Back to 10

The program runs instructions from 10 to 600 and sometimes executes the CALL “do something else”. In the loop at instruction 30 it checks if the emergency button is pressed, and if yes, it switches on a big red warning sign saying “STOP RIGHT NOW !”

But, the program does this emergency button check only when it is at instruction 30. If the emergency button is pressed when the program is at instruction 50, then the check has to wait until the program has gone to all instruction up to 600, go back to 10 and then at 30 the emergency button is checked. This obviously takes time, this takes even longer if the program executes the CALL “do something else”. Most likely you don’t have this much time to wait. Upon pressing the emergency button, you want the sign “STOP RIGHT NOW !” to turn on DIRECTLY. Otherwise it might come way too late.

Here comes an interrupt controller in handy, what does it do. An interrupt controller has inputs, and when these are activated the interrupt controller takes over control of the computer.
Imagine having an interrupt controller and at instruction 50 they emergency button is pressed.

The following happens:

- The interrupt controller waits until instruction 50 is completed,
- Then the interrupt controller steps in, and takes over control,
- Now the interrupt controller start execution a program somewhere else in memory,
- This program could be: switch on the sign “STOP RIGHT NOW !”
- After this the program continues execution at instruction 60.

So now with the interrupt controller no matter were the main program is in execution, the big red sign “STOP RIGHT NOW !” is directly activated if the emergency button is pushed.

Interrupt vectors and the interrupt service routine

An interrupt vector is a memory address were the program counter jumps to upon activation of an interrupt input. On my 8 bit computer:

- If TRAP is activated the PC and PR jump to 0x0006
- If RST5.5 is activated the PC and PR jump to 0x0021
- If RST6.5 is activated the PC and PR jump to 0x0036
- If RST7.5 is activated the PC and PR jump to 0x0051

An ISR, interrupt service routine is just a piece of software which gets executed at activation of an interrupt. So on my computer if TRAP is pushed the PR and PC jump to 0x0006. On 0x0006 I hard programmed in ROM a jump to 0x8100. At location 0x8100, RAM, I can program an ISR with whatever program I would like. Like turn on a big red warning sign saying: “STOP RIGHT NOW !”

To summarize

- If TRAP is activated the ISR at 0x8100 is executed
- If RST5.5 is activated the ISR at 0x8200 is executed
- If RST6.5 is activated the ISR at 0x8300 is executed
- If RST7.5 is activated the ISR at 0x8400 is executed

Interrupt control on the 8085 microprocessor

I tried to build part of the interrupt facilities as provided on the 8085 microprocessor of the 80’s, yes it’s an ancient device. The 8085 has hardware and software interrupts, I only tried to build the hardware interrupt. The 8085 has the following hardware interrupts:

- TRAP
- RST5.5
- RST6.5
- RST7.5
- INTR

I did not built the INTR functionality. Next to be discussed is interrupt enabling/disabling and masking.

For RST5.5, RST6.5 and RST7.5 the interrupt functionality has to be activated with the software EI command, EI stands for enable interrupt. There is also the software command DI, disable interrupt. So RST5.5, RST6.5 and RST7.5 interrupt facility can be turned on or off. With TRAP this cannot be done, it's always active.

Next there is the possibility by setting an interrupt mask, this switches on or off the individual interrupt facility of RST5.5, RST6.5 and RST7.5.

How does the interrupt controller work I built.

The microcontroller scans the TRAP, RST5.5, RST6.5, RST7.5 and the scr (step counter reset) inputs. If one of the interrupt inputs is active and a running instruction reaches the scr step the microcontroller halts execution of the computer. (and of course depending on EI/DI and SIM but that does not change the principle) It does this by raising the CS signal of the control register EEPROM's high. The same mechanism I use to access the RAM to write programs via the Arduino Nano in RAM.

The microcontroller now takes over control of the computer. The microcontroller writes the PUSH_PRPC instruction into the instruction register. This is a new instruction I made and it uses a part of the CALL instruction. In specific the part where the page register and program counter are saved on the stack.
The control then goes back to the control register EEPROM's and the PUSH_PRPC instruction is executed. When that's done, the return address were I need to go back to after the ISR has run is saved on the stack. When reaching the SCR step in the PUSH_PRPC instruction the microcontroller steps in and takes over control again.

Next depending on which interrupt was active the relevant interrupt vector is written into the page register and program counter. Then the microcontroller gives control back to the EEPROM's and the interrupt specific ISR is run. At the end of the ISR the RET instruction is executed and the program proceeds happily were it was interrupted.

That's roughly it, obviously I also need to PUSH/PULL the content of the PSW and A, B and C register to/from the stack. And I had to resolve a couple of nasty details.

It's a bit cheating using a microcontroller but as said required space on the breadboard was a serious consideration. I had only 2 breadboards left and one is intended for something else, eliminating the permanent OE low on the EEPROM's. That left me with 1 breadboard on which already one chip was mounted for the ALU zero flag. Then 4 pushbuttons. That leaves room for 2 TTL chips, or 1 EEPROM and 1 TTL chip. I don't see how you could implement an interrupt controller with that little hardware. The microcontroller has over 200 lines of code, so do that with 74HCxxx logic.

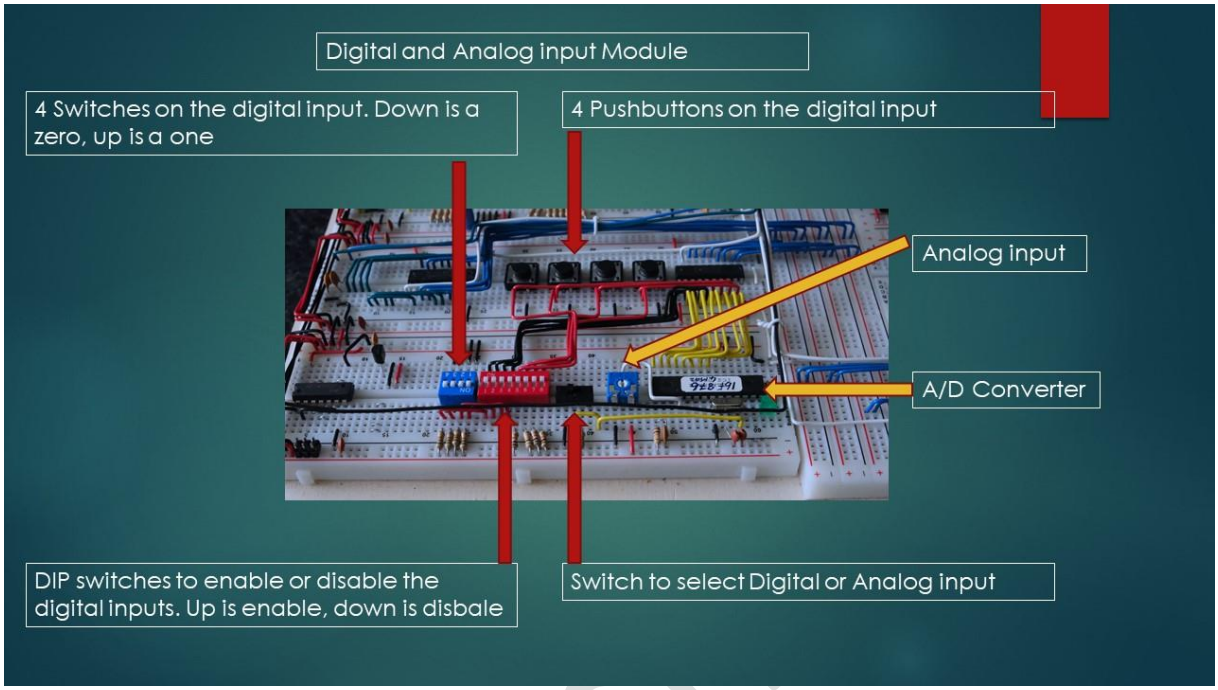
Software wise I implemented the EI, DI and flag masking instructions.

Below is a link to a video showing the working interrupt controller:

https://www.reddit.com/r/beneater/comments/g5m6qu/fully_functional_interrupt_controller/

Adding a more flexible input module.

I have three output mechanisms, the HEX display, the 2*16 LCD and the 320*240 LCD screen but only 4 input buttons. This module gives a bit more flexibility. Better would be to have more control signals and have the digital and analog inputs seperatly available.



I added a more flexible input module. It can handle 8 digital inputs or 1 analog input. With the center black selection switch analog or digital is chosen. The red DIP switch block enables or disables the pushbutons and switches. For analog mode the switches must be down for digital up. When up, it connect the 4 resistors to Vcc for the pushbuttons. With the blue switches a digital input can be set high or low. This for instance is comfortable during start up. The 320*240 LCD display takes a long time to initialize, but on start up I select digital mode and the red block DIP switches all up and the left most switch on the blue DIP switch block is tested at startup and determines if I run or scip the initialization of the 320*240 LCD screen.

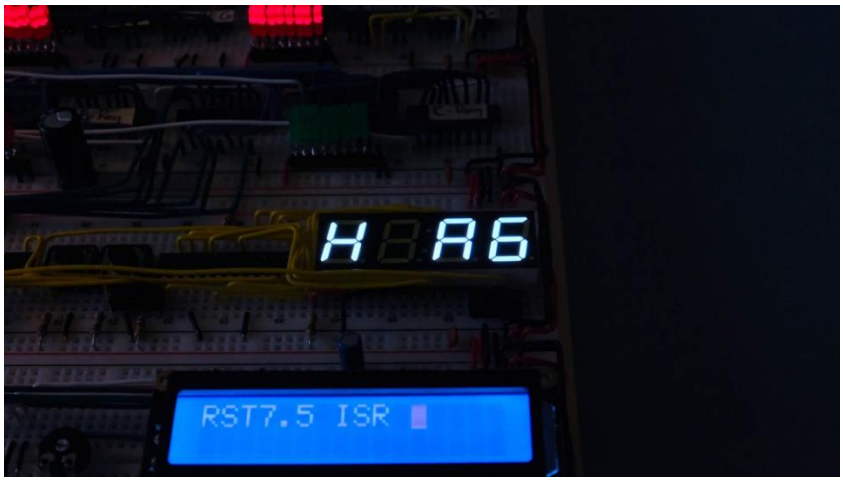
Somehow separating the analog from the digital input signals would be a nice feature.

Storing the SP-H and SP-L, creating a D and E register

I created the instructions ST_SPH, ST-SPL, LD_SPH, LD_SPL this give the possibility to use these registers as D and E register as long as I don't use the CALL and/or the PUSH and POP instruction. Before executing these instructions I have to set back the SP, buth that's a minor effort.

