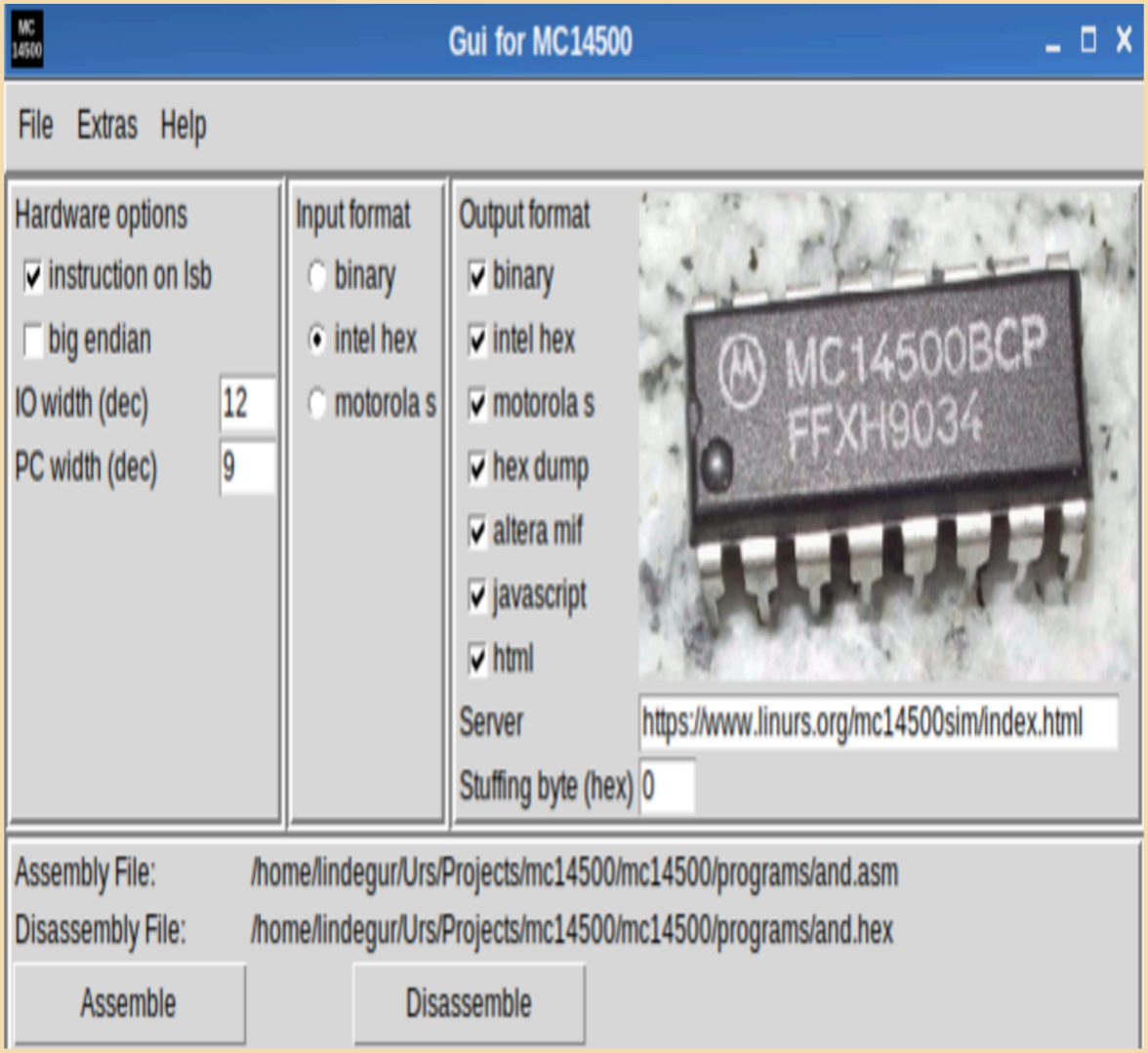

Gui for MC14500
File Extras Help
Hardware options
instruction on lsb
big endian
IO width (dec) 12
PC width (dec) 9
Input format
binary
intel hex
motorola s
Output format
binary
intel hex
motorola s
hex dump
altera mif
javascript
html
MC14500BCP
FFXH9034
Server https://www.linurs.org/mc14500sim/index.html
Stuffing byte (hex) 0
Assembly File: /home/lindegur/Urs/Projects/mc14500/mc14500/programs/and.asm
Disassembly File: /home/lindegur/Urs/Projects/mc14500/mc14500/programs/and.hex
Assemble
Disassemble

MC14500

MC14500

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Copyright

Chapter 2. Foreword

For a customer I started (2009) a redesign for a machine containing a MC14500. A goal was re-using the many MC14500 programs that have been written and being field-proved over many decades. Since the MC14500 is no more in production, a replacement with all its low voltage peripherals has been done using a FPGA. A second step vision was to convert the MC14500 code back to parallel processing VHDL for the FPGA.

In parallel to this activity, I published the stuff not restricted to the non-disclosure agreement on the Internet. Since I'm surprised about the echo from the Internet I'd like to finish a complete MC14500 opens source package.

The MC14500 chip can still be found at https://www.ebay.com

The main goal of this project is having a MC14500 that runs a program assembled by the mc14500 assembler. Other goals are:

- have a hardware around the MC14500 that is state of the art and available long term or be easily portable (Unfortunately FPGA families and their tool chain are complex and are not very stable over time)
- test the assembler and disassembler on a hardware and on a simulator
- reduce the hardware to a minimum
- allow debugging and be flexible for different MC14500 architectures

Chapter 3. MC14500

The MC14500 has different names, the term ICU (Industrial Control Unit) is used, but often the MC14500 is simply called 1 bit processor.

Figure 3.1. MC14500

The MC14500 allows to read an input bit using a IO-address. This bit can be processed using a 4 bit instruction and the internal one bit result register RR. The results can be written to an output bit at an IO-address.

Input and output data bits can be physical inputs and outputs where wires can be attached, but they can be connected to other devices as RAM or timers.

The addressing of the IO's is done completely externally of the MC14500. The 4 bits for the selected instruction of the MC14500 and the IO-address lines for the external IO multiplexer result in the data width of the program memory. The MC14500 used the term "memory word" for the data that comes out of the program memory. This document uses the term "command" as a synonym to "memory word". A command consists therefore of two things, the instruction and the IO-address.

Since the IO-address is externally of the MC14500, the command can be different between the different implementations. Additionally the location of the 4 instruction bits within the command depend also on the design. The instruction bits might occupy the high bits or the low bits within the command.

The MC14500 does not contain the program counter that addresses the command to be processed from the program memory. The number of program counter bits can therefore differ between the MC14500 designs.

The width of the program memory or the command is 4 bit of the instruction plus the number of IO-address lines. Small MC14500 designs will use 8 bit wide program memories being able to select up to 16 IO-addresses. Since this is not much, many MC14500 designs made use of 12 bit wide program memories being able to select up to 256 IO-addresses. 12 bit wide program memories made use of 4 bit wide ROM devices that existed in the past. Other MC14500 designs used 8 bit wide program memories, but two bytes per MC14500 command got read forming therefore a 16 bit wide command and being able to address up to 4096 IO-address lines used for physical IO's, single bit wide RAM and timer hardware. Reading two times from program memory creates an other variation in MC14500 design, the first byte read from the program could be in one design the low byte but in an other design the high byte of the command.

If the low significant byte comes first (the incrementing program counter reads the lower address in the program memory first) then it is a "little endian" (since the little end of the memory word comes first).

If the high significant byte comes first (the incrementing program counter reads the lower address in the program memory first) then it is a "big endian" (since the big end of the memory word comes first).

If the program memory is wide enough (as in FPGA designs and simulators) then the endian topic does not exist.

JMP and RTN

The MC14500 does not support on its own jumps to addresses. In common implementations the jump command will simply reset the program counter and so restart the program and form a main loop. The JMP instruction does therefore not make use of the IO-address.

The RTN command is in most MC14500 not used (it could be used to implement subroutine calls).

More fancy designs might add extra hardware to support jumps to addresses and return features.

NOPO and NOPF

No Operation O and F produce pulses on there pins. A empty program memory (or holes in the program) defaults to either 0x00 or 0xFF depending the technology used and therefore to NOPO or NOPF commands. If the program counter runs over a not programmed program memory section then NOPO or NOPF instructions are executed.

Some designs use the corresponding pins to reset the program counter.

Branch and if

Usually the program counter increments its output from zero until a reset (or over run) of the program counter appears. JMP instruction (or RTN, NOPO, NOPF) might be used by the hardware design to resets the counter to zero.

Therefore branches in the sequence of instructions are not possible. Always the same instruction sequence appears. One exception is SKZ the skip next instruction. SKZ can skip any instruction and therefore also a JMP instruction and prevent a reset of the program counter.

There are features to disable the effects the program sequence does. The most common way is setting the OEN to zero. If OEN is zero the running instruction sequence can not modify any output or RAM. Care is only required for the RR bit that alters during such a sequence.

Obviously disabling and enabling output effects should depend on a condition. Condition how OEN can be manipulated are:

1. Since OEN contains a IO-address, the IO-address from RAM or Input defines if the Outputs get enabled or disabled. (RR might be wired to one of the inputs)

2. The SKZ skip next instruction can be used to skip a OEN instruction

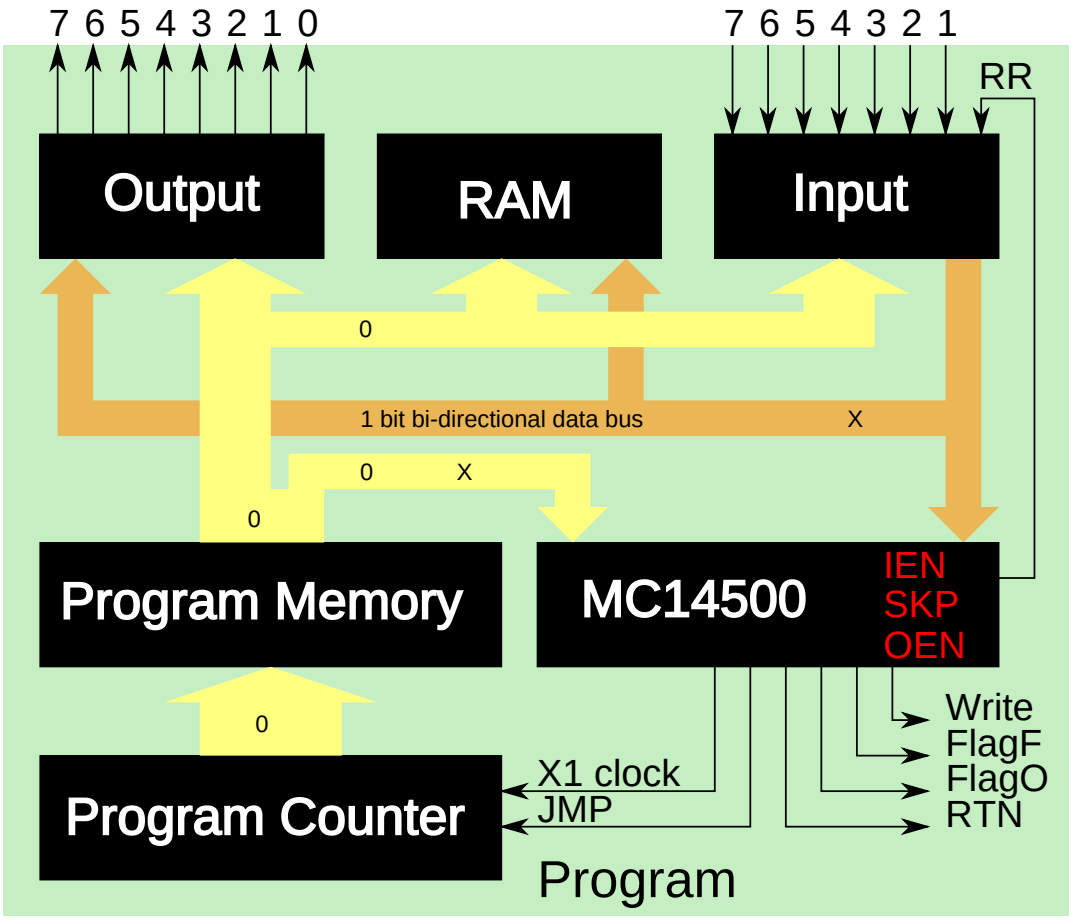
There is also IEN that could be used for similar things. However IEN has some side effects that can manipulate the output or RAM bits. Therefore it is not recommended to be used.

Chapter 4. MC14500 Simulator

The MC14500 simulator uses web technology and javascript to be platform independent and be usable also in the future. A drawback of this implementation is that the javascript simulator can not access the users file-system or hardware.

The simulator runs on browsers with good svg support. The simulator uses the hardware architecture as shown in the following svg figure.

Figure 4.1. MC14500 Simulator



Pressing the "Step" button means falling clock edge and therefore the MC14500 loads the instruction and input data.

Releasing the "Step" button means rising clock edge and therefore MC14500 writes data. The program counter increases. After the program memory access time, the next instruction and IO-address appears at the MC14500 and the Outputs, Inputs or RAM.

The simulator consists of 3 files:

1. `index.html` is the web page to be opened in a browser. It then loads two javascript files. It contains the svg graphic as shown above that is modified using the javascript files.

2. `mc14500.js` is the simulator that runs the program and alters the svg graphics in the `index.html` page.

3. `program.js` holds the program to be run.

The simulator can be stored on web server as it is under https://www.linurs.org/mc14500sim/index.html but it can also run from a local directory.

To run an other program, simply use the **mc14500gui.py** to create html. Then open the created html file in a web browser. There will be a link as https://www.linurs.org/mc14500sim/index.html?rom=60A0B011328821425880117281C0&program=xnor&ilsb=False&iowidth=4&pcwidth=8&endian=little that points to the default web server including the program to be used.

Alternatively produce a `<name>.js` file. Then rename it to `program.js` or better create a link **ln -s <*name*>.js program.js program.js**

Important

The simulator supports ilsb True or False but has a fixed iowidth=4, pcwidth=8 and endian=little.

Chapter 5. MC14500 assembler and disassembler

The assembler and disassembler have been written using python3. The dependencies to other python packages are:

1. bincopy version 14.0.0 https://github.com/eerimoq/bincopy to deal with different file formats as intel hex and motorola S. Newer version have dependencies to other packages.

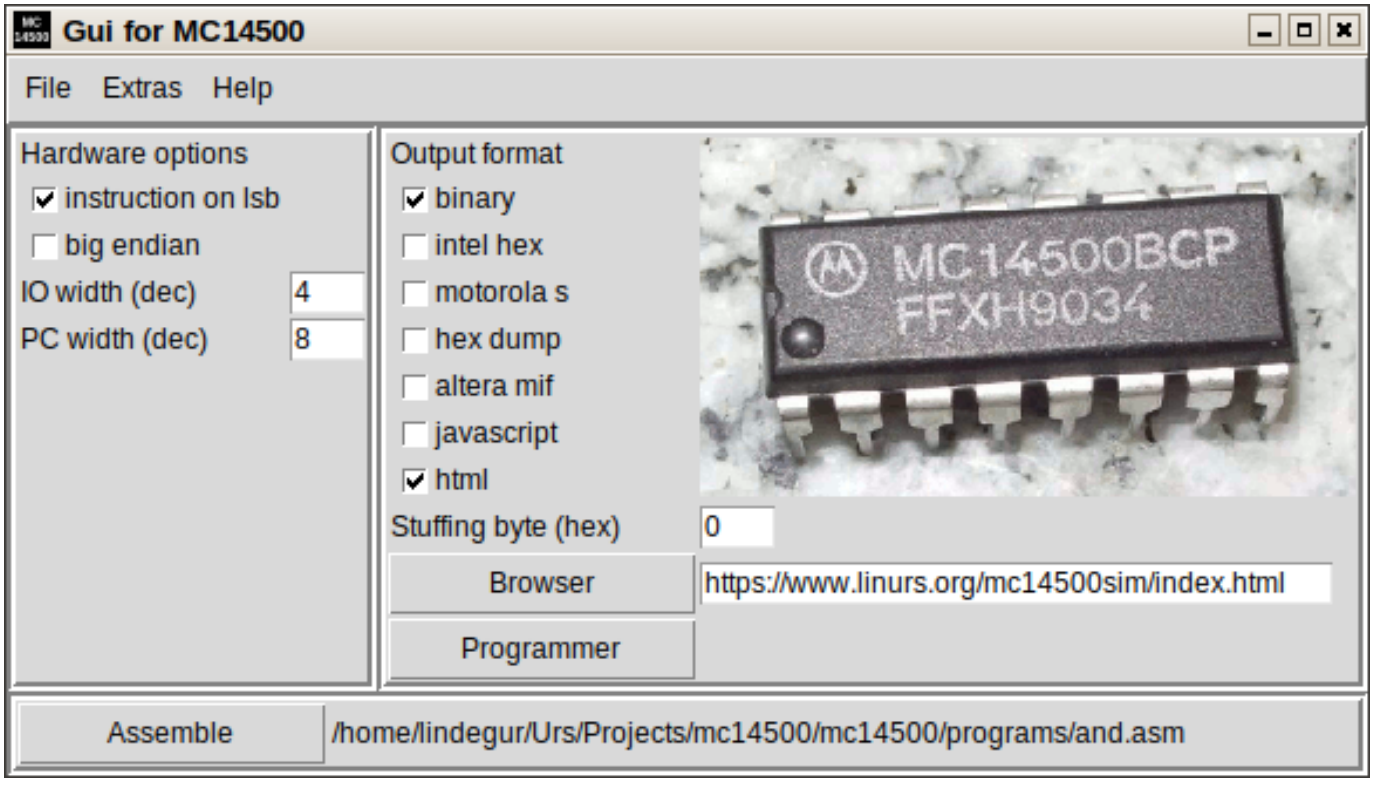
2. pyserial https://github.com/pyserial/pyserial to used to program external hardware

Install them according the systems package manager.

mc14500.py --help or **mc14500.py -h** will show all the command line options for the assembler.

mc14500gui.py is a gui for the assembler.

Figure 5.1. MC14500 Gui



mc14500dis.py --help or **mc14500dis.py -h** will show all the command line options for the disassembler. The disassembler is mainly used to convert old mc14500 program into asm files to be used for the assembler.

The following file extensions are used:

Table 5.1. assembler disassembler file extensions

ASM	Assembly file
DIS	Assembly file produced by the disassembler. It uses an other extension to not accidentally overwrite assembly files

LST	List file

Important

The following options need to be adjusted to the hardware implementation.

1. ilsb defines where the 4bit MC14500 instruction is located in the command.
2. bigendian defines that the program memory uses big endian. Default is little endian. This is important if more than one address is required for a command.
3. iowidth defines the number of IO-address lines
4. pcwidth defines the width of the program counter that is equal to the number of program memory address lines

Important

All numbers for IO-addresses and program counters are interpreted as hexadecimal

As alternative to adjust the hardware using command line parameters the adjustments can be done in the ASM file:

```
OPT  ILSB
OPT  PCwidth 8
OPT  IOwidth 10
```

Aliases can be assigned to IO addresses or other aliases

```
RR     EQU 000  Pin RR is wired to input 0
IN0    EQU RR
IN1    EQU 001
```

Lines containing comments and not being assembled have to start with the * character

```
* this is a comment line
```

Files can be included by using the filename without file extension. Possible file extensions are: ASM, INI, INC. The `init.asm` can be included as

```
INIT
```

The ORG command has been implemented for backward reasons. It allows to set the program counter and have instruction placed on defined locations in the program memory. Since almost every MC14500 design can not jump over memory addresses it makes no sense to use it and it has the potential danger that the program memory can overlap and undefined holes are created. Depending on the output format certain checks are performed that can create warnings and errors.

Chapter 6. Working under Windows

Python and all what is required is usually already installed under Linux but missing under Windows.

Windows exe

Windows exe versions of the assembler, disassembler and gui are therefore an easy to use alternative not requiring to install python at all.

Running python code

Python Version 3.5.3 can be installed as every other Windows program (Other 3 version will work but might be not supported by programs as **pyinstaller** used to create the windows exes).

The module bincopy is required to be installed.

Chapter 7. MC14500 programming

The following makes use of the MC14500 simulator and the mc14500 assembler.

Initialize

After reset typically ones have to be written into the input and output enable registers (IEN & OEN).

No command exist to set the result register RR to a known state.

The OEN and IEN commands can just write the Data signal to the IEN and OEN registers. In most MC14500 designs the result register RR is externally to the MC14500 fed to an input and can therefore be read and used in commands. This means both operands of a logical operation can be the result register RR. To set the result register RR to 1 and then set OEN and IEN to 1 do:

```
ORC RR
IEN RR
OEN RR
```

RR is the IO address where the RR signal is fed to an MC14500 external input.

ANDC RR would set RR to zero.

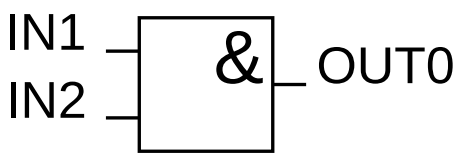
A minimalistic `init.asm` file looks as follows:

```
RR     EQU 000  Pin RR is wired to input 0
IN1    EQU 001
IN2    EQU 002
OUT0   EQU 000
ORC    RR       Set RR to 1
IEN    RR       Enable inputs
OEN    RR       Enable outputs
```

And

The first program `and.asm` reads the two inputs IN1 and IN2, uses the boolean operation AND and finally writes the result to OUT0.

Figure 7.1. And



It includes the `init.asm` file.

```
INIT
LD     IN1
```

```
AND     IN2
STO     OUT0
JMP     X       Resets Program Counter
```

The command **./mc14500.py --html and.asm** produces the `and.html` file that contains the link https://www.linurs.org/mc14500sim/index.html?rom=60A0B0113280C0&program=and&ilsb=False&iowidth=4&pcwidth=8&endian=little that starts the simulator with the program.

Similar commands can be written using the OR and XNOR commands. Inputs can be inverted using LDC, ANDC or ORC. The output can be inverted using STOC instead of STO. This way NAND, NOR and XOR can be made.

RAM

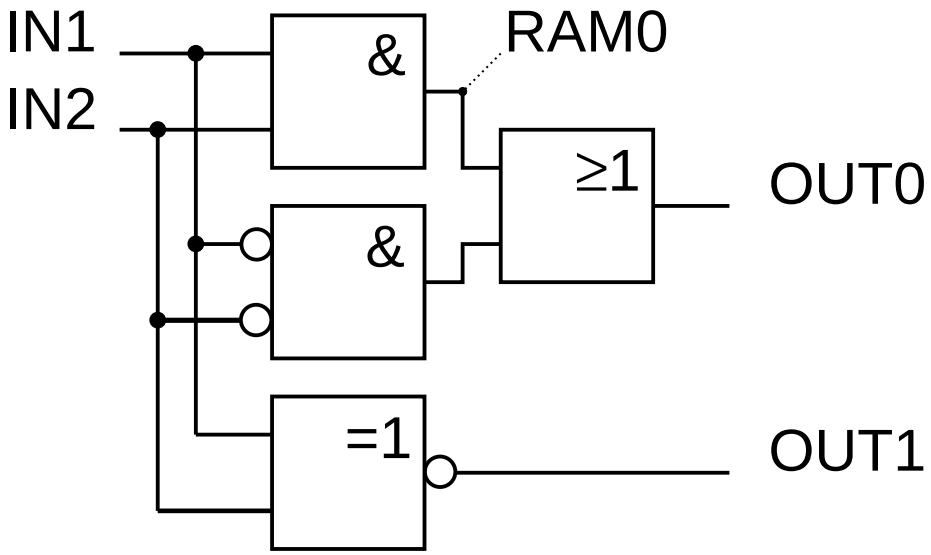
The result register RR is the only single bit memory location of the mc14500. Sometimes a result must be stored and used later. This could be achieved by wiring a physical output to an input, having outputs that can be read back or the more easy way: use RAM.

RAM needs also addresses so the `init.asm` will get lines to add 8 bits of RAM. It is also time to add symbols for all outputs and inputs of the simulator.

```
RR      EQU 000  Pin RR is wired to input 0
IN1     EQU 001
IN2     EQU 002
IN3     EQU 003
IN4     EQU 004
IN5     EQU 005
IN6     EQU 006
IN7     EQU 007

OUT0    EQU 000
OUT1    EQU 001
OUT2    EQU 002
OUT3    EQU 003
OUT4    EQU 004
OUT5    EQU 005
OUT6    EQU 006
OUT7    EQU 007

RAM0    EQU 008
RAM1    EQU 009
RAM2    EQU 00A
RAM3    EQU 00B
RAM4    EQU 00C
RAM5    EQU 00D
RAM6    EQU 00E
RAM7    EQU 00F

ORC     RR      Set RR to 1
IEN     RR      Enable inputs
OEN     RR      Enable outputs
```

The following program `xnor.asm` uses RAM to implement a XNOR function programmed without using the XNOR instruction. IN1 and IN2 will control OUT0. To check if the mc14500 does the same the XNOR function is used to control OUT1.

Figure 7.2. Xnor




```
INIT
LD     IN1
AND    IN2
STO    RAM0  Write Result to RAM
LDC    IN1
ANDC   IN2
OR     RAM0  Read from RAM
STO    OUT0

LD     IN1   Implement the same using the XNOR instruction
XNOR   IN2
STO    OUT1
JMP    X      Resets Program Counter
```

The command **./mc14500.py --html xnor.asm** produces the `xnor.html` file that contains the link https://www.linurs.org/mc14500sim/index.html?rom=60A0B011328821425880117281C0&program=xnor&ilsb=False&iowidth=4&pcwidth=8&endian=little that starts the simulator with the program.

RS Flip Flop

The RS (Reset Set) flip flop has a set input IN1 and a reset input IN2. Once set the OUT