Minimal UART CPU System 1.x

by Carsten Herting (slu4)

last update Jul 24th 2022

Manual Rev. 1.x

Welcome to the Minimal CPU System - let's just call it the 'Minimal'. I've designed this little computer entirely from TTL logic to be as enjoyable and educational as I possibly can. It's made as a learning platform and to facilitate a deep understanding of the basic principles of computers. Despite being deliberately simple, this CPU is powerful enough to "never stop being usable" and more than doubles the processing power of a Commodore C64 or Apple II. It can run some serious software including early video game classics like TETRIS, IEEE 32-bit floating point math, a text editor, a native assembler capable of assembling itself and even a Python-like high-level programming language.

This document provides a comprehensive step-by-step introduction to first-time users of the Minimal and at the same time serves as a reference and programming handbook. See the section 'CPU Architecture' for a brief explanation of how it all works.

Have fun and let me know what you think!

Board Revisions

There are several revisions of the 'Minimal' publicly available:

Revision 1.2 (EEPROM PCB Edition)

- 1.8432MHz clock speed
- 8KB ROM, 24KB RAM
- memory-mapped UART I/O
- not software-compatible to higher revisions

Revision 1.3 (FLASH PCB Edition)
- o 1.8432MHz clock speed
- o 32KB FLASH SSD, 32KB RAM
- o dedicated UART I/O instructions
- o fully software-compatible to revision 1.5

Revision 1.5 (Expanded PCB Edition)
- o selectable clock speed up to 3.6864MHz
- o 512KB FLASH SSD, 32KB RAM
- o dedicated UART I/O instructions
- o expansion port and selectable clock speed

Redux Breadboard Revision 1.6 (aka Beast Mode Edition)

same as revision 1.5 except for:
- o 8.3MHz maximum clock speed
- o improved microcode efficiency
- o runs all software of revision 1.3 - 1.5
- o optimized for breadboards and lowest chip count

Earlier breadboard prototypes no longer exist (revisions 0.9 and 1.0). PCB revisions 1.1 and 1.4 represent internal validation steps.

This text applies to board revisions 1.5 and higher, although from 1.3 onwards, all revisions are software-compatible. A cycle-exact emulator of revision 1.5 is available on Windows (see chapter 'Emulator').

Building the Hardware

Step 1. I have documented the development of the 'Minimal CPU System' on my YouTube channel:
www.youtube.com/channel/UCXYQcMpUBT3aaQKfmAVJNow
- Humble beginnings:
www.youtube.com/playlist?list=PLYlQj5cfIcBVRMsr9yxHmvCzMqonI6O6N
- 'Minimal' is taking shape:
www.youtube.com/playlist?list=PLYlQj5cfIcBU5SqFe6Uz4Q31_6VZyZ8h5
- Build information on revision 1.5 (Expanded Edition):
www.youtube.com/watch?v=osVi06VKvA0
- Build information on revision 1.6 (Breadboard Redux Edition):
https://www.youtube.com/watch?v=Gz1VVOsNn_8

There is also a discussion board where you can engage with other 'minimalists' to get some help or browse through different builds:
https://minimal-cpu-system.boards.net/

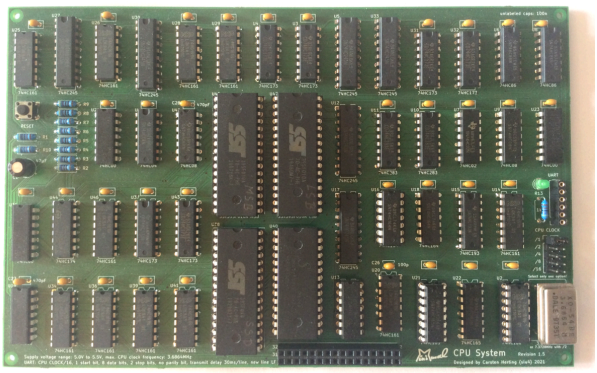
All build information is contained in the Minimal's GitHub repository:
https://github.com/slu4coder/Minimal-UART-CPU-System
This is what you'll find:

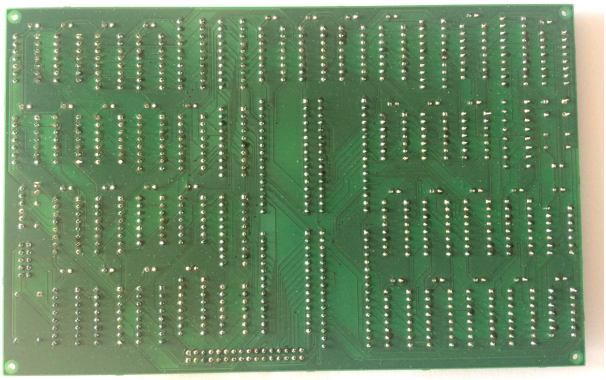
- KiCAD project files with detailed schematics and PCB layout
- PCB Gerber files and bill of materials (BOM)
- Source code of all programs available for the Minimal
  These programs can be assembled and uploaded either on real hardware or into the 'Minimal Emulator'.
- Binary images of the FLASH memory (OS, demos and games) and the CPU's control microcode
- Cross-assembler executable (Windows) to write your own programs
- Cross-assembler written in Python
  www.youtube.com/watch?v=rdKX9hzA2lU
- Emulator executable simulating the Minimal cycle-exactly

Step 2. Download the GitHub repository. If you want to go for the PCB version, send the Gerber files over to a PCB manufacturer of your liking. Going for the breadboard version you will need 8 good-quality 65-row breadboards as shown below (point-to-point resistance <1 Ohm is recommended), around 240 jumper wires plus some longer connections and lots of smaller bits and pieces of wire.

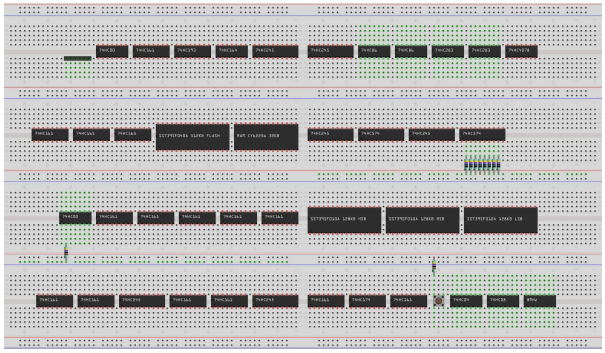
Step 3. Shop for all the parts needed (see bill of materials). Depending on the region you live in, this may be either very easy or quite complicated. Almost all parts are pretty standard, though.

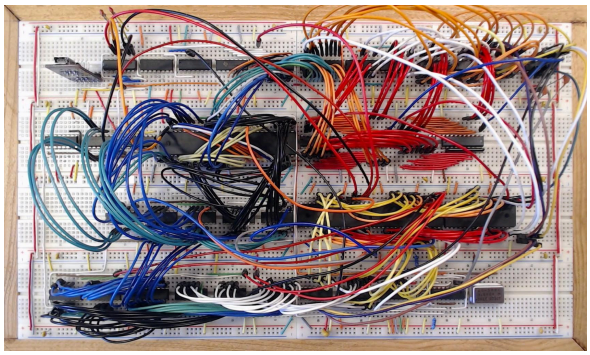
Step 4. Assembly! It usually takes about 2-3 hours of soldering 😜.
Since I have deliberately used old-school through-hole parts and DIL IC packages, soldering can easily be done by hand.
The 74HCxx IC family is susceptible to ESD damage, so it's a good idea to ensure proper grounding of your workspace before you start. The result on PCB should look something like this:



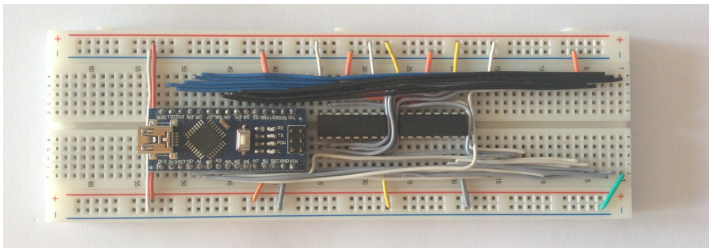


If you go for the breadboard version, plan around 6 hours of build time. Any mistake will be a pain to track down. So carefully double-check every connection.

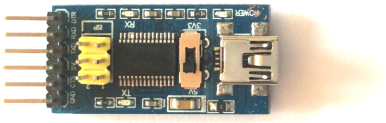




Step 5. Burn the FLASH images (Control FLASHs and the OS/SSD FLASH). If you do not own a FLASH EEPROM programmer I recommend building my DIY version: www.youtube.com/watch?v=2crXqNlBazg



Step 6. Get a 5V output USB-to-serial breakout board. Not all boards share the same pinout. My board here is based upon the IC FT232. From bottom to top the pins are labeled GND, CTS, 5V, TXD, RXD and DTR (not used).



Step 7. Read the section ‘Before Power-Up’ of this document before connecting the PCB to your PC.

Before Power-Up

Select the Minimal's clock speed divider (applies to revision 1.5 only) by placing a jumper horizontally on exactly one of the rows of the 5x2 pin header on the bottom right of the PCB. If you are using a 3.6864MHz crystal you can use any jumper position. If you are using a 7.3728MHz crystal the smallest divide you can select is 2, since the

maximum CPU clock rate is 3.6864MHz. Let's for now choose a clock speed of 1.8432MHz and call this the default speed.

5x2 Divider		3.6864MHz Crystal	7.3728MHz Crystal
/1	|O O|	3.6864MHz (230400bps)	do not connect
/2	|O O|	1.8432MHz (115200bps)	3.6864MHz (230400bps)
/4	|O O|	0.9216MHz (57600bps)	1.8432MHz (115200bps)
/8	|O O|	0.4608MHz (28800bps)	0.9216MHz (57600bps)
/16	|O O|	0.2304MHz (14400bps)	0.4608MHz (28800bps)

Other oscillator and clock speed combinations are also possible, e. g. a 16MHz oscillator with a 2MHz (/8) system clock selected will run at a serial speed of 125kbps. You can even use a manual clock. I recommend socketing the oscillator for maximum flexibility.

WARNING: "Hot-plugging" the Minimal or ICs of the Minimal can cause unwanted write operations to the FLASH IC, potentially leading to data loss, making a re-programming of the FLASH IC necessary.

Always disconnect your USB-to-serial breakout board from the USB port before plugging it into the Minimal's UART socket. Make sure that the GND and 5V lines connect correctly. Verify that the UART TXD (transmit) line will cross-connect to the breakout board's RXD (receive) line and vice versa. Only then connect the USB plug to power (usually a USB port on your PC).
*If you plan to connect to the Minimal UART socket with jumper wires, always make sure to connect GND *before* connecting any other lines.*

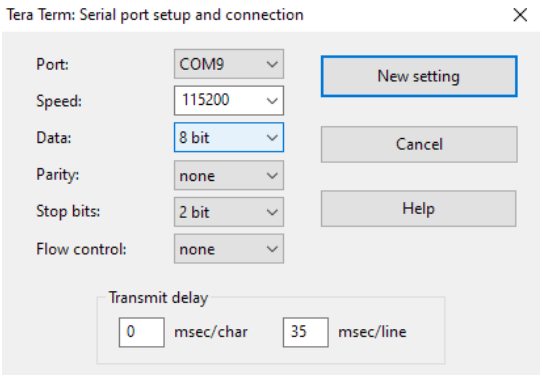
Serial Port Configuration

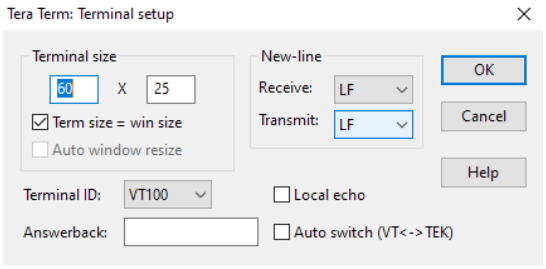
After power-up, the LED should be on. Now is the time to configure the serial port of your PC. This configuration will look a bit different in each terminal emulation but generally we need to set the following:

- Set the baud rate to 1/16th of the CPU clock speed you have selected. Since we have chosen 1.8432MHz, this is 115200bps.
- Select 1 start bit, 8 data bits, 2 stop bits, no parity bits, no flow control, local echo off.
- Set the 'new line character' to be LF (0x0a) like in UNIX.
- Set a transmit delay of 35ms/line to ensure that Minimal will have enough time to process each input line. The delay varies with clock speed (3.6864MHz: 17ms, 230.4kHz: 95ms) and also with your host system's load. Just experiment a bit.
- Set the terminal emulation to 60 x 25 characters and chose a

proper font. I like this C64-style TrueType terminal font: https://style64.org/c64-truetype.

In 'TeraTerm' on my Windows machine, this will look like this:





NOTE: With a minor modification the Minimal also supports RTS/CTS flow control. See the appendix for more information.

Boot Monitor

Now press the RESET button in the upper left corner of the PCB. The Minimal should greet you with the following start screen:

```
+----------------------------------------+
| MINIMAL CPU SYSTEM 1.5.3 by C. Herting |
| 512KB SSD - 32KB RAM - Type m for menu |
+----------------------------------------+
8000 _
```

You are now inside the boot monitor. Typing 'm <ENTER>' displays the menu options:

```
HEX [r] Set A [run]
[A].B   Show [A]..B [q]
:C[ D]  Store C[D] at A..
v A B C Fill A..B with C
k A B C Copy A..B to C..
i A   DisAsm A.. [q]
s A B F Save A..B as file F
l file  Load file
z file  Zap file
n 0..f  Set SSD bank
t       Show SSD content
w       Wipe SSD bank
```

Let's go through them by using examples. By typing any 2-byte HEX address followed by <ENTER> you can change the address the monitor is displaying at the start of the input line. The OS only accepts lower-case letters.
By pressing 'r <ENTER>' the monitor will jump to that location and execute whatever program is located there. Let's try that by typing:

```
f000 <ENTER>
r <ENTER>
```

You should again see the start screen since 0xf000 is the start of the operating system in RAM. Bring back the menu with m. You can display the memory content by using the '.' symbol:

```
.f0ff <ENTER>
```

will display the memory content starting from the current address (which we have set to 0xf000) until 0xf0ff. You can take a look at larger sections, too. After each page, the OS waits for a keystroke before displaying the next page. You can quit by pressing 'q'. Try

```
0000.ffff <ENTER>
```

to display the whole memory content starting at address 0x0000. Let's display the memory area at 0x8000 by typing:

```
8000.801f <ENTER>
```

This should show something similar (but not identical) to this:

```
8000  ab 9f e6 6a 28 f7 b6 0a  65 ae 8e f5 73 43 d7 b2
8010  d2 dd ab c6 8f d7 26 f3  8f 0e 08 62 9f 2d c0 ee
8000
```

since after power-up the RAM is usually filled with random garbage. Let's fill the first 16 bytes with zeros by using the 'fill' command 'v':

```
v 8000 800f 0 <ENTER>
8000.801f <ENTER>
```

which will now display

```
8000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
8010  d2 dd ab c6 8f d7 26 f3  8f 0e 08 62 9f 2d c0 ee
```

```
8000
```

You can fill larger memory areas, too. Just be a bit careful until you know the memory layout. There is a copy command ‘k’ as well. Let’s copy the second row starting from 0x8010 to the first row starting at 0x8000 by typing

```
k 8010 801f 8000 <ENTER>
8000.801f <ENTER>
```

which will output the result

```
8000  d2 dd ab c6 8f d7 26 f3  8f 0e 08 62 9f 2d c0 ee
8010  d2 dd ab c6 8f d7 26 f3  8f 0e 08 62 9f 2d c0 ee
8000
```

as expected. Let’s try to disassemble something. Since the only program in memory right now is the OS itself at 0xf000, let’s type

```
i f000 <ENTER>
```

which shows us

```
f000 JPA f015
f003 JPA f03c
f006 JPA f327
f009 JPA f347
f00c JPA f376
f00f JPA f8bb
f012 JPA f91b
f015 LDI fe
f017 STA ffff
```

and so on. Press ‘q’ if you have seen enough. Until now, we haven’t actually changed any memory content. Let’s store some data by using the ‘:’ command:

```
8000: 0 1 2 3 4 5 6 7 8 9 a b c d e f <ENTER>
```

Taking a look at that memory again with ‘8000.800f <ENTER>’ shows

```
8000  00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f
8000
```

We can use that feature to input mnemonics, too. Try the following:

```
8000: INP BEQ 00 80 OUT JPA 00 80 <ENTER>
8000 r <ENTER>
```

Congratulations! You have just written your first assembler program on the Minimal. It is reading the UART, printing out any keystrokes immediately to the screen. Try typing something!

```
Hello World! ajshkjdahskdjhaskdjhaskd
```

Okay, we can't get back to the monitor since our program is running in an endless loop. Simply press RESET. You don't have to enter longer programs by hand of course! Just copy and paste the following assembled program into your serial terminal:

```
8000
:0e fe 16 ff ff 0e 13 35 fd 0e 80
:35 fc 38 22 80 14 05 80 48 65 6c
:6c 6f 2c 20 57 6f 72 6c 64 21 0a
:00 34 ff 16 44 80 34 fe 16 45 80
:1d 44 80 11 00 3b 43 80 02 00 00
:00 00 00 00 00 00 2e 44 80 14 2c
:80 39 00 00
8000
```

and enter 'r' for run and you should see this:

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

You can use this method to upload any (even very large) programs from your host computer into the RAM of the Minimal.

Memory Layout

The address space of the Minimal is 16 bits wide and reaches 64KB at 0x0000-0xffff. This address space is expanded to 512KB of FLASH memory, which can be accessed in 16 chunks of 32KB called banks. The active bank is controlled by the 4-bit bank register.

```
0x0000 - 0x7fff   32KB of active FLASH SSD bank
0x0000 - 0x0033   Bank 0: OS boot loader and dummy file header
```

```
0x0034 - 0x0fff   Bank 0: OS image, gets copied to RAM upon RESET
0x1000 - 0x7fff   Bank 0: free
0x8000 - 0xffff   32KB of RAM
0xe000 - 0xefff   used temporarily by the SSD file system
0xf000 - 0xfeff   MinOS operating system
0xfeb0 - 0xfec8   MinOS variables
0xfec9 - 0xfeff   MinOS line input buffer
0xff00 - 0xffff   256 bytes of CPU stack
0xffff            LSB of stack pointer (SP)
```

On bank 0 the first 4KB (0x0000-0x0fff) hold a write-protected and self-loading image of the operating system MinOS. The rest of bank 0 and all other SSD banks can be used for your programs and data.

Upon pressing RESET, the program counter and bank register are set to zero. The code starting at 0x0000 on bank 0 - the OS bootloader if you will - copies the OS image into the RAM area 0xf000-0xfeff and then jumps to 0xf000 into the boot monitor which in turn presents its start screen.

Please also keep in mind that several of Minimal's native software development tools, e. g. the text editor and assembler, make use of additional memory areas. See section 'Native Tool Chain' for more information.

SSD File System

Although it is possible to access the FLASH memory area directly with read and write operations as detailed in the datasheet of the SST39SF040 FLASH, the operating system features a minimalistic file system that transforms the FLASH EEPROM into a viable SSD drive which greatly facilitates storing and retrieving your data by offering the following basic functionality:

SAVE

To save a block of memory to the currently selected SSD bank, type:

```
s <firstaddr> <lastaddr> <filename> ENTER
```

<firstaddr> and <lastaddr> denote the memory hex addresses of the first and last byte to store and <filename> can be a string with a maximum length of 19 characters. The start address <first> of the data is stored as part of the file as detailed in the section 'file format'.

LOAD

To load a saved block of memory back into RAM, just type:

```
l <filename> ENTER
```

Note that the operating system only searches for the specified filename within the currently selected SSD bank.

ZAP (DELETE)

In order to delete an existing file, type:

```
z <filename> ENTER
```

In case there exist two or more files of the same filename, it is always the oldest file that is deleted.

TABLE OF CONTENT (DIRECTORY)

To show the content of the currently selected SSD bank, type:

```
t ENTER
```

WIPE (FORMAT)

In case you want to format the currently selected SSD bank, type:

```
w ENTER
```

All data stored on this SSD bank will be erased.

File Format

Data is stored as a 'file' by prepending a file header containing the following additional information to the data:

20 bytes	<filename>	zero-terminated string
2 bytes	<address>	destination address of <data>
2 bytes	<N>	byte size of <data>
N bytes	<data>	data section

Properties

Within one SSD bank, it is possible to have multiple files with identical names. The file system acts like an upward-growing stack where the last-written file resides on the top of the stack. Deletion on the other hand always searches for the *oldest* occurrence of a given filename and only deletes this version.

When a file is deleted from the SSD, other files at higher addresses are moved down within the active SSD bank to close the gap. To

accomplish this, the file system uses the RAM area 0xe000-0xefff as a temporary data buffer.

Assembler Programming

Writing your programs “close to the metal” will yield the fastest code. And it really helps develop a deep understanding of the inner workings of a CPU. Depositing byte values or mnemonics at memory locations - as we have seen above - already allows you to input short programs. It is much more convenient however, to let an assembler translate your program into machine code.

Currently there are two cross-platform assemblers and even a native assembler available for the Minimal. The use of the native assembler is described in the section ‘Native Tool Chain’. It supports the same syntax as the cross-platform versions.
Both cross-assemblers are simple command line tools running on your host PC that let you specify a filename of your source code and that output machine code to the console. That code will be in a format matching the ‘set address’ and ‘deposit’ commands of the OS that you can then cut & paste to the Minimal via a terminal emulation. The Minimal will “think” that you are just typing very quickly.

Assembler Syntax

The assembler ‘asm.exe’ is written in C++ and runs on Windows whereas ‘asm.py’ runs platform-independently in any Python interpreter. To assemble a file, just type

asm <file> or python asm.py <file>.

The supported basic syntax is identical for both assembers but ‘asm.exe’ offers an additional #include functionality and a command line option for printing out symbol tables. Type asm -h or asm --h or asm for more information. The following statements must be placed at the start of a line:

```
#org 0x80ff      sets the program counter address to 0x80ff
#include file   includes a file prior to assembling (asm.exe only)
#begin           begins emitting the opcode (default)
#end             ends emitting opcode (but the PC is still advanced)
label:           defines a label as the start address of a line
```

The following statements may be placed within a line in any order:

```
MNEMONIC        emits the opcode of a mnemonic written in upper-case
label           emits the LSB and MSB of the address label
<label          least significant byte (LSB) of address label
>label          most significant byte (MSB) of address label
label+off       emits the LSB and MSB of the address label+off
label-off       emits the LSB and MSB of the address label-off
                <off> can be any decimal number between 0 and 99.
0x8fff          16-bit hex word (will be decoded to LSB MSB)
0x8f            8-bit hex byte (negative numbers in 2's complement)
-123 or 123     8-bit signed decimal byte
'a'             equivalent to 65 or 0x41
'hello'         byte string in memory (0x68 0x65 0x6c 0x6c 0x6f)
1,2,3 or 1 2 3  defines multiple bytes in memory
; blablabla     comment
```

Labels and mnemonics are case-sensitive. Mnemonics are only accepted in upper-case. Use lower-case for hex numbers. Mathematical expressions and definitions of constants beyond the limited functionality described above are not supported.

Instruction Set Overview

Here you find a list of the available instruction types and their different address modes. Please note that not every address mode is available for every instruction type.

Legend: dark gray = not applicable, light gray = not implemented

Description	Accumulator No Operand	Accumulator Immediate	Accumulator Abs Address	Accumulator Rel Address	Byte at Abs Address	Word at Abs Address	Stack at SP + Offset
Load A from		LDI	LDA	LDR			LDS
Store A to			STA	STR			STS
Clear					CLB	CLW	
Negate	NEG				NEB	NEW	
Increment	INC				INB	INW	
Decrement	DEC				DEB	DEW	
Add		ADI	ADA	ADR	ADB	ADW	
Subtract		SBI	SBA	SBR	SBB	SBW	
Compare		CPI	CPA	CPR			
Add with Carry In		ACI	ACA	ACR	ACB	ACW	
Subtract with Carry In		SCI	SCA	SCR	SCB	SCW	

Jump to			JPA	JPR			
Jump to Subroutine			JPS				
Return from Subroutine	RTS						
No Operation / Wait	NOP						
Terminal Input	INP						
Set FLASH bank *	BNK						
Output A to Terminal	OUT						
Clear Carry In Flag	CLC						
Set Carry In Flag	SEC		Description		Accumulator No Operand	Accumulator Immediate	Accumulator Abs Address
Logical Shift Left	LSL						
Rotate Shift Left	ROL		Branch on Non-Zero				BNE
Logical Shift Right	LSR		Branch on Zero				BEQ
Rotate Shift Right	ROR		Branch on Carry Clear				BCC
Arithmetic Shift Right	ASR		Branch on Carry Set				BCS
Push on Stack	PHS		Branch on Plus				BPL
Pull from Stack	PLS		Branch on Minus				BMI

* has no effect on revision 1.3 (32KB FLASH Edition)

Instruction Set 1.3 - 1.5

Software written for revision 1.3 - 1.5 will run on higher revisions.

Legend: A=accumulator, R=result, M=most significant byte, ?=undefined, -=unchanged

Instruction			Description	Target	Operand		Accumulator	Flags			Clock
Name	DEC	HEX			Type	Size	Change	N	C	Z	Cycles
NOP	0	**0**	No Operation	none	none	0	-	-	-	-	16
BNK	1	**1**	Set FLASH bank *	bank	A	0	-	-	-	-	4
OUT	2	**2**	UART Output	UART	A	0	-	1	0	0	4
CLC	3	**3**	Clear Carry In Flag	none	none	0	-	1	0	0	5
SEC	4	**4**	Set Carry In Flag	none	none	0	-	0	1	1	5
LSL	5	**5**	Logical Shift Left (=ASL)	A	none	0	R	R	R	R	5
ROL	6	**6**	Rotate Shift Left	A	none	0	R	R	R	R	5
LSR	7	**7**	Logical Shift Right	A	none	0	R	R	R	R	13
ROR	8	**8**	Rotate Shift Right	A	none	0	R	R	R	R	12
ASR	9	**9**	Arithmetic Shift Right	A	none	0	R	R	R	R	15
INP	10	**0A**	UART Input incl. CPI 0xff	A	none	0	R	R	R	R	6
NEG	11	**0B**	Negate	A	none	0	R	R	?	R	6
INC	12	**0C**	Increment	A	none	0	R	R	R	R	5
DEC	13	**0D**	Decrement	A	none	0	R	R	R	R	5

LDI	14	**0E**	Load from	A	immediate	1	R	-	-	-	4
ADI	15	**0F**	Add	A	immediate	1	R	R	R	R	5
SBI	16	**10**	Subtract	A	immediate	1	R	R	R	R	5
CPI	17	**11**	Compare	A	immediate	1	-	R	R	R	5
ACI	18	**12**	Add with Carry In	A	immediate	1	R	R	R	R	5
SCI	19	**13**	Subtract with Carry In	A	immediate	1	R	R	R	R	5
JPA	20	**14**	Jump to	PC	abs addr	2	-	-	-	-	6
LDA	21	**15**	Load from	A	abs addr	2	R	-	-	-	7
STA	22	**16**	Store A to	byte @	abs addr	2	-	-	-	-	8
ADA	23	**17**	Add	A	abs addr	2	R	R	R	R	8
SBA	24	**18**	Subtract	A	abs addr	2	R	R	R	R	8
CPA	25	**19**	Compare	A	abs addr	2	-	R	R	R	8
ACA	26	**1A**	Add with Carry In	A	abs addr	2	R	R	R	R	8
SCA	27	**1B**	Subtract with Carry In	A	abs addr	2	R	R	R	R	8
JPR	28	**1C**	Jump to	PC	rel addr	2	-	-	-	-	9
LDR	29	**1D**	Load from	A	rel addr	2	R	-	-	-	10
STR	30	**1E**	Store A to	byte @	rel addr	2	-	-	-	-	10
ADR	31	**1F**	Add	A	rel addr	2	R	R	R	R	11
SBR	32	**20**	Subtract	A	rel addr	2	R	R	R	R	11
CPR	33	**21**	Compare	A	rel addr	2	-	R	R	R	11
ACR	34	**22**	Add with Carry In	A	rel addr	2	R	R	R	R	11
SCR	35	**23**	Subtract with Carry In	A	rel addr	2	R	R	R	R	11
CLB	36	**24**	Clear	byte @	abs addr	2	R	0	1	0	8
NEB	37	**25**	Negate	byte @	abs addr	2	R	R	?	R	10
INB	38	**26**	Increment	byte @	abs addr	2	R	R	R	R	10
DEB	39	**27**	Decrement	byte @	abs addr	2	R	R	R	R	10
ADB	40	**28**	Add	byte @	abs addr	2	-	R	R	R	9
SBB	41	**29**	Subtract	byte @	abs addr	2	-	R	R	R	10
ACB	42	**2A**	Add with Carry In	byte @	abs addr	2	-	R	R	R	11
SCB	43	**2B**	Subtract with Carry In	byte @	abs addr	2	-	R	R	R	11
CLW	44	**2C**	Clear	word @	abs addr	2	-	0	1	0	10
NEW	45	**2D**	Negate	word @	abs addr	2	?	M	?	M	13
INW	46	**2E**	Increment	word @	abs addr	2	?	M	M	M	13
DEW	47	**2F**	Decrement	word @	abs addr	2	?	M	M	M	13
ADW	48	**30**	Add	word @	abs addr	2	?	M	M	M	12
SBW	49	**31**	Subtract	word @	abs addr	2	?	M	M	M	13
ACW	50	**32**	Add with Carry In	word @	abs addr	2	?	M	M	M	13

SCW	51	**33**	Subtract with Carry In	word @	abs addr	2	?	M	M	M	14
LDS	52	**34**	Load from Stack	A	offset	1	R	?	?	?	9
STS	53	**35**	Store A on Stack	stack	offset	1	-	?	?	?	16
PHS	54	**36**	Push on Stack	stack	none	0	-	?	?	?	12
PLS	55	**37**	Pull from Stack	A	none	0	R	?	?	?	10
JPS	56	**38**	Jump to Subroutine	PC	abs addr	2	?	?	?	?	16
RTS	57	**39**	Return from Subroutine	PC	none	0	?	?	?	?	14
BNE	58	**3A**	Branch on Non-Zero	PC	abs addr	2	-	-	-	-	5/6**
BEQ	59	**3B**	Branch on Zero	PC	abs addr	2	-	-	-	-	5/6**
BCC	60	**3C**	Branch on Carry Clear	PC	abs addr	2	-	-	-	-	5/6**
BCS	61	**3D**	Branch on Carry Set	PC	abs addr	2	-	-	-	-	5/6**
BPL	62	**3E**	Branch on Plus	PC	abs addr	2	-	-	-	-	5/6**
BMI	63	**3F**	Branch on Minus	PC	abs addr	2	-	-	-	-	5/6**

* has no effect on board revisions prior to 1.5
** 6 cycles if branching

Instruction Set 1.6

This table shows the detailed properties of each instruction of revision 1.6. Improvements with respect to revision 1.3 - 1.5 are small but noticeable and are highlighted below in cyan. Software explicitly exploiting these changes will not run on earlier revisions.

Legend: A=accumulator, R=result, M=most significant byte, ?=undefined, -=unchanged

Instruction			Description	Target	Operand		Accumulator	Flags			Clock
Name	DEC	HEX			Type	Size	Change	N	C	Z	Cycles
NOP	0	**0**	No Operation	none	none	0	-	-	-	-	16
BNK	1	**1**	Set FLASH bank *	bank	A	0	-	-	-	-	4
OUT	2	**2**	UART Output	UART	A	0	-	1	0	0	4
CLC	3	**3**	Clear Carry In Flag	none	none	0	-	1	0	0	5
SEC	4	**4**	Set Carry In Flag	none	none	0	-	0	1	1	5
LSL	5	**5**	Logical Shift Left (=ASL)	A	none	0	R	R	R	R	5
ROL	6	**6**	Rotate Shift Left	A	none	0	R	R	R	R	5
LSR	7	**7**	Logical Shift Right	A	none	0	R	R	R	R	13
ROR	8	**8**	Rotate Shift Right	A	none	0	R	R	R	R	12
ASR	9	**9**	Arithmetic Shift Right	A	none	0	R	R	R	R	15
INP	10	**0A**	UART Input and CPI 0xff	A	none	0	R	R	R	R	6
NEG	11	**0B**	Negate	A	none	0	R	R	?	R	6
INC	12	**0C**	Increment	A	none	0	R	R	R	R	5
DEC	13	**0D**	Decrement	A	none	0	R	R	R	R	5

LDI	14	**0E**	Load from	A	immediate	1	R	-	-	-	4
ADI	15	**0F**	Add	A	immediate	1	R	R	R	R	5
SBI	16	**10**	Subtract	A	immediate	1	R	R	R	R	5
CPI	17	**11**	Compare	A	immediate	1	-	R	R	R	5
ACI	18	**12**	Add with Carry In	A	immediate	1	R	R	R	R	5
SCI	19	**13**	Subtract with Carry In	A	immediate	1	R	R	R	R	5
JPA	20	**14**	Jump to	PC	abs addr	2	-	-	-	-	5
LDA	21	**15**	Load from	A	abs addr	2	R	-	-	-	7
STA	22	**16**	Store A to	byte @	abs addr	2	-	-	-	-	7
ADA	23	**17**	Add	A	abs addr	2	R	R	R	R	8
SBA	24	**18**	Subtract	A	abs addr	2	R	R	R	R	8
CPA	25	**19**	Compare	A	abs addr	2	-	R	R	R	8
ACA	26	**1A**	Add with Carry In	A	abs addr	2	R	R	R	R	8
SCA	27	**1B**	Subtract with Carry In	A	abs addr	2	R	R	R	R	8
JPR	28	**1C**	Jump to	PC	rel addr	2	-	-	-	-	8
LDR	29	**1D**	Load from	A	rel addr	2	R	-	-	-	10
STR	30	**1E**	Store A to	byte @	rel addr	2	-	-	-	-	10
ADR	31	**1F**	Add	A	rel addr	2	R	R	R	R	11
SBR	32	**20**	Subtract	A	rel addr	2	R	R	R	R	11
CPR	33	**21**	Compare	A	rel addr	2	-	R	R	R	11
ACR	34	**22**	Add with Carry In	A	rel addr	2	R	R	R	R	11
SCR	35	**23**	Subtract with Carry In	A	rel addr	2	R	R	R	R	11
CLB	36	**24**	Clear	byte @	abs addr	2	-	-	-	-	8
NEB	37	**25**	Negate	byte @	abs addr	2	R	R	?	R	10
INB	38	**26**	Increment	byte @	abs addr	2	R	R	R	R	10
DEB	39	**27**	Decrement	byte @	abs addr	2	R	R	R	R	10
ADB	40	**28**	Add	byte @	abs addr	2	-	R	R	R	8
SBB	41	**29**	Subtract	byte @	abs addr	2	-	R	R	R	10
ACB	42	**2A**	Add with Carry In	byte @	abs addr	2	-	R	R	R	9
SCB	43	**2B**	Subtract with Carry In	byte @	abs addr	2	-	R	R	R	11
CLW	44	**2C**	Clear	word @	abs addr	2	-	-	-	-	10
NEW	45	**2D**	Negate	word @	abs addr	2	?	M	?	M	13
INW	46	**2E**	Increment	word @	abs addr	2	?	M	M	M	13
DEW	47	**2F**	Decrement	word @	abs addr	2	?	M	M	M	13
ADW	48	**30**	Add	word @	abs addr	2	?	M	M	M	12
SBW	49	**31**	Subtract	word @	abs addr	2	?	M	M	M	13
ACW	50	**32**	Add with Carry In	word @	abs addr	2	?	M	M	M	13

SCW	51	**33**	Subtract with Carry In	word @	abs addr	2	?	M	M	M	14
LDS	52	**34**	Load from Stack	A	offset	1	R	-	-	-	8
STS	53	**35**	Store A on Stack	stack	offset	1	-	-	-	-	15
PHS	54	**36**	Push on Stack	stack	none	0	-	-	-	-	11
PLS	55	**37**	Pull from Stack	A	none	0	R	-	-	-	9
JPS	56	**38**	Jump to Subroutine	PC	abs addr	2	?	-	-	-	14
RTS	57	**39**	Return from Subroutine	PC	none	0	-	-	-	-	12
BNE	58	**3A**	Branch on Non-Zero	PC	abs addr	2	-	-	-	-	5
BEQ	59	**3B**	Branch on Zero	PC	abs addr	2	-	-	-	-	5
BCC	60	**3C**	Branch on Carry Clear	PC	abs addr	2	-	-	-	-	5
BCS	61	**3D**	Branch on Carry Set	PC	abs addr	2	-	-	-	-	5
BPL	62	**3E**	Branch on Plus	PC	abs addr	2	-	-	-	-	5
BMI	63	**3F**	Branch on Minus	PC	abs addr	2	-	-	-	-	5

* has no effect on board revisions prior to 1.5

Addressing Modes

Where applicable the last letter of the mnemonic indicates the addressing mode. The CPU expects 16-bit addresses in little-endian format, i. e. the least significant byte (LSB) at the lower memory address and the most significant byte (MSB) at the higher address.

‘A’ indicates absolute addressing mode. The following two bytes are interpreted as an address of the argument of the instruction. The result is stored in the accumulator. Example: ADA 0x04 0xf0 adds the content of address 0xf004 to the accumulator.

‘R’ indicates relative addressing mode. The address following the opcode is interpreted as a pointer to the address of the argument. The result is stored in the accumulator. Example: ADR 0x04 0xf0 adds the content of the address stored at 0xf004 to the accumulator.

‘B’ indicates ‘to byte at absolute address’ mode. Here, the accumulator is the argument and the result is stored in the byte specified by the address following the opcode. Example: LDI 10 ADB 0x04 0xf0 adds 10 to the content of 0xf004.

‘W’ indicates ‘to word at absolute address’ mode. Again the accumulator provides the argument but the result is stored in the word at the address following the opcode. Example: LDI 250 ADW 0x04 0xf0 adds 250 to the content of 0xf004. In case of an overflow (carry flag set) the content of 0xf005 automatically increments.

UART Operations

Using the UART as an I/O device is really easy and only involves the following two instructions:

OUT sends the content of the accumulator via UART. Please keep in mind that the Minimal (being minimal) does not feature an output buffer. So it is up to you to wait for the UART transmission to complete before you can send another byte. The shortest possible interval between two consecutive OUT instructions is 160 cycles or 10 NOP instructions.

INP moves and clears the content of the UART receiver register into the accumulator. An empty register will yield 0xff and the zero flag is set (Z=1). Thus, INP provides non-blocking input. Blocking input can be implemented with:

```
Wait:     INP BEQ Wait
```

Stack Operations

The memory address 0xffff holds the LSB of the CPU stack pointer (the MSB is always 0xff). The stack pointer is pointing to the next free memory location on the stack page 0xff00-0xffff and is growing downwards. The stack pointer has to be initialized with

```
LDI 0xfe STA 0xffff
```

which is usually done by the OS. If you plan on using your own software, please keep in mind to include this line in your code. The Minimal offers the following instruction that make use of the stack:

JPS <subroutine> pushes the LSB and MSB of the address following the opcode (program counter + 1) to the stack and decrements the stack pointer twice. Next the processor loads the address <subroutine> into the program counter, entering your subroutine.

RTS increments the stack pointer twice and pulls two bytes from the stack and stores them in the program counter. Before the execution is continued, the program counter is incremented twice, effectively pointing it to the next instruction that follows the JPS <subroutine> call.

PHS pushes the content of the accumulator onto the stack and decrements the stack pointer.

PLS increments the stack pointer and pulls the value stored at that stack position into the accumulator.

LDS <index> loads the content of the address (stack pointer + <index>) into the accumulator. <index> can be any signed byte value.

STS <index> stores the content of the accumulator to the address (stack pointer + <index>). <index> can be any signed byte value.

Calling Convention

Although it is really up to you, the programmer, how you want to handle data transfer to and from subroutines within your code, it can be helpful to stick to a certain use pattern which for the Minimal I will outline here. Let's suppose, a subroutine expects two parameters <A> and <B> to work with and returns a result <C>.

The caller then has the responsibility to push these data onto the stack prior to calling the subroutine. Once the caller has regained focus, it cleans up and in case of <C> processes all data that it has pushed onto the stack:

```
    LDI <C> PHS     ; pushes a container <C> for the return value
    LDI <B> PHS     ; pushed parameter <B>
    LDI <A> PHS     ; pushes parameter <A>
    JPS <subroutine>
    PLS             ; cleans up <A> from stack
    PLS             ; cleans up <B> from stack
    PLS             ; moves the return value <C> into the accumulator
    STA ...         ; stores the result <C> somewhere
```

The callee in turn can access parameters and store results within the stack area provided by the caller. Note that after the JPS instruction has pushed the return address below the parameters, they - to the callee - appear two bytes further up the stack.

```
<subroutine>:   LDS 5 STA ...   ; loads parameter <A> from stack
                LDS 4 STA ...   ; loads parameter <B> from stack
                ...             ; do some stuff
                STS 3           ; stores return value in container <C>
                RTS             ; caller regains focus
```

Stack as viewed from the caller's perspective prior to JPS:

```
Offset:     -4    -3   -2    -1     0     1     2     3    (relative to SP)
Stack:     ---   ---   ---   ---   ---   <C>   <B>   <A>
```

Stack as viewed from the callee's perspective right after JPS:

```
Offset:     -2    -1    0     1     2     3     4     5    (relative to SP)
Stack:     ---   ---   ---   MSB   LSB   <C>   <B>   <A>
```

API Functions

Why not reuse helpful OS subroutines for your own code? Here is how and why: OS subroutines are called via a fixed jump table address. With a new OS release, API functions may be added or improved but the addresses of existing API functions WILL NOT CHANGE over time. The programs you write with the API will work no matter what OS version you use. This will keep your own code nice and concise, and these functions are already pretty optimized. Here is an overview of the API subroutines and other useful data:

```
Address          API label        Description
-------------------------------------------------------------------------
#org 0xf000      _Start:          ; OS entry point for restart
#org 0xf003      _Prompt:         ; OS entry point for line input prompt
#org 0xf006      _Print:          ; prints a null-terminated string
#org 0xf009      _PrintHex:       ; prints a byte in HEX format
#org 0xf00c      _WaitUART:       ; waits for UART transmission
#org 0xf00f      _LoadFile:       ; loads a file from the SSD
#org 0xf012      _SaveFile:       ; saves data as a file to SSD bank
#org 0xf015      _MemMove:        ; moves an (overlapping) memory block
#org 0xf018      _FindFile:       ; returns a pointer to a stored file
#org 0xf01b      _Mnemonics:      ; pointer to mnemonic table
#org 0xf01d                       ; pointer (unused)
#org 0xf01f      _HexToWord:      ; converts a hex string into 2 bytes
#org 0xf022      _CursorX:        ; sets the cursor x position 1..60
#org 0xf025      _CursorY:        ; sets the cursor y position 1..25
#org 0xf028      _Random:         ; pseudo-random byte generator
#org 0xf02b      _XOR:            ; byte XOR(A, B)
#org 0xfeb0      _MemAddr:        ; displayed address of the OS monitor
#org 0xfeb2      _ParsePtr:       ; input parsing pointer
#org 0xfec2      _RandomState:    ; random generator state (4 bytes)
#org 0xfec6                       ; unused (3 bytes)
#org 0xfec9      _InpBuf:         ; input line buffer (55 bytes)
```

Most of the functions expect parameters to be pushed onto the stack prior to calling. Some may return values to be pulled from the stack. Please note that the caller - not the callee - is responsible for cleaning up the stack. See section 'Calling Convention' for more information.

_MemMove
Moves potentially overlapping N bytes from [A..] to [B..].
push: B_lsb, B_msb, A_lsb, A_msb, N_lsb, N_msb
Pull: #, #, #, #, #, #

_Print
Prints out a null-terminated string located at <stradr>.
push: stradr_lsb, stradr_msb
pull: #, #

_CursorX
Sets the horizontal cursor position to 1..60.
push: pos
pull: #

_CursorY
Sets the vertical cursor position to 1..25.
push: pos
pull: #

_Random
Returns a pseudo-random byte <num> 0..255.
push: #
pull: <num>

Algorithm described by EternityForest (2011)
https://www.electro-tech-online.com/threads/ultra-fast-pseudorandom-number-generator-for-8-bit.124249/

```
uint8_t random()
{
  x++;
  a = a ^ c ^ x;
  b = b + a;
  c = (c + (b >> 1)) ^ a;
  return c;
}
```

_XOR
Exclusive OR (XOR) operation.
push: A, B

pull: B, XOR(A,B)

_PrintHex
Prints out a byte value <val> in HEX format.
push: val
pull: #

_HexToWord
Parses a lower-case hex number 0000 - ffff from <address>
push: #, address_lsb, address_msb
pull: word_lsb, word_msb, status
success: status = 0x00, failure: status = 0xf0

_FindFile
Searches for <filename> stored at <nameptr>
<filename> must be terminated with either 0x00 or 0x10 (LF)
push: nameptr_lsb, nameptr_msb
pull: fileptr_msb, fileptr_lsb
success: fileptr_msb < 0x80

_LoadFile
Loads file <name> from SSD, <name> must be terminated by 0 or ENTER.
Only the first 20 bytes are accepted.
push: nameptr_lsb, nameptr_msb
pull: target_lsb, target_msb
success: target_msb >= 0x80

This example loads the file "blocks" from the SSD and runs it.

```
#org 0xa000     LDI <filename PHS     ; push pointer to filename
                LDI >filename PHS
                JPS _LoadFile         ; call API function
                PLS STA target+0      ; pull target address
                PLS STA target+1
                CPI 0x80 BCC _Start   ; check for errors
                  JPR target          ; SUCCESS => jump to target

filename:       "blocks", 0
target:         0x0000
                #end
#org 0xf000     _Start:               ; declare some API symbols
#org 0xf00f     _LoadFile:
```

_SaveFile

Saves data as a file <name> to the SSD. Data will be written from address <first> to (and including) <last>.
push: nameptr_lsb, nameptr_msb, 1st_lsb, 1st_msb, last_lsb, last_msb
pull: #, #, #, #, #, status
success: status = 1

The following example saves data to the SSD as file "testdata".

```
#org 0xa000     LDI <filename PHS
                LDI >filename PHS
                LDI 0x00 PHS          ; push 0x8000
                LDI 0x80 PHS
                LDI 0xff PHS          ; push 0x8fff
                LDI 0x8f PHS
                JPS _SaveFile         ; call API function
                LDI 5 ADB 0xffff      ; clean up stack
                PLS CPI 0 BEQ _Start  ; check for failure
                JPA _Prompt

filename:       "testdata", 0
                #end

#org 0xf000     _Start:               ; declare some API symbols
#org 0xf003     _Prompt:
#org 0xf012     _SaveFile:
```

_Mnemonics
Holds a pointer to the following ordered list of mnemonics (the list index corresponds to the instruction code):

```
'NOP', 'BNK', 'OUT', 'CLC', 'SEC', 'LSL', 'ROL', 'LSR',
'ROR', 'ASR', 'INP', 'NEG', 'INC', 'DEC', 'LDI', 'ADI',
'SBI', 'CPI', 'ACI', 'SCI', 'JPA', 'LDA', 'STA', 'ADA',
'SBA', 'CPA', 'ACA', 'SCA', 'JPR', 'LDR', 'STR', 'ADR',
'SBR', 'CPR', 'ACR', 'SCR', 'CLB', 'NEB', 'INB', 'DEB',
'ADB', 'SBB', 'ACB', 'SCB', 'CLW', 'NEW', 'INW', 'DEW',
'ADW', 'SBW', 'ACW', 'SCW', 'LDS', 'STS', 'PHS', 'PLS',
'JPS', 'RTS', 'BNE', 'BEQ', 'BCC', 'BCS', 'BPL', 'BMI'
```

_MemAddr
Current address displayed by the OS memory monitor.

_ParsePtr

A pointer to the command line one step beyond the last character processed by the operating system. This grants the called program access to command line parameters. For example, after

```
8000 r <filename> ENTER
```

the pointer will point to the space following 'r' in the above command line. The called program may then use and change the pointer to extract the <filename> data.

_RandomState
4-byte state of the pseudo-random number generator. Enhance the entropy of the generator by XOR-ing external values into these data.

_InpBuf
Start of the input buffer used by the OS. It has a size of 55 bytes with the last bytes address being 0xfeff. Since the OS resets this buffer before using it, you can use it for your programs whenever the OS is not in focus (see _ParsePtr).

Native Tool Chain

You want the full retro immersion without a host system? You can write, edit, assemble, organize and run your programs natively on the Minimal! Here is how.

Text Editor

Upload and store the 'Minimal Editor' file to your SSD under the filename 'editor'. It's 4KB in size and resides in the memory area 0x8000-0x8fff. Text data start from 0x9000, with lines separated by LF (0x0a). The end of the text is designated by EOF (0x00). After a power-up, upload and start the editor with

```
8000 l editor ENTER
8000 r ENTER          or          8000 r <filename> ENTER
```

The option shown on the right will directly load a text file from the SSD. You can immediately enter text. The following editor commands are available:

```
Ctrl q                quits the editor (file stays in memory)
Ctrl n                brings up a 'new' dialog
Ctrl l <filename>     loads a text file from SSD into memory
Ctrl s <filename>     saves the current text file to SSD
```

Ctrl a	marks the beginning of a text block
Ctrl x	cuts out a marked blocks until cursor
Ctrl c	copies a marked block until cursor
Ctrl v	pastes a copied or cut text block
Ctrl z	toggles the display of line numbers

Please note that the Minimal is just capable of providing all these text editing functions. Redrawing the whole screen after a PAGE UP/DOWN, for example, takes quite some time. PAGE UP/DOWN keystrokes come encoded as a sequence of characters. Keep in mind that if you use these keys in the REPEAT mode, the Minimal will inevitably miss some of these characters and display the remaining ones in an unwanted way. The Minimal - being a minimal machine - will miss these characters because it is busy redrawing the screen and because its serial UART does not feature input buffering.

Text File Transfer

Data can be transferred to the RAM of the Minimal by pasting it to the terminal emulation you are using (e. g. Tera Term). The Minimal will receive the data as “keystroke input”, i. e. the OS will interpret the data in hexadecimal format, so that assembler output can be directly uploaded (see section ‘Boot Monitor’ for more information).

Due to Minimal’s speed constraints, the native text editor is unable to provide this direct ‘cut & paste’ functionality for a complete source text. However, text files can directly be uploaded to the SSD file system of the Minimal. A little tool called ‘textloader’ provides this functionality. Just assemble or load it, select the SSD bank you want to save the file to and type

```
8000 r <textfile> ENTER.
```

Until you hit ESC, the ‘textloader’ program will await ASCII data you paste to the terminal window as “keystrokes” and save them in Minimal’s text file format under the filename <textfile>.

Note: By using serial RTS/CTS flow control you can also directly cut & paste text into the editor itself. See the appendix for more information.

Assembler

Upload and store the ‘Minimal Assembler’ to your SSD. It is 4KB in size and resides in the memory area 0xd000-0xdfff. For temporary data

storage the area 0xe000-0xefff is being used. The native assembler supports the same general syntax as the cross-platform version (see section ‘Assembler Programming’). This section only briefly describes the usage from within the Minimal. Invoking the assembler is possible via the OS command line with two options:

```
d000 r ENTER     ... or...     d000 r <filename> ENTER
```

The first option starts the assembler without specifying a filename. The assembler then expects to find text data of the text editor in memory starting at address 0x9000. Hence, this mode allows you to quickly switch between the text editor and the assembler without having to save your source text in between. Assembling the text data will not affect your code in any way. The assembler will use the memory area that is left beyond the text file. Therefore, use this option preferably only for shorter programs and quick tests. Check for conflicts between your specified build address and the memory usage of the assembler using the memory monitor.

The second option tells the assembler to process a source file located on the SSD in FLASH memory. This is the preferred method for larger files, since the program text itself does not have to be present in RAM. The assembler will use the entire RAM starting from address 0x8000 to build your project. Again, it’s your own responsibility to check for potential memory conflicts.

A typical assembler run will look like this:

```
d000 r hello.txt ENTER

Minimal Assembler 1.5
123.........45
c046
```

The numbers 1 to 5 indicate the current pass the assembler is performing. Pass 3 usually takes the longest and is substituting all labels with their corresponding addresses, with each dot representing a single substitution.
Once finished, the assembler hands back control to the OS. The address displayed is the first free address beyond the assembler output. This makes saving your assembled program easy with

```
s c000 c045 hello ENTER.
```

When encountering an error, the assembler halts and reports a short message together with the corresponding line number, e. g.

```
ERROR in line 021: “blabla”.
```

Min (Python-Like Language)

Min stands for ‘Min is not Python’ and is a high-level Python-like programming language that can run as an interpreter natively on the ‘Minimal CPU System’. As a matter of fact, Min runs quite slowly and is meant mainly for demonstration purposes.
Min interprets program text that is currently being edited within the native text editor. So in order to use Min, upload both the editor and the Min interpreter into memory. The editor resides at 0x8000, program text starts at 0x9000 and Min resides at 0xc000.
I have made a small video series about the inner workings of Min:
https://www.youtube.com/playlist?list=PLYlQj5cfIcBW9oNldqIPtkfaKt_rixoBc

Quick Example

Let’s start the editor by typing

```
8000 r ENTER
```

and input this Min example:

```
001|while key() == 0xff
002|  print “Hello, World!”
```

Now leave the editor by pressing ‘CTRL Q’ and start Min by typing

```
c000 r ENTER
```

You will see ‘Hello, World!’ fill your screen until you hit any key.

```
8000 c000 r
Hello, World!Hello, World!Hello, World!Hello, World!Hello,
Hello, World!Hello, World!Hello, World!Hello, World!Hello,
Hello, World!Hello, World!Hello, World!Hello, World!Hello,
Hello, World!Hello, World!Hello, World!Hello, World!Hello,
Hello, World!Hello, World!Hello, World!...
```

Min’s Syntax is almost like Python’s

The exact language definition in ‘Enhanced Backus-Naur Form (EBNF)’ is given in the appendix. If you are familiar with Python you will easily

find your way in Min. The following list of the supported features, commands and differences with respect to Python will get you started.

- Python-style indentation (two spaces)
- if-elif-else, while-break, and, not, or like in Python
- Local and global variables (unlike Python Min has static typing)
- include “<filename>” functionality (see str, int, input below)
- def-return like in Python, additional C-style ‘&’ referencing
- print <expression> like in Python but without parentheses
- len(<string>),
- key() for keyboard (UART) input polling
- rnd() for pseudo-random bytes
- str(<int>), int(<string>), input() via ‘include “std.min”’
- Strings (“hello”) and integer (16-bit) data types
- 1-dimensional arrays only
- Array slicing operator '..' replacing Python’s ':'
- 'A' replacing Python’s ord("A")
- ':' and ';' are optional syntax sugar
- bitwise 16-bit operators &, |, ^, >>, <<
- decimal and hex lower-case '0xffff' notation

I recommend taking a quick look at the many example programs provided.

Error Codes

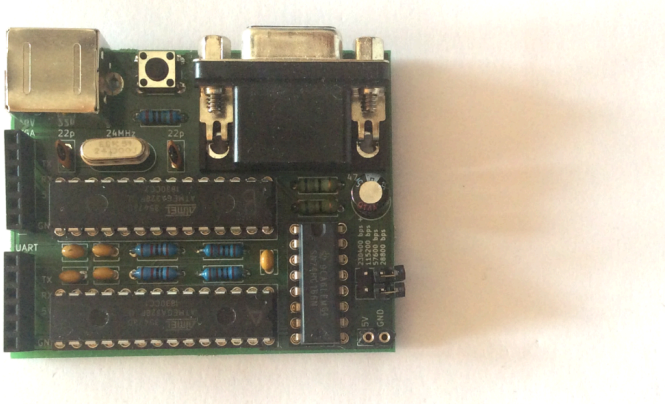
The following table lists the error codes Min uses in its messages alongside with the error line number.

0	unexpected end of file
1	invalid identifier
2	call dictionary full
3	call already exists
4	undefined variable
5	invalid expression
6	type mismatch
7	data memory full
8	variable dictionary full
9	expecting array
10	expecting basetype
11	unexpected indentation
12	unexpected end of block
13	array buffer overflow
14	unexpected break
15	unexpected return
16	unexpected definition
17	invalid parameter

```
18    unexpected include
19    file not found
20    invalid filename
21    invalid index
22    element access without array
34    expecting "
40    expecting (
41    expecting )
44    expecting '
46    expecting .
61    expecting =
93    expecting ]
```

VGA – PS/2 – Terminal

The ‘Minimal Terminal’ is a small PCB add-on (see project repository on GitHub: https://github.com/slu4coder/Minimal-Terminal) that allows you to operate the ‘Minimal CPU System’ without a host PC - it’s basically a pocket-sized stand-alone serial terminal. Just hook up an old PS/2 keyboard and a VGA monitor to the ‘Minimal Terminal’ and connect the Minimal to the UART interface.



The ‘Minimal Terminal’ offers a resolution of 60x25 characters with each character being displayed as an 8x8 pixel matrix and processes the following ANSI control sequences:

```
ESC [ H          Moves the cursor to the left upper corner.
ESC [ J          Clears the screen from cursor position onwards.
ESC [ K          Clears the line from cursor position onwards.
ESC [ <n> A      Moves the cursor <n> steps up.
ESC [ <n> B      Moves the cursor <n> steps down.
ESC [ <n> C      Moves the cursor <n> steps to the right.
ESC [ <n> D      Moves the cursor <n> steps to the left.
ESC [ <n> G      Moves the cursor to column <n>.
ESC [ <n> d      Moves the cursor to row <n>.
```

```
ESC [ S         Scrolls one step up (blank line at the bottom)
ESC [ T         Scrolls one step down (blank line at the top)
```

<n> can be any positive decimal number. The default for <n> is 1 for the 1st column or row. The terminal has a configurable bitrate as described below and otherwise expects the following serial settings:

```
8 data bits
1 start bit
2 stop bits
no parity bit
flow control: none
new line: LF (0x0a)
```

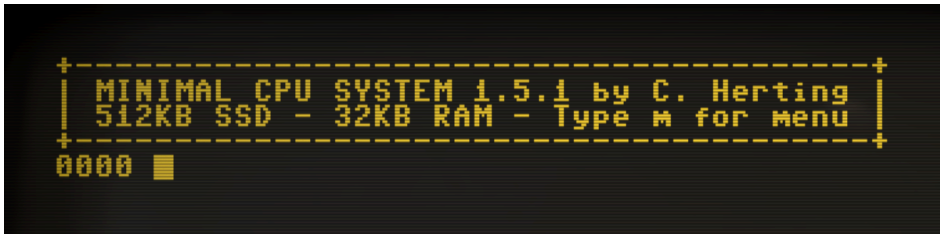
The bitrate of the terminal can be configured with two jumpers. The possible configurations for the pin header on the right edge of the PCB are shown as viewed from the top. Please note that any changes only take effect after a reset or power-up.

```
O  O|      230400 bps
O  O|

O==O|      115200 bps
O  O|

O  O|      57600 bps
O==O|

O==O|      28800 bps
O==O|
```

Please note that the sketch currently only implements German keyboard layout. For any other localization you will have to fiddle with the PS/2-to-ASCII lookup table within the Arduino sketch of the terminal. I am grateful for any contribution here ;-)

Emulator

The ‘Minimal Emulator’ models the Minimal CPU in its revision 1.5, its SSD storage and terminal output cycle-exactly in real time and with a vintage CRT vibe.

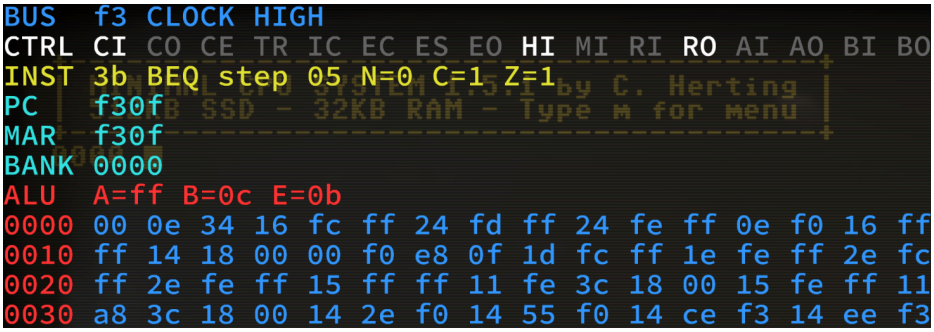


Note: Using the emulator on a slower PC may impose an upper limit on the maximum clock frequency you can emulate.

Commands

The behavior of the emulator can be controlled via keyboard shortcuts:

ALT TAB	Changes focus to another application
ESC END	Quits the emulator The content of the 512KB FLASH is written to 'flash.bin'.
ESC HOME	Resets the CPU (emulated reset key)
ESC 0	Halts the clock
ESC 1..7	Sets clock speed to 10^1..7Hz but no larger than 1.8432MHz
ESC 9	Enables maximum speed mode 3.6864MHz
F1	Toggles the emulator's HUD control screen. Here you can monitor the exact state of the CPU components clock, bus, control word, instruction register, step register, flags register, program counter, memory address and bank register, ALU with A and B registers and a 256-byte RAM page.



F2/3	Changes the RAM page address on the HUD by +/- 0x0100
SHIFT F2/3	Changes the RAM page address on the HUD by +/- 0x1000
ESC s	Performs a full clock cycle and halts the clock
ESC h	Performs half a clock cycle and halts the clock
ESC x	Executes until next instruction fetch (II) and halts

You can copy and paste text into the terminal just like on real hardware by using Alt V or by clicking on the right mouse button.

Character Set

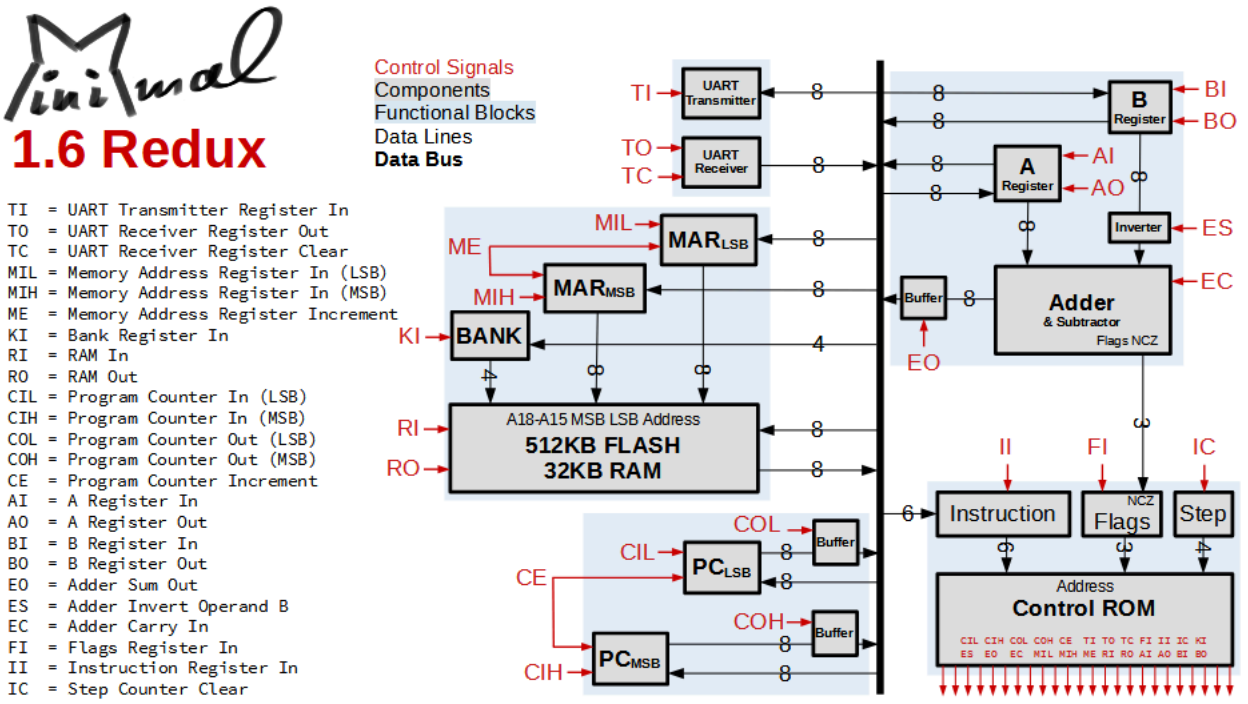
The emulator terminal uses a set of 256 8x8 pixel characters enhancing the capabilities of a standard ASCII terminal and roughly resembling the style of a Commodore C64 or VIC-20. The order of the characters follows ASCII conventions. In the picture below, the character set is shown in rows of 32 characters from left to right, starting at index zero in the left upper corner.

CPU Architecture

So you want to understand how it all works? I'll do my best to describe the various parts and functions of the CPU here. Let's start with a list of the basic features of this machine:

- Von-Neumann architecture
- 8-bit data bus
- 16-bit address space
- 24-bit control word (revision 1.5 uses only 16 control words)
- 2 data registers A and B
- Simple adder with 3 flags (negative N, carry C and zero Z)
- 64 instructions (incl. subroutines, stack- and word operations)
- 32KB RAM
- 512KB FLASH (OS and SSD with file system)
- UART interface (terminal display, keyboard input and data I/O)

For the rather general description I am going to give, let's use the block diagram of the 'Minimal CPU' revision 1.6 ('Redux') shown below as an overview. It exposes the functionality in its most accessible form.



Components & Control Signals

At first glance, any CPU architecture looks daunting since it consists of various different functional blocks (shown in blue). Each block may contain several components (shown in gray). To a large part, the complex behavior of a CPU system stems from the fact that - like most complex systems - it involves feedback. In the following I will talk about components and their control signals. On the one hand control signals manage components and on the other hand components manage control signals. See the feedback?

Generally, each component of the CPU has an input and an output. It's behavior is controlled by little switches. These switches are operated by control signals. Part of the "magic" of a CPU is to generate its own control signals, press its own buttons, if you will. Before we discuss how that is accomplished, let me give a brief description of each component, its control switches (labeled in RED in the above diagram) and its inputs and outputs.

8-Bit Data Bus

The data bus is a "glue" component that connects inputs and outputs of most components. This backbone of the CPU is nothing more than 8 data lines (shown above as a single black vertical line). The statement 'a component outputs something to the bus' means that it drives each bus line to a well-defined voltage level (HIGH or LOW). With each bus line

n = 0..7 a value 2^n is associated such that in total values 0..255 can be represented.

There is one property of central importance when it comes to understanding the “magic” of the ‘Minimal’: The data bus lines are pulled HIGH via 470 Ohm resistors. That means that if no component is *driving* the bus, *reading* the bus will yield 0b11111111 = 0xff or -1.
At this point, let me just say that the ability of generating this value is the key to the more sophisticated stack operations the ‘Minimal’ is capable of. But let’s continue by discussing all the other components.

System Clock

The system clock connects to almost every component of a CPU and synchronizes all activities to a common pace so that components can work together. The system clock defines the processing speed of a CPU.
At the start of a clock cycle (clock going LOW) new control signals are set. This may force a component to output data onto the bus and another component to read data from the bus. It is important to note that the reading operation happens in the middle of the clock cycle with the rising edge of the clock – during a time when the output has stabilized.

```
System clock:    ----_________________---------------____

Clock cycle:         <------------------------------>

Control signal: ____--------------------------------____
```

A-Register (Accumulator)

The A-register can hold a single data byte. Its input and output is connected to the data bus. The control switch AI (for A in) reads the value on the data bus into the register. AO (for A out) puts the stored value onto the bus. Furthermore, the value stored in A is permanently connected to the input of the adder component.

B-Register

The B-register works just the same as the A-register and is also permanently connected to the adder component.

Adder (ALU)

The adder adds the values of A and B. EO (for sum out) puts this result onto the bus. ES (for inversion) inverts B prior to adding it to A. EC (for carry in) adds one to the result. This allows us to perform subtractions, since in 2’s complement A - B = A + (~B) + 1,

where ~B denotes the inverse of B. Thus, in order to output A - B, the control signals EO, ES and EC must all be active at once.
Depending on its result, the adder outputs three flags: N negative, C carry and Z zero.

N = 1 if the most significant bit of the result is HIGH.
C = 1 if the result of a calculation exceeds 0xff.
Z = 1 if the result of a calculation is zero or 0x00.

Memory (RAM and FLASH)

Memory integrates a large number of single-byte register cells into one chip. Each cell has a unique index or address. To select a certain cell, the address bits have to be present at the address inputs of the memory chip. RO (for RAM out) will output the content of the selected cell onto the bus. RI (for RAM) stores the bus value into the selected cell.

Memory Address Register (MAR)

The memory address register (MAR) consists of three registers: A 16-bit counter with its most significant byte (MSB) MAR_H and least significant byte (LSB) MAR_L part and an additional 4-bit BANK register. The output lines of all three registers are permanently connected to the memory address lines of the RAM and FLASH chip, where memory line M15 selects whether RAM is enabled (M15 = HIGH, address range 0x8000-0xffff) or FLASH is active (M15 = LOW, address range 0x0000-0x7fff).
Two control signals MIL and MIH (for memory address register input low and high) read a value from the bus into either the MAR_L or MAR_H part. ME increments the current value held by MAR_L and MAR_H. KI (for memory bank in) reads the lower 4 bus bits into the memory BANK register, defining which 32KB bank of the FLASH IC will be accessed by the processor.

Program Counter (PC_L, PC_H)

The program counter consists of two 8-bit registers, holding the most significant byte PC_H and least significant byte PC_L of a 16-bit memory address. The program counter points to the instruction in memory which is currently being processed. Like that MAR, the program counter can also increment (count up one): If CE (for count enable) is active, incrementing happens at the rising edge of the clock. For writing and reading data to and from the register, four control signals are used: COL/COH for counter out low/high and CIL/CIH for counter in low/high.

Terminal Registers (TR)

The terminal ‘sender register’ transmits its content via UART. TI reads bus data into the register for transmission. The terminal ‘receiver register’ reads a byte from the UART. TO outputs the last received byte to the bus and TC clears the register to 0xff.

Flags Register (FR)

Upon FI the state of the three adder flags N, C and Z is being stored in this register.

Instruction Register (IR)

The 6-bit instruction register holds the opcode of the current instruction to be processed. II reads in a new opcode from the bus.

Step Counter (Step)

The step counter is 4 bits wide and holds the current step of an instruction. Upon IC (for instruction step clear) this counter is reset to zero.

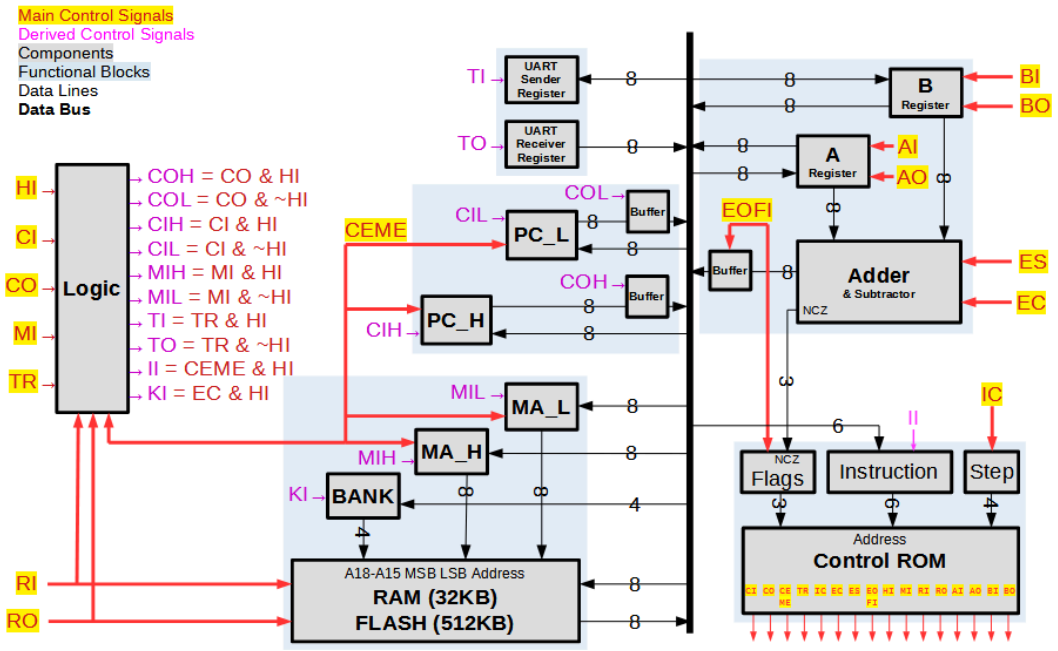
Control Word Generation

The 4 bits of the step counter with the 6 bits of the instruction register and the 4 bits of the flags register together form a 14-bit address input to the control ROMs. Each control ROM outputs 8 control signals. The state of all these control signals is called a control word.

Let’s briefly discuss the difference between revision 1.5 (‘Expanded PCB’) and revision 1.6 (‘Redux’) here. Revision 1.6 uses 3 control ROMs to generate the 24 control signals we have discussed above directly. Revision 1.5 however only uses two control ROMs for the 16 main control signals

AI, AO, BI, BO, MI, CI, CO, HI, EO, ES, EC, TR, CE, RI, RO, IC.

The other control signals are derived by the use of logic ICs as given in the block schematic of revision 1.5 shown below.



Processing Instructions

Upon pressing RESET, both the program counter and the instruction step counter are set to zero. Regardless of the state of the flags register and instruction register, the instruction steps 0, 1 and 2 have fixed control words:

```
Step 0:    COL|MIL
Step 1:    COH|MIH
Step 2:    RO|II|CE|ME
Step 3:    ...
.
Step 15:   ...
```

Let's see what they do: In step 0 and 1 the LSB and MSB part of the program counter is copied to the memory address register, respectively. In step 2, RO puts the content of the memory location the PC is pointing to onto the bus. That is the opcode of the next instruction! II stored this opcode in the instruction register. CE|ME increments both the program counter and the memory address register so that further arguments of the instruction can be read in. Since by now, the instruction register "knows" about the new instruction, steps 3 to 15 are different for each instruction, since they are selected by the opcode inside the instruction register and the content of the flags register. Each instruction consists of a meaningful step sequence of control words called microcode. A list of the microcode of each instruction is given in the appendix.

The IC signal marks the end of an instruction and resets the step counter back to zero.

Reaching 8.3MHz

What is that critical path limiting the clock speed? Well, it is the longest streak of operations that must occur in sequence. For the Minimal, it's the path where the step counter is reset at the end of an instruction: (A) the step counter advances, (B) the control word updates pulling (C) the reset of the step counter, (D) immediately resulting in another or 2nd update of the control word, ending with a stable bus output of the microcode step 0: COL|MIL. Compared to other paths, this sequence leaves us with the least time until the rising edge of the clock looms and all outputs need to be stable for reading:

#	Time	Operation
I	10ns	clock inversion (NAND)
A	15ns	step counter increments
B	30ns	1st CTRL update (resetting step counter)
C	15ns	step counter reset after ~MR
D	30ns	2nd CTRL update (0: COL|MIL of next instr.)
E	20ns	PC 74HC245 buffer has stable bus output
.	10ns	overall jitter safety-margin
Z	15ns	all outputs assume high-Z state
T	40ns	bus pull-up to 2.5V (trigger level)
H	25ns	bus pull-up to 3.5V (worst case high level)

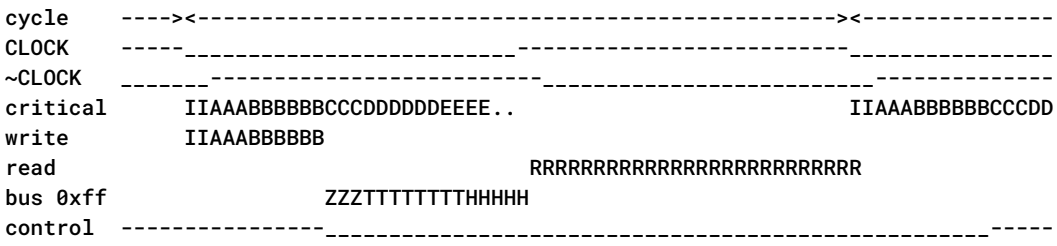
Items T and H result from the RC time constant T formed by the pull-up resistor and line capacitance of each bus line:

R = 470Ω and C = 14pF + 11 * 3.5pF + 7 * 10pF = 123pF

=> T = ln(2) * R * C = 40ns

where C is the sum of the capacitance (d=0.5mm, h=1.5cm, l=1.2m) of the bus wire above a ground plane and the input and output capacitances of the connected ICs. The times given in the table above have all been experimentally verified to +/-5ns.

The following timing diagrams show the critical path for a standard clock cycle as described above (and as used by revision 1.5) for a cycle time of 260ns (3.8MHz). Each digit represents a duration of 5ns:



The diagram below shows the highly optimized clock cycle of revision 1.6 for a cycle time of 120ns (8.3MHz). Again, each digit represents a duration of 5ns:

Processing Power

Let me first point out that measuring computer processing power can be very dependent on the benchmark test. So choosing a task that represents the actual work the computer is supposed to do is essential. In the early days of computing the unit 'Mips' (million instructions per second) was popular for comparing different CPUs.
This unit is a bit unfortunate though, since it is insensitive to the amount of work a CPU gets done within a single instruction.
To give you an example, a specialty of the Minimal is its word addressing mode: ADW adds the accumulator to the LSB of a 16-bit value at a given memory address and in case of an overflow also automatically increments the MSB. The famous MOS 6502 processor lacks such a functionality and thus needs multiple 8-bit instructions for the same task. Misleadingly, this results in a potentially higher Mips value.

Having that said, the 6502 processor is rated 0.43Mips at 1MHz translating into an average of 2.3cpi ('clocks per instruction'). Obviously, the number of instructions per second (ips) can be calculated by dividing the clock frequency by the number of clocks per instruction. At its time the 6502 was quite an efficient pipelined CPU.

The Minimal, while running the MIN interpreter as a benchmark, uses on average 8.18cpi. Compared to the 6502, this value is much higher since the Minimal a) by design lacks the efficiency of a pipelined CPU and b) features more advanced instructions than the 6502. You see, it's complicated.

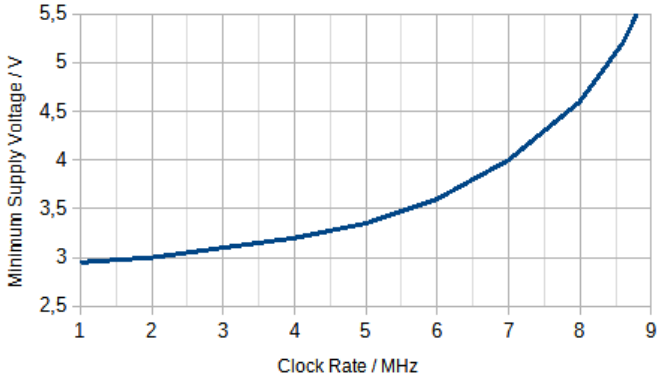
On the Minimal we get 0.122Mips per MHz clock rate, i. e. 0.45Mips at 3.6864MHz (comparable to a C64 or Apple II) or 1.01Mips at 8.3MHz.

Appendix

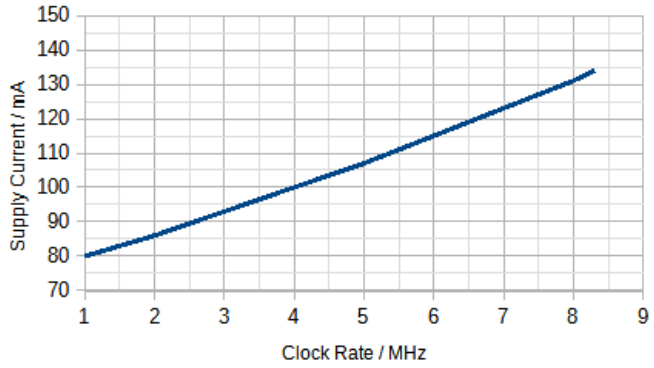
Technical Specification

Maximum Supply Voltage		5.5V
Maximum Clock Frequency	(rev1.5 @ 5V)	3.8MHz
	(rev1.6 @ 5V)	8.3MHz
Recommended Clock Frequency	(rev1.5 @ 5V)	3.6468MHz
	(rev1.6 @ 5V)	8.0000MHz
Relative UART Bitrate Tolerance		+/-4%
Recommended UART Line Delay	(8.0000MHz)	10ms
	(3.6864MHz)	20ms
	(1.8432MHz)	40ms
Processing Power	(3.6864MHz)	0.45Mips
	(8.0000MHz)	0.98Mips
	(8.3000MHz)	1.01Mips

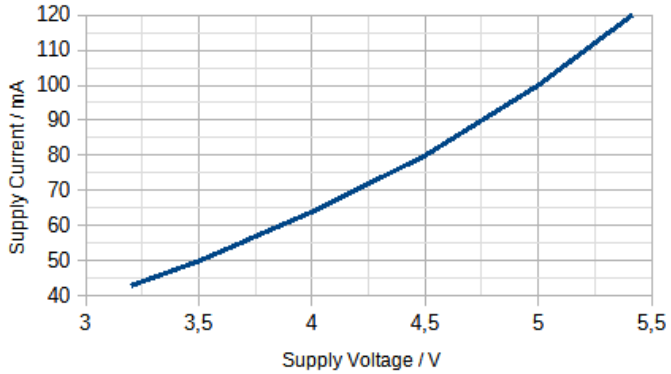
Minimum supply voltage of rev1.6 as a function of clock frequency:



Supply current of rev1.6 at 5V supply voltage as a function of clock frequency:



Supply current of rev1.6 at 4MHz clock frequency as a function of supply voltage:



Microcode 1.5 ('Enhanced')

```
#define NOP     CO|MI, CO|MI|HI, RO|HI|CEME, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define BNK     CO|MI, CO|MI|HI, RO|HI|CEME, AO|EC|HI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define Out     CO|MI, CO|MI|HI, RO|HI|CEME, AO|TR|HI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define CLC     CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|ES,          IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define SEC     CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|ES|EC,       IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define LSL     CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|AI,          IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ROL0    CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|AI,          IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ROL1    CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|AI|EC,       IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define LSR0    CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|ES,          EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    IC, 0, 0
#define LSR1    CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|ES,          EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, IC, 0, 0
#define ROR0    CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|AI|BI,       EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    IC,            0,  0, 0
#define ROR1    CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI, EOFI|AI|BI|EC,    EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, IC,            0,  0, 0
#define ASR00x  CO|MI, CO|MI|HI, RO|HI|CEME, BI, EOFI|EC, AO|BI,      EOFI|ES,       EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    IC
#define ASR01x  CO|MI, CO|MI|HI, RO|HI|CEME, BI, EOFI|EC, AO|BI,      EOFI|ES,       EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, IC
#define ASR10x  CO|MI, CO|MI|HI, RO|HI|CEME, BI, EOFI|EC, AO|BI,      EOFI|ES|EC,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    EOFI|AI|BI,    IC
#define ASR11x  CO|MI, CO|MI|HI, RO|HI|CEME, BI, EOFI|EC, AO|BI,      EOFI|ES|EC,    EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, EOFI|EC|AI|BI, IC
#define INP     CO|MI, CO|MI|HI, RO|HI|CEME, TR|AI|BI, EOFI|ES|BI,    EOFI|ES|EC,    IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define NEG     CO|MI, CO|MI|HI, RO|HI|CEME, AO|BI,    EOFI|ES|EC|AI, EOFI|ES|EC|AI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define Inc     CO|MI, CO|MI|HI, RO|HI|CEME, BI,       EOFI|ES|EC|AI, IC,             0,  0, 0, 0, 0, 0, 0, 0, 0, 0
#define Dec     CO|MI, CO|MI|HI, RO|HI|CEME, BI,       EOFI|AI,       IC,             0,  0, 0, 0, 0, 0, 0, 0, 0, 0

#define LDI     CO|MI, CO|MI|HI, RO|HI|CEME, RO|AI|CEME, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ADI     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI, EOFI|AI|CEME, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define SBI     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI, EOFI|ES|EC|AI|CEME, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define CPI     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI, EOFI|ES|EC|CEME, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ACI0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI, EOFI|AI|CEME, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ACI1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI, EOFI|EC|AI|CEME, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define SCI0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI, EOFI|ES|AI|CEME, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define SCI1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI, EOFI|ES|EC|AI|CEME, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0

#define JPA     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|CI|HI, BO|CI, IC, 0, 0, 0, 0, 0, 0, 0, 0
#define LDA     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|CEME, IC, 0, 0, 0, 0, 0, 0, 0
#define STA     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|RI, CEME, IC, 0, 0, 0, 0, 0, 0
#define ADA     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|AI|CEME, IC, 0, 0, 0, 0, 0, 0
#define SBA     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|ES|EC|AI|CEME, IC, 0, 0, 0, 0, 0, 0
#define CPA     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|ES|EC|CEME, IC, 0, 0, 0, 0, 0, 0
#define ACA0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|AI|CEME, IC, 0, 0, 0, 0, 0, 0
#define ACA1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|EC|AI|CEME, IC, 0, 0, 0, 0, 0, 0
#define SCA0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|ES|AI|CEME, IC, 0, 0, 0, 0, 0, 0
#define SCA1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|ES|EC|AI|CEME, IC, 0, 0, 0, 0, 0, 0

#define JPR     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|CI|HI, BO|CI, IC, 0, 0, 0, 0, 0, 0
#define LDR     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI, IC, 0, 0, 0, 0, 0
#define STR     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, AO|RI, IC, 0, 0, 0, 0, 0
#define ADR     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|AI, IC, 0, 0, 0, 0
#define SBR     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|ES|EC|AI, IC, 0, 0, 0, 0
#define CPR     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|ES|EC, IC, 0, 0, 0, 0
#define ACR0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|AI, IC, 0, 0, 0, 0
#define ACR1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|EC|AI, IC, 0, 0, 0, 0
#define SCR0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|ES|AI, IC, 0, 0, 0, 0
#define SCR1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|ES|EC|AI, IC, 0, 0, 0, 0

#define CLB     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI|AI, EOFI|ES|EC|RI, EOFI|ES|EC|AI|CEME, IC, 0, 0, 0, 0, 0, 0
#define NEB     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|AI, EOFI|ES|EC|RI, EOFI|ES|EC|AI|CEME, IC, 0, 0, 0, 0, 0
#define INB     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|BI, EOFI|EC|RI, EOFI|EC|AI|CEME, IC, 0, 0, 0, 0, 0
#define DEB     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|BI, EOFI|ES|RI, EOFI|ES|AI|CEME, IC, 0, 0, 0, 0, 0
#define ADB     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|RI, CEME, IC, 0, 0, 0, 0, 0, 0
#define SBB     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|ES|EC|RI, BO|AI|CEME, IC, 0, 0, 0, 0, 0
#define ACB0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|AI,      AO|RI, BO|AI|CEME, IC, 0, 0, 0, 0
#define ACB1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|EC|AI,   AO|RI, BO|AI|CEME, IC, 0, 0, 0, 0
#define SCB0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|ES|AI,   AO|RI, BO|AI|CEME, IC, 0, 0, 0, 0
#define SCB1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|ES|EC|AI, AO|RI, BO|AI|CEME, IC, 0, 0, 0, 0

#define CLW     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI,    EOFI|ES|EC|RI, CEME, EOFI|ES|EC|RI, IC, 0, 0, 0, 0, 0
#define NEW0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|AI, EOFI|ES|EC|RI, CEME, RO|BI, EOFI|ES|AI,    AO|RI, IC, 0, 0
```

```
#define NEW1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|AI, EOFI|ES|EC|RI, CEME, RO|BI, EOFI|ES|EC|AI,  AO|RI, IC, 0, 0
#define INW0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|BI, EOFI|EC|RI, CEME|BI, RO|AI, EOFI|ES|AI,    AO|RI, IC, 0, 0
#define INW1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|BI, EOFI|EC|RI, CEME|BI, RO|AI, EOFI|ES|EC|AI,  AO|RI, IC, 0, 0
#define DEW0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|BI, EOFI|ES|RI, CEME|BI, RO|AI, EOFI|AI,       AO|RI, IC, 0, 0
#define DEW1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|AI|BI, EOFI|ES|EC|BI, EOFI|ES|RI, CEME|BI, RO|AI, EOFI|EC|AI,    AO|RI, IC, 0, 0
#define ADW0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|RI, CEME|BI, RO|AI, EOFI|ES|AI,    AO|RI, IC, 0, 0
#define ADW1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|RI, CEME|BI, RO|AI, EOFI|ES|EC|AI, AO|RI, IC, 0, 0
#define SBW0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|ES|EC|RI, CEME|BI, RO|AI, EOFI|AI,    AO|RI, IC, 0, 0
#define SBW1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|ES|EC|RI, CEME|BI, RO|AI, EOFI|EC|AI, AO|RI, IC, 0, 0
#define ACW0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|BI,    BO|RI, CEME|BI, RO|AI, EOFI|ES|AI,    AO|RI, IC, 0, 0
#define ACW1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, RO|BI, EOFI|EC|BI, BO|RI, CEME|BI, RO|AI, EOFI|ES|EC|AI, AO|RI, IC, 0, 0
#define SCW0    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|ES|BI,    BO|RI, CEME|BI, RO|AI, EOFI|AI,    AO|RI, IC, 0
#define SCW1    CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|MI|HI, BO|MI, AO|BI, RO|AI, EOFI|ES|EC|BI, BO|RI, CEME|BI, RO|AI, EOFI|EC|AI, AO|RI, IC, 0

#define LDS     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, MI|HI, MI, RO|AI, EOFI|MI, RO|AI, IC, 0, 0, 0, 0, 0, 0
#define STS     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, MI|HI, MI, RO|MI, AO|RI, MI, RO|AI, EOFI|BI, MI, RO|MI, RO|AI, BO|MI, AO|RI
#define PHS     CO|MI, CO|MI|HI, RO|HI|CEME, MI|HI, MI, RO|MI|BI, AO|RI, BO|AI,  EOFI|ES|BI|MI, EOFI|RI, AO|MI, RO|AI, IC, 0, 0, 0
#define PLS     CO|MI, CO|MI|HI, RO|HI|CEME, MI|HI, MI|BI, RO|AI, EOFI|ES|EC|AI, AO|RI, AO|MI, RO|AI, IC, 0, 0, 0, 0, 0
#define JPS     CO|MI, CO|MI|HI, RO|HI|CEME, MI|HI, MI|BI, RO|AI|MI, CO|RI, EOFI|AI|MI, CO|RI|HI, BO|MI, EOFI|RI, CO|MI, CO|MI|HI, RO|BI|CEME, RO|CI|HI, BO|CI
#define RTS     CO|MI, CO|MI|HI, RO|HI|CEME, MI|HI, MI|BI, RO|AI|MI, EOFI|ES|EC|AI|MI, RO|CI|HI, EOFI|ES|EC|AI|MI, RO|CI, BO|MI, AO|RI, CEME, CEME, IC, 0

#define BRA     CO|MI, CO|MI|HI, RO|HI|CEME, RO|BI|CEME, RO|CI|HI, BO|CI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ___     CO|MI, CO|MI|HI, RO|HI|CEME, CEME,       CEME,    IC,   0,  0, 0, 0, 0, 0, 0, 0, 0, 0
```

Microcode 1.6 (‘Redux’)

```
#define NOP     COL|MIL, COH|MIH, RO|II|CE|ME, 0,     0,            0,  0, 0, 0, 0, 0, 0, 0, 0, 0
#define BNK     COL|MIL, COH|MIH, RO|II|CE|ME, AO|NI, IC,           0,  0, 0, 0, 0, 0, 0, 0, 0, 0
#define OUT     COL|MIL, COH|MIH, RO|II|CE|ME, AO|TI, IC,           0,  0, 0, 0, 0, 0, 0, 0, 0, 0
#define CLC     COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|ES,     IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define SEC     COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|ES|EC,  IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define LSL     COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|AI,     IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ROLx0x COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|AI,     IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ROLx1x COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|AI|EC,  IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define LSRx0x COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|ES,     EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    IC,           0,            0
#define LSRx1x COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|ES,     EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, IC,           0,            0
#define RORx0x COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|AI|BI,  EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    IC,           0,            0
#define RORx1x COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|FI|AI|BI|EC, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, IC,           0,            0
#define ASR00x COL|MIL, COH|MIH, RO|II|CE|ME, FF|BI, EO|FI|EC,     AO|BI,          EO|FI|ES,       EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    IC
#define ASR01x COL|MIL, COH|MIH, RO|II|CE|ME, FF|BI, EO|FI|EC,     AO|BI,          EO|FI|ES,       EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, IC
#define ASR10x COL|MIL, COH|MIH, RO|II|CE|ME, FF|BI, EO|FI|EC,     AO|BI,          EO|FI|ES|EC,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    EO|FI|AI|BI,    IC
#define ASR11x COL|MIL, COH|MIH, RO|II|CE|ME, FF|BI, EO|FI|EC,     AO|BI,          EO|FI|ES|EC,    EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, EO|FI|EC|AI|BI, IC
#define INP0    COL|MIL, COH|MIH, RO|II|CE|ME, FF|BI|FI, TO|AI, EO|ES|EC|FI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define INP1    COL|MIL, COH|MIH, RO|II|CE|ME, FF|BI|FI, FF|AI, EO|ES|EC|FI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define NEG     COL|MIL, COH|MIH, RO|II|CE|ME, AO|BI, EO|ES|EC|AI, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0, 0, 0, 0, 0
#define INC     COL|MIL, COH|MIH, RO|II|CE|ME, FF|BI, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define DEC     COL|MIL, COH|MIH, RO|II|CE|ME, FF|BI, EO|FI|AI,      IC, 0, 0, 0, 0, 0, 0, 0, 0, 0

#define LDI     COL|MIL, COH|MIH, RO|II|CE|ME, RO|AI|CE, IC,           0,  0, 0, 0, 0, 0, 0, 0, 0, 0
#define CPI     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, EO|FI|ES|EC,   IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ADI     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, EO|FI|AI,      IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ACIx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, EO|FI|AI,      IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ACIx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, EO|FI|EC|AI,   IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define SBI     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define SCIx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, EO|FI|ES|AI,   IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define SCIx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0

#define JPA     COL|MIL, COH|MIH, RO|II|CE|ME, RO|CIL|ME,   RO|CIH,    IC, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define LDA     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI, IC, 0, 0, 0, 0, 0, 0, 0
#define STA     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|RI, IC, 0, 0, 0, 0, 0, 0, 0
#define CPA     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|ES|EC,   IC, 0, 0, 0, 0, 0, 0
#define ADA     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|AI,      IC, 0, 0, 0, 0, 0, 0
#define ACAx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|AI,      IC, 0, 0, 0, 0, 0, 0
#define ACAx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|EC|AI,   IC, 0, 0, 0, 0, 0, 0
#define SBA     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0, 0, 0
#define SCAx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|ES|AI,   IC, 0, 0, 0, 0, 0, 0
#define SCAx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0, 0, 0

#define JPR     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|CIL|ME, RO|CIH, IC, 0, 0, 0, 0, 0, 0
#define LDR     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, RO|AI, IC, 0, 0, 0, 0, 0
#define STR     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, AO|RI, IC, 0, 0, 0, 0, 0
#define CPR     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, RO|BI, EO|FI|ES|EC,   IC, 0, 0, 0, 0
#define ADR     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, RO|BI, EO|FI|AI,      IC, 0, 0, 0, 0
#define ACRx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, RO|BI, EO|FI|AI,      IC, 0, 0, 0, 0
#define ACRx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, RO|BI, EO|FI|EC|AI,   IC, 0, 0, 0, 0
#define SBR     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, RO|BI, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0
#define SCRx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, RO|BI, EO|FI|ES|AI,   IC, 0, 0, 0, 0
#define SCRx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI|ME,  RO|MIH, BO|MIL, RO|BI, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0

#define CLB     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI,    EO|ES|EC|RI, IC, 0, 0, 0, 0, 0, 0
#define NEB     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|AI, EO|ES|EC|RI, EO|FI|ES|EC|AI, IC, 0, 0, 0, 0, 0
#define INB     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|BI, EO|EC|RI, EO|FI|EC|AI, IC, 0, 0, 0, 0, 0
#define DEB     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|BI, EO|ES|RI, EO|FI|ES|AI, IC, 0, 0, 0, 0, 0
#define ADB     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI,    EO|FI|RI,    IC, 0, 0, 0, 0, 0, 0
#define ACBx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI,    EO|FI|BI,    BO|RI, IC, 0, 0, 0, 0, 0
#define ACBx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI,    EO|FI|EC|BI, BO|RI, IC, 0, 0, 0, 0, 0
#define SBB     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI,    RO|AI, EO|FI|ES|EC|RI, BO|AI, IC, 0, 0, 0, 0, 0
#define SCBx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI,    RO|AI, EO|FI|ES|AI,    AO|RI, BO|AI, IC, 0, 0, 0, 0
#define SCBx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI,    RO|AI, EO|FI|ES|EC|AI, AO|RI, BO|AI, IC, 0, 0, 0, 0

#define CLW     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI,    EO|ES|EC|RI, ME, EO|ES|EC|RI, IC, 0, 0, 0, 0, 0
#define NEWx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|AI, EO|FI|ES|EC|RI, ME,       RO|BI, EO|FI|ES|AI,    AO|RI, IC, 0, 0
#define NEWx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|AI, EO|FI|ES|EC|RI, ME,       RO|BI, EO|FI|ES|EC|AI, AO|RI, IC, 0, 0
#define INWx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|BI, EO|FI|EC|RI,    ME|FF|BI, RO|AI, EO|FI|ES|AI,    AO|RI, IC, 0, 0
#define INWx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|BI, EO|FI|EC|RI,    ME|FF|BI, RO|AI, EO|FI|ES|EC|AI, AO|RI, IC, 0, 0
#define DEWx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|BI, EO|FI|ES|RI,    ME|FF|BI, RO|AI, EO|FI|AI,      AO|RI, IC, 0, 0
#define DEWx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|AI|BI, EO|ES|EC|BI, EO|FI|ES|RI,    ME|FF|BI, RO|AI, EO|FI|EC|AI,    AO|RI, IC, 0, 0
#define ADWx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|RI,          ME|FF|BI, RO|AI, EO|FI|ES|AI,    AO|RI, IC, 0, 0
#define ADWx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|RI,          ME|FF|BI, RO|AI, EO|FI|ES|EC|AI, AO|RI, IC, 0, 0
#define ACWx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|BI,    BO|RI, ME|FF|BI, RO|AI, EO|FI|ES|AI,    AO|RI, IC, 0, 0
#define ACWx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, RO|BI, EO|FI|EC|BI, BO|RI, ME|FF|BI, RO|AI, EO|FI|ES|EC|AI, AO|RI, IC, 0, 0
#define SBWx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI, RO|AI, EO|FI|ES|EC|RI,       ME|FF|BI, RO|AI, EO|FI|AI,    AO|RI, IC, 0, 0
#define SBWx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI, RO|AI, EO|FI|ES|EC|RI,       ME|FF|BI, RO|AI, EO|FI|EC|AI, AO|RI, IC, 0, 0
#define SCWx0x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI, RO|AI, EO|FI|ES|BI,    BO|RI, ME|FF|BI, RO|AI, EO|FI|AI,    AO|RI, IC, 0
#define SCWx1x COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE|ME, RO|MIH|CE, BO|MIL, AO|BI, RO|AI, EO|FI|ES|EC|BI, BO|RI, ME|FF|BI, RO|AI, EO|FI|EC|AI, AO|RI, IC, 0

#define LDS     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, FF|MIH|MIL, RO|AI, EO|MIL, RO|AI, IC, 0, 0, 0, 0, 0, 0, 0
#define STS     COL|MIL, COH|MIH, RO|II|CE|ME, RO|BI|CE, FF|MIH|MIL, RO|MIL, AO|RI, FF|MIL, RO|AI, EO|BI, FF|MIL, RO|MIL, RO|AI, BO|MIL, AO|RI, IC
#define PHS     COL|MIL, COH|MIH, RO|II|CE|ME, FF|MIH|MIL, RO|MIL|BI, AO|RI, BO|AI,  EO|ES|BI|MIL, EO|RI, AO|MIL, RO|AI, IC, 0, 0, 0, 0
#define PLS     COL|MIL, COH|MIH, RO|II|CE|ME, FF|MIH|MIL|BI, RO|AI, EO|ES|EC|AI, AO|RI, AO|MIL, RO|AI, IC, 0, 0, 0, 0, 0, 0
#define JPS     COL|MIL, COH|MIH, RO|II|CE|ME, FF|MIH|MIL|BI, RO|AI|MIL, COL|RI, EO|AI|MIL, COH|RI, BO|MIL, EO|RI, COL|MIL, COH|MIH, RO|CIL|ME, RO|CIH, IC, 0
#define RTS     COL|MIL, COH|MIH, RO|II|CE|ME, FF|MIH|MIL, RO|MIL|CIL, CE|ME, CE, COL|BI, RO|CIH|ME, RO|CIL, FF|MIL|CE, BO|RI|CE, IC, 0, 0, 0

#define BRA     COL|MIL, COH|MIH, RO|II|CE|ME, RO|CIL|ME, RO|CIH, IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
#define ___     COL|MIL, COH|MIH, RO|II|CE|ME, CE,       CE,     IC, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

EBNF of Min

```
letter      = 'a' | ... | 'z' | 'A' | ... | 'Z' ;
digit       = '0' | ... | '9' ;
rel-op      = '==' | '!=' | '<' | '>' | '<=' | '>=' ;
str-op      = '==' | '!=' ;
add-op      = '+'  | '-' ;
mul-op      = '*'  | '/' ;
```

```
number       = digit, { digit } ;
identifier   = letter, { letter | digit | '_' } ;
whitespace   = ' ' | '\r' | '\t' | ':' | ';' ;
character    = ? any ASCII character ? ;
hexdigit     = digit | 'a' | ... | 'f' ;
hexnumber    = '0x', hexdigit, { hexdigit } ;

file         = { statement }, { NEWLINE }, ENDMARKER ;
statement    = { NEWLINE }, ? OKDENT ?, simple-line | compound-stmt ;
simple-line  = simple-stmt, { simple-stmt }, NEWLINE ;

simple-stmt  = 'print', expr-list
             | 'break'
             | 'return', [ expr ]
             | identifier, '(', [ expr-list ], ')'         (* sub call *)
             | identifier, '=', expr ;                     (* assignment *)

compound-stmt   = 'if', bool-expr, block,
                      { { NEWLINE }, ? OKDENT ?, 'elif', block },
                      [ { NEWLINE }, ? OKDENT ?, 'else', block ]
             | 'while', bool-expr, block
             | 'def', ? NODENT ?, identifier, '(', [ param-list ], ')', block ;
             | 'include', '"', { character }, '"'

block        = NEWLINE, ? INDENT ?, statement, { statement }, ? DEDENT ?
             | simple-line ;

expr         = array-expr | math-expr ;
expr-list    = expr, { ',', expr } ;
param-list   = [ '&' ], identifier, { ',', [ '&' ], identifier } ;

array        = '[', [ math-expr, { ',', math-expr } ], ']'   (* returns int[] *)
             | '"', { character }, '"'                       (* returns chr[] *)
             | identifier, '(', [ expr-list ], ')'
             | identifier
             | array, '[', math-expr, '..', math-expr, ']' ; (* array slicing *)
array-expr   = array { '+' array } ;                         (* concatenation *)

math-factor  = '(', math-expr, ')'
             | number | hexnumber
             | "'", character, "'"
             | 'key', '(', ')'
             | 'len', '(', array-expr, ')'
             | 'rnd', '(', ')'                               (* pseudo-random byte *)
             | identifier '(', [ expr-list ], ')'
             | identifier
             | array, '[', math-expr, ']' ;                  (* element access *)
math-inv     = [ '~' ], math-factor
math-bitwise = math-inv, { ( '&' | '|' | '^' | '>>' | '<<' ), math-inv } ;
math-term    = math-bitwise, { mul-op, math-bitwise } ;
math-expr    = [ add-op ], math-term { add-op, math-term } ;

bool-factor  = [ 'not' ], math-expr, [ rel-op, math-expr ]
```

```
            | [ 'not' ], array-expr, [ str-op, array-expr ] ;
bool-term    = bool-factor, { 'and', bool-factor } ;
bool-expr    = bool-term, { 'or', bool-term } ;
```

Updating the OS

For the very first power-up of your Minimal, it is necessary to write the operating system onto the FLASH IC by using an external programmer. Once you have a version of the OS running, you can update the OS 'in situ'. Anytime you press the reset button, the OS copies itself from its FLASH image location (first 4KB of bank 0) into RAM 0xf000 and runs there. And while the OS is running from RAM you can update it's FLASH image like so:

1) Clear the RAM area 0xe000-0efff by typing: v e000 efff 0 ENTER.
2) Assemble the operating system 'os.txt' on your host system.
3) On the Minimal type: e000 ENTER and copy & paste the OS hex code.
4) Assemble and upload the program 'prom.txt' to the Minimal.
5) Start 'prom' by typing: 8000 r ENTER.
6) 'prom' will ask you if you want to copy 0xe000-0xefff to 0x0000. Just hit 'y' and you are greeted by the new OS ;-)

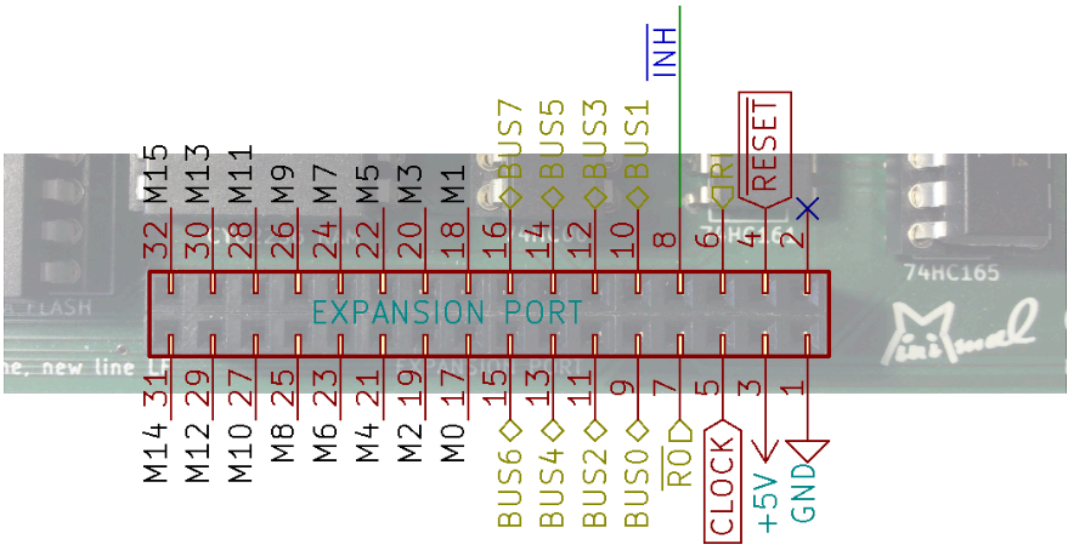
Expansion Port

The Minimal features a stackable expansion port allowing you to expand the CPU system with any functionality that can be accessed by the 'memory-mapping' technique: Within the Minimal, a so-called inhibit signal ~INH is pulled inactive by default. Whenever an expansion actively pulls ~INH low, the FLASH memory and RAM of the Minimal are disabled (i. e. the ~CE chip enable lines are forced low) and their outputs will remain in a high-impedance state.

By pulling ~INH low, the expansion card effectively hijacks the memory address and bus lines, thus redirecting memory I/O operations of the CPU to the expansion hardware.

The expansion card pulls the ~INH signal only low if the CPU accesses certain memory locations associated with the expansion card's functionality. The necessary address decoding logic resides on the expansion card.

Here is a pinout of the expansion port (top view of the PCB):



Pin	Direction	Description
1	GND	ground
2	N/C	not connected
3	+5V	supply voltage
4	out	reset line of the CPU (active low)
5	out	system clock
6	out	CPU control line ‘RAM input’ (active high)
7	out	CPU control line ‘RAM output’ (active low)
8	in	RAM/ROM access inhibit signal (active low)
9 - 16	in/out	bus line bit 0-7
17 - 32	out	memory address register bit 0-15

Please note that the extension port provides direct unbuffered access to vital system signals. Proceed with caution. Do not overload any provided signal with too much additional capacitance. This is particularly important with the system clock, since it already is driving many of the Minimal’s ICs. It is good design practice to buffer the system clock on your extension card prior to using it for driving your own circuit.

VGA Expansion Card

You can add VGA output to your Minimal revision 1.5 by plugging the ‘Minimal VGA Expansion Card’ into the expansion port of the CPU.

Set up your CPU with a 16MHz oscillator, select a system clock frequency of 2MHz (jumper to /8 position) and adjust the serial speed to 125kbps (2MHz/16).

Next, connect the 16MHz (/1) and 4Mhz (/4) signals marked as 'X' in the diagram of the Minimal's 5x2 clock divider to the appropriate socket on the VGA extension.

```
/1    |O  X| - 16MHz
/2    |O  O|
/4    |O  X| - 4MHz
/8    |O--O| - jumper
/16   |O  O|
```

In case you plan on using a modern VGA monitor you are good to go now!

*But before connecting your vintage CRT monitor to this VGA, a word of warning: This minimalistic VGA card generates the necessary blanking intervals from software, relying on *you* to initialize the VRAM to zero *before* switching on your precious vintage CRT. Failing to do so may cause interesting and permanent damage to your CRT since its electron beam will hit spots it's not supposed to hit.*

The CPU interacts with the VGA expansion via two memory locations:

0xdff9 - 0xdffa	16-bit VRAM pointer register (MSB, LSB)
0xdffc	8-bit VRAM I/O register

To store a value 0xff at VRAM index 0x0811 you can either type:

```
0xdff9: 08 11 ENTER
0xdffc: ff ENTER
```

or use the program:

```
LDI 0x08 STA 0xdff9 LDI 0x11 STA 0xdffa LDI 0xff STA 0xdffc
```

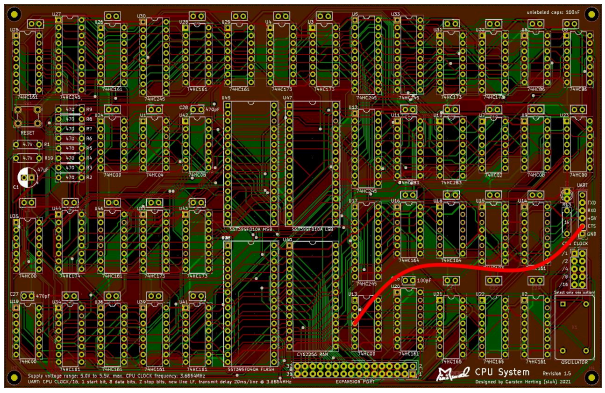
To read back a byte from VRAM just type dffc.dffc ENTER or use the instruction LDA 0xdffc.
I have written several small functions that allow to use the VGA card quite comfortably (see source code for further information on how to call these functions):

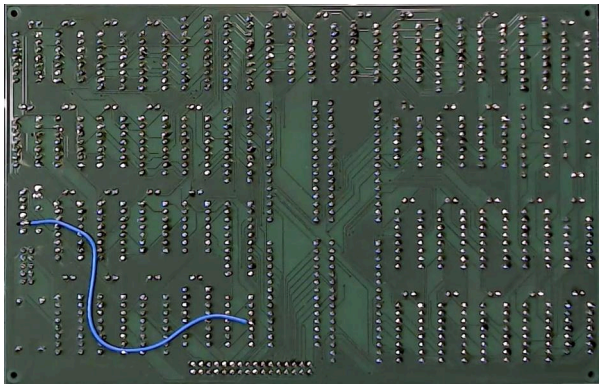
VGA_Init	Erases VRAM (including the blank area) to zero
VGA_Fill	Lets you fill the display area with a byte value
VGA_Print	Lets you print out a text in the display area
VGA_SetPixel	Lets you set pixel at a specific (x|y) position within the display area

These functions comply with the allowed display area of the VGA card.

RTS/CTS Flow Control

By a simple modification as shown below, using RTS/CTS flow control is possible on the Minimal. This gives you higher upload speed since unnecessary line delays can be omitted and additionally allows for cutting and pasting text directly into the native text editor of the Minimal. Since the Minimal is already generating the required CTS signal, only a single bodge wire is needed to feed CTS to the appropriate pin of the USB-to-serial connector. I have made a video about this topic: https://www.youtube.com/watch?v=lwuZ7xYtMKk





Please note that not all USB-to-serial converter chips support RTS/CTS flow control in a way that suits the Minimal CPU. As of today, CH340G works fine while FTDI232 and CP2102 are not suitable due to differences in their hardware implementations.