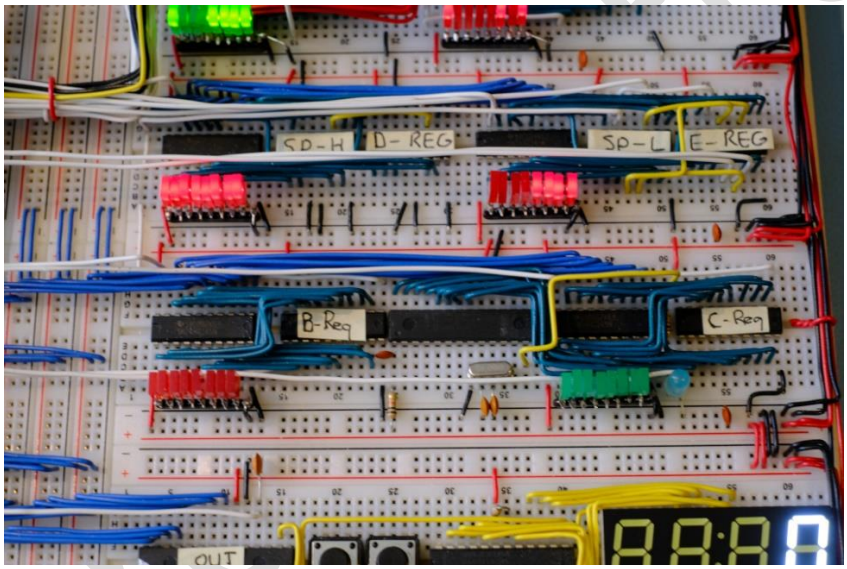
I created instruction like LDD and STE, similar to LDA and STA and more instruction for move immediates on the new D and E register. I also created move instructions between the new D and E register and the A, B and C register. Obviously also here I need to take care when to use the B and C register separate or when I use them as pointer for indirect addressing. But with the new ST_SPH, ST-SPL, LD_SPH, LD_SPL instructions I have more programming flexibility.

Hexadecimal indication on the 7-segment display



Basically this is done by modifying the code in the 16F874 microcontroller which drives the MAX7219CNG. I added a switch under the 7-segement display (little difficult to see) for selection of HEX or decimal digit display.

Analog output on the C-register



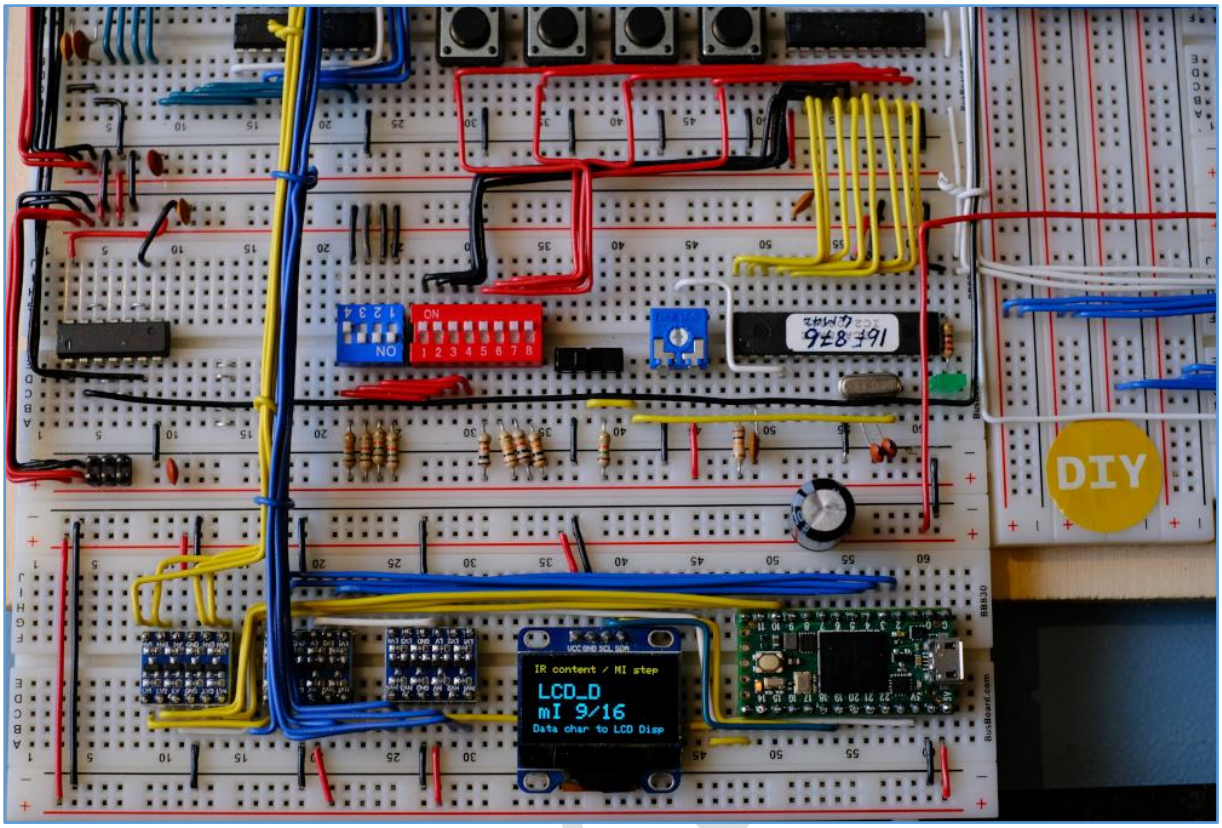
I added an analog output which is connected to the C-regsiter. Basically it converts the 8 bits of the C register to an analog signal. The blue LED is an indication for the voltage level on the analog output.

Again I used a microcontroller, a 16F874, just because I have many of them and I have the EasyPIC platform. Obviously a DA converter can be built with other components like a DAC8080.

The microcontroller uses a 4 MHz crystal.
Following link shows the AO connected to a vintage class room analog meter.

https://www.reddit.com/r/beneater/comments/gtdr9x/i_already_had_an_analog_input_today_i_added_a/

MNEMONICS Display



The MNEMONICS display, displays the content of the instruction register and the step counter on an OLED display.

I used the following components:

- a Teensy 4.0
- a 128*62 I2C OLED screen
- three voltage converters to convert 5 volt into 3.3 volt signals. The Teensy cannot handle 5.0 volt, this will damage the microcontroller.

In my build all LED's are followed by a current limiting resistor to GND. I connected the top of the current limiting resistor to the Teensy. I think the voltage converters can be removed but I didn't want to take the risk to blow up the Teensy at start-up of the computer. The Teensy is a very powerfull microcontroller but not cheap, it's much more expensive as an Arduino Navo. But it's so much faster, it can run at 600 MHz and you can overclock the device, with cooling, to 1 GHz. I would need to measure to which voltage levels the spikes reach at start-up.

The instruction register and step counter are hooked up to the Teensy. The OLED display only has power and SDA and SCL for the I2C protocol. The wiring is hooked up as shown on below screen dump of the Teensy code.
The Teensy needs the correct voltage level as power supply. The power level on the breadboard above the one for the MNEMONICS display was a little low. I had to run a separate power supply for the 2*916 LCD, since this also had issues. So the power for the MNEMONICS display comes from the LCD breadboard.

```
// Digital input signals from the Instruction Register
#define IR_0     9     // Instruction Register - LSB
#define IR_1     8
#define IR_2     7
#define IR_3     6
#define IR_4     5
#define IR_5     4
#define IR_6     3
#define IR_7     2     //Instruction Register - MSB

// Digital input signals from the Step Counter
#define step_1   12    //Step Counter - LSB
#define step_2   11    //MSB-2
#define step_3   10    //MSB-1
#define step_4   13    //Step Counter - MSB

/* Connect Teensy to +5 Volt and GND
 * Teensy inputs shall not be connected to +5 Volt
*/

// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
// The pins for I2C are defined by the Wire-library.
// On an arduino UNO:              A4(SDA), A5(SCL)
// On an arduino MEGA 2560:        20(SDA), 21(SCL)
// On an arduino LEONARDO:         2(SDA),  3(SCL),
// On an Teensy 4.0:               18(SDA), 19(SCL),
```

The code in the Tensy is basically simple:

- scan the inputs
- have a buch of if statements for selection and based on that write something to the screen

Example:

```
if(instr_no==0b00000000)

{ MNEMONIC="LDA";

  write_MNEMONIC(MNEMONIC, step_no);

 write_MNEMONIC_desc("Load Accumulator");

}
```

Having the instruction as MNEMONICS is practical since the LED's are not very handy to quickly identify which instruction is in the instruction register. Displaying the step counter is more a nice to have, the four LED's of the step counter can easily be interpreted.

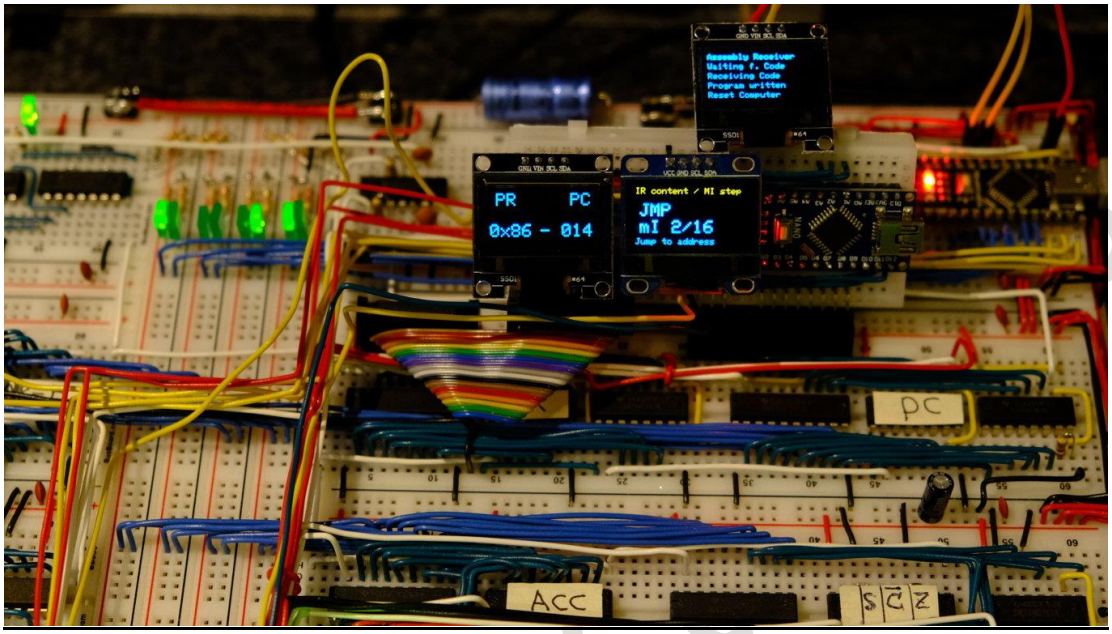
The display can be extended in many ways. In case the clock is on a slow speed, you could make a rolling display giving a more detailed explanation of the instruction.

Even more nice would be to make it a hardware emulator, but that's quite a bit of work. Actually on the original 8 bit form Ben Eater I think you can program it in a weekend with sufficient pizza and beer supply, but I have too many instructions to do this quickly.

Using CUSOMASM for programming in Assembly

The top OLED display, shows the interface to the CUSTOMASM program enabling programming in assembly.

A short video on how this roughly works is on the reddit channel:
https://www.reddit.com/r/beneater/comments/nmai8e/this_is_a_video_on_how_i_program_my_breadboard/



With CUSTOMASM you can write programs in assembly. Assembly makes it possible to write instructions like: Load a, 64. This would load the value 64 in the A register.

Assembly programs are useually written in a text editor, after which they are assembled to machine specific instructions by an assembler. Obviously this is more practical than programming the computer via switches or send numerical codes via an Arduino or use the Arduino IDE as programming tool for the RAM.

There are many assemblers each fo a specific CPU, like for instance for an 8085 or x86, the advantage of CUSTOMASM is that you can define your own instruction set. On top CUSTOMASM has many practical functions, like using labels for loops or define constants and it's even possible to use functons.

Basically a program is made and loaded to the computer in the following manner:

- in a texteditor, I use Notepad++, a program is written,
  A file specifying the machine specific instructions has to be included,
- CUSTOMASM is used to assemble the text from Notepad++ to hexadecimal instructions,
  Load a, 64 would translate to 0x00 and 0x40.
  One number for the instruction and one for the value.
- CUSTOMASM produces a .bin file with all the machine specific codes,
  Assembling has to take place using the annotated option, this is the only option which places the software addresses in the .bin file.
- Python is used to customize the .bin file and upload it to the Aduino Nano. This customizing is mainly removing text, before it can be send.
- The Arduino Nano programs the RAM.

It's also possible to program the ROM, I added a ROM programming enable switch, to avoid unwanted programming of the ROM.

Details on CUSTOMASM can be found on: https://github.com/hlorenzi/customasm
There is a good help file and examples on how to use it and how to define your own CPU.

The Python program transfers the CUSTOMASM .bin file into the following format;

#PR #PC #Instr #second byte #third byte

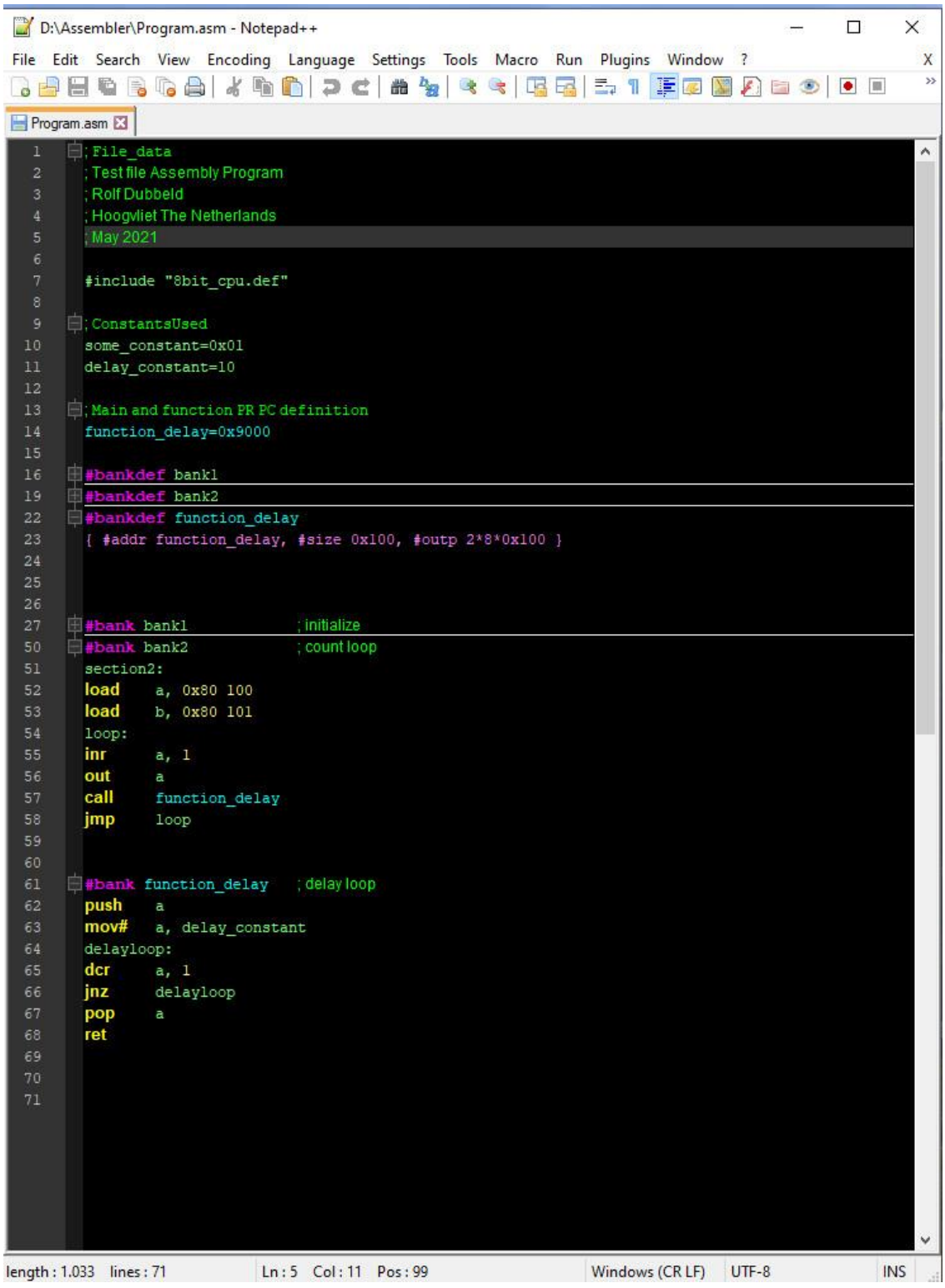
PR is the page register. PC is the program counter. My instructions are always three bytes, which I might change, the first byte being the instructions and the other two specifying instruction specific data. This file is send to the Arduino Nano via the USB link.

The Arduino Nano hardware is defined in the following manner in my built, it's pretty straight forward:

```
// Control Signals for communication to the 8 bit computer
#define Data_A0          2     // LSB of the databus
#define Data_A           9     // MSBof the databus
#define mar_l            10    // To mar-l of the memory adrres register
#define mar_h            11    // To mar-h of the memory adrres register
#define ro               12    // To ro of the RAM
#define RI               13    // To RI of the RAM
#define CLK              A0    // Clock Signal Input
#define ROM_PROG         A1    // Push Button, enabling ROM programming
#define EEPROM_CE        A2    // To CE line Control EEPROM's
#define EEPROM_OE        A3    // To OE ROM Memory, work in progress
//                       A4    used for the OLED display
//                       A5    used for the OLED display
#define EEPROM_WE        A6    // To WE ROM Memory work in progress
```

For this setup it's required that all control lines from the control EEPROM are connected to +5 volt or ground via a resisitor, I used 2K7. Active low signals are connected to +5 volt and active high signals are connected to GND. When the CE of the control EEPROM's is made high than the Arduino Nano can take over the control of the memory section, allowing to write a program.

Example of a program in assembly



The statement #include “8bit_cpu.def” defines all the instruction.

delayloop is an example of how labels can be used. This avoids to have to use specific addresses in instruction which have to be modified if the program is modified. Labels can even be defined with a local scope.

Chapter 13, Possible Improvements & Extensions

I see the following improvements and lessons learned points:

- The use of 28C256 EEPROM's for the control logic and leaving OE low is a bad design which I countered with the pull-up and pull-down resistors. It helps but in essence it's a design flaw. So either use another type of EEPROM or change the timing were OE is used which would require more breadboard space, which I don't have anymore.

- The use of demultiplexers on the EEPROM data output lines. I realized way too late that many combinations never happen. The A, B, C, the temporary register and the ALU will never write at the same time to the bus. So a 3 to 8 demultiplexer will save 5 control lines. This would free up 15 control lines, or something like that. With these lines I could have built the DE and HL registers. Which would make it a functional 8085. This would require at least two more breadboards for which I now have no place. In order to do this effectively I would have to re-group and re-wire the control lines. This is the most ineffective part of my whole 8 bit breadboard build, and unfortunately it's not easy to fix.

- Above results in looking for possibilities to extend the number of breadboards. I tried adding a piece of plywood under the existing one, it makes it really chunky. Best option would be to straight away go for a 38*60 cm plywood backplane, but that's in hindsight. In doing this I can add four breadboards basically for the DE and HL register pair. Which would leave 2 breadboards free for I/O. Next limit would be the size of my workbench, larger than 38*60 is just not feasable.

- Use the memory in the open memory space. The ROM is using 8K starting at 0x0000, while RAM starts at 0x8000. leaving 24k non addressed memory space. For this I would also need more breadboard space.

- Power rails on the left and right side, and a separate rail for the clock signal.

- 16 bit wide bus, make a couple of things a lot easier and the computer quicker. Or Harvard architecture. But this makes the build a lot larger. I also have no intention to build a product.

- All above improvements require a serious amount of rework and hardware modifications.

What I also found out is that changing from an 8 bit PC to a 16 bits PC makes many things more complex.

As example, I studied the build from someone else who had an 8 bit address, he only needs a 3 bit step counter I need 4 bits. The 8 bit PC can do a CALL with 8 steps, the 16 bit needs 14 or 15 steps. You have double the wiring for the PC, MAR and address lines to the memory chips and the stack pointer.

The advantage is obviously having way more RAM enabling writing larger programs like resolving A^3 + B^3 + B^3 = K. Where A, B,C are integers and K a positive integer <= 100. A memory of 256 bytes looks nice but is quickly exhausted with somewhat larger program or using look-up tables.

This is a list of possible software extensions without modifying the hardware as described above:

(color green indicated that this has been done, it's my check list)

- Optimizing the microinstructions. It would also increase the speed of the computer,
- Test all instructions once more to make sure all instructions are 100% correct. This is a serious amount of work, but several errors found and corrected. Still work in progress.
- Create normal jump instuctions relative to the current page. This reduces updating the MAR-H and increases the speed of the computer,
- Make specific instruction for SP-H and SP-L to use as D and E register. For more programming flexibility. Create instruction like Store SP Load SP. Same for BC register pair. Maximize programming efficiency for not having a DE and HL register pair,
- Add more basic functionality in the ROM including double and floating point numbers. Find a proper solution for time critical loops which can handle manual mode, slow, fast and crystal oscillator clock speeds,
- Improve ADDC and SUBB instructions,
- Complete and test the missing conditional CALL and returns,
- Create conditional jump instuctions relative to the current page,
- Create variable instruction lengths, all instructions are now fixed at three bytes, which makes programming in the Arduino IDE relative simple. This will require the use of an assembler. This will also increase the speed of the computer.

This is a list of minor hardware modifications:

- Improve reliability, still work in progress
- Optimize maximum clock frequency for running and programming mode both RAM and ROM, experiment (again) with bus termination to improve maximum clock speed,
- Make a qwerty keyboard entry, I would need to mount another Arduino Nano for which I have no space. So this will be difficult.
- Adding a more flexible input module. I have three output mechanisms, the HEX display, the 2*16 LCD and the 320*240 LCD screen but only 4 input buttons, make range 0-5 volt by separate supply line or some other scaling,
- Check why the LCD blinks. Added dedicated power supply wiring for the LCD screen
- Improve the mounting of the pull-up and pull-down resistors on control EEPROM data output lines,
- Add an analog output. Connected to the C register, remove the 74HC377 inbetween B and C register for the HEX display. Increase 16F876 HEX display clock speed, now 16 MHz. Make selection on HEX display for indication as 0xhh,
- Use the 320*240 output for other purposes, forget about it. Makes no sense
- Relocate the 74HC00 for the step counter reset, this gives more possibilities to use the 320*240 port for something else. Not possible, no space available.
- Find a better use for the DIP switches for manual address and data entry, these are not used anymore. Can it be used as input module. So far no good ideas.
- Found a bug, programming by means of the DIP switches does not work correctly anymore. The boot ROM jumps to page 0x85. The DIP switches are programmed for access to page 0x80.
- Suddenly the stack pointer does not work correctly.
- Make a power-on reset, makes no sense without no program loaded

List of other things:

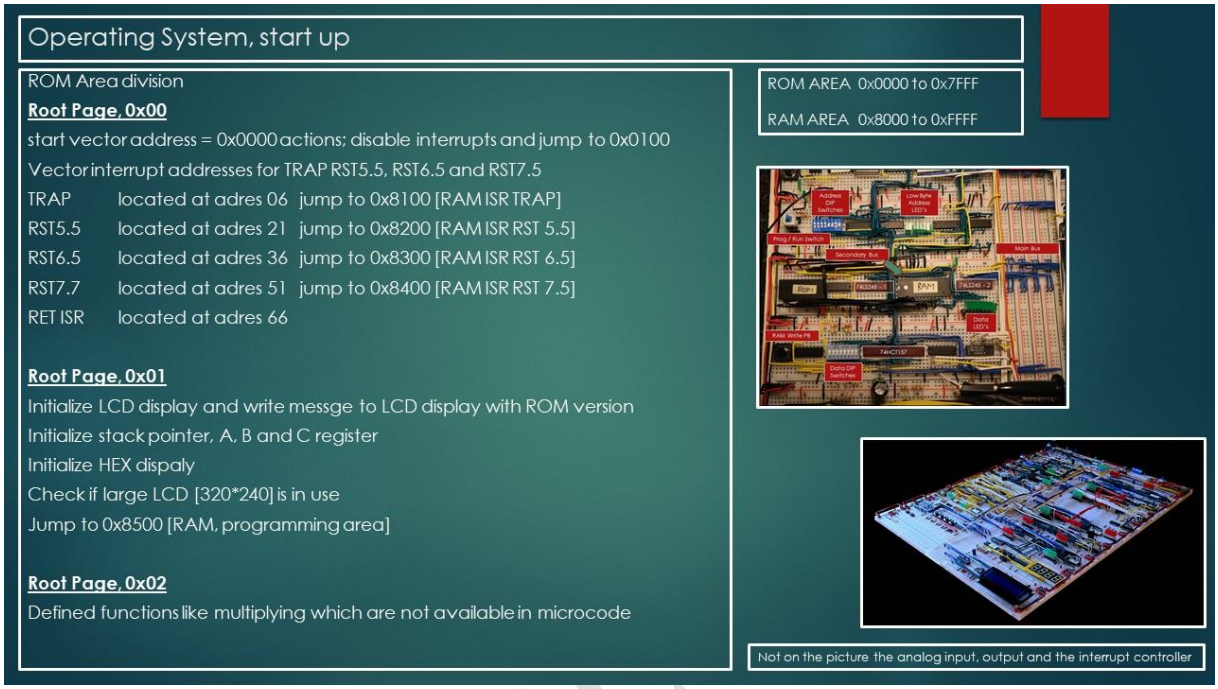
- Keep documenting the SAP-3+ build, use this document.
- Make all drawings in KiCAD,
- Last but not least, having it all on a PCB, but it's a staggering amount of work.

The list of interesting extensions without serious hardware modifications is much larger than the list of extensions with serious hardware mods, it also requires more creativity and thinking. Conclusion make it better not bigger.

Chapter 14, Operating Manual

This is a must because with an extensive build like this, if you have not worked with it for a couple of month's you forget how it works.

Build up of the ROM section, start-up and interrupt vectors and ISR's.



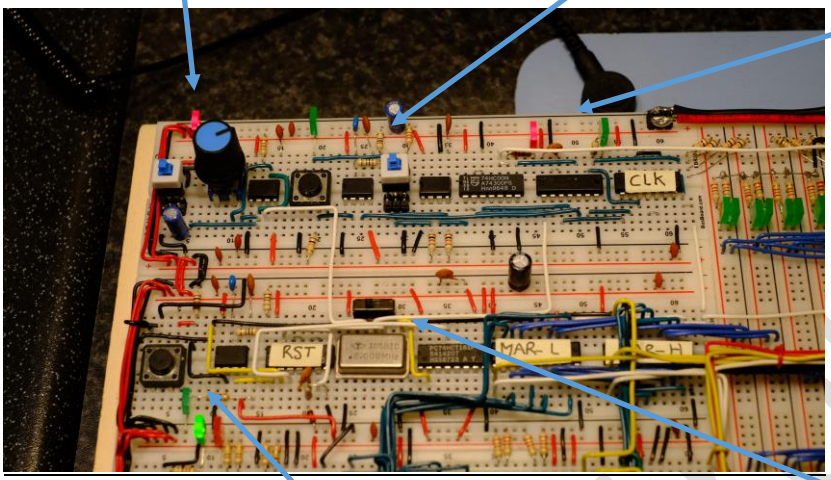
RAM Area division
Page 0x80 addresses space reserved for variables in programs, address from 0 to address <200
Page 0x80 addresses space reserved for variables in ROM functionality, address>=200 and <240
0x80-240 fixed address for saving SP-H
0x80-241 fixed address for saving SP-L
0x80-242 fixed address for saving A-register
0x80-243 fixed address for saving B-register
0x80-244 fixed address for saving C-register
0x80-245 fixed address for saving D-register
0x80-246 fixed address for saving E-register

page 0x81 TRAP ISR Jump
page 0x82 RST 5.5 ISR Jump
page 0x83 RST 6.5 ISR Jump
page 0x84 RST 7.5 ISR Jump
page 0x85 and above Custom Programming Area

Description of all the buttons

Clock Module

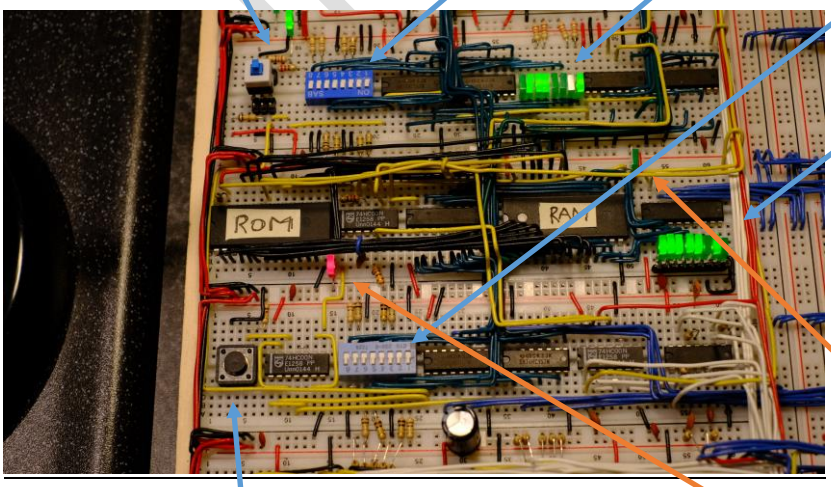
Switch SW1 Pushed in Higher Clock Speed, red LED lights up Pushed Out lower Clock speed Potentiometer, clock speed	Switch SW2 Pushed in automatic mode Pushed out manual mode, green LED lights up Pushbutton PB1, single step per microinstruction	Red LED, computer halted Green LED, clock signal



Reset pushbutton PB2, green LED lights up when pushed	Clock signal selection switch SW3 Left is NE555 Right is crystal oscillator

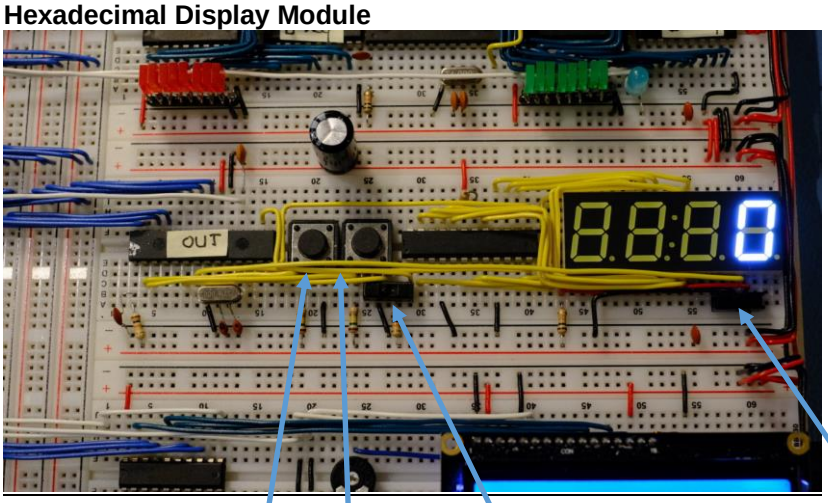
Memory Module RAM/ROM

Switch SW4 Pushed in run mode, green LED lights up Pushed Out programming mode, red LED lights up	Adress LED's and DIP switches	Data LED 's and DIP switches



Write pushbutton PB3, write data into RAM	Red LED, ROM accessed Green LED, RAM accessed

Hexadecimal Display Module



Brightness Pushbuttons PB 4 and 5 Left dimmer Right brighter	Switch SW5 1 Complement is left position 2 Complement is right position	Switch SW6 Left position is decimal Right position is hexadecimal

Commands for the LCD Display

LCD Initialization

0b00111000	LCD Init Function set, 8 bit/2 Lines/5*8 dot char
0b00001111	LCD Init Display on/off control, Display on/Cursor on/Cursor blink
0b00000110	LCD Init,Entry mode set, Increment adress by one/No scroll
0b00000001	LCD Clear Display
0b000001ab	LCD Entry mode a=1 Write character to the right a=0 Write character to the left b=0 No Scroll 1=Scroll
0b00001dcb	Display Control d=0 display off 1=display on c=0 cursor off 1=cursor on b=0 blink off 1= blink on
0b0001 s/c r/l xx	Cursor or display shift, use this to scroll display 0b0001 11 xx, shift all to the right Each instruction gives 1 shift s/c=0 cursor move s/c=1 display shift r/l =0 shift to left r/l =1 shift to right
0b10000000	Goto first line 128 + 0. Char on place 10 = 0b10000000 + 9, counting starts from 0
0b10101000	Goto second line 128 + 40

Commands for the 320*240 LCD Display

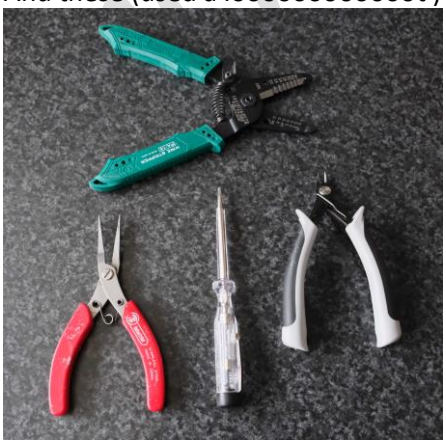
Chapter 15, KiCAD drawings

All drawings I might be making will be stored on my Github account
https://github.com/rolf-electronics/The-8-bit-SAP-3

Chapter 16, Equipment used

Siglent 1202X-E	Scope (I have this one now over a year, damn good scope)
Siglent SDG 1032X	AWG
Siglent SPD3303X-E	Power supply
Unitrend UNI-T UT804	Bench top Mulitmeter
YitIUA 8786D	Soldering station
Zero Plus LogicCube	Logic Analyser
Elenco LP-560	Logic probe
EasyPIC V4 and V7	MCU development kit

And these (used a looooooooooot)



A screwdriver comes in handy to get IC's from the board without damaging pins

And a couple of boxes with all kind of electronics components in a variety of values, very handy

Chapter 17, List of used Components

(to be detailed later)

Chapter 18, References

Ben Eater Video's on Youtube

A.P.Malvino, digital Computer Electronics

http://buildyourcomputer.org/aGuideToComputerScience.pdf.

An introduction to microprocessors from D.K.Kaushik

RC2014.co.uk (don't ask I have one)

EEVblog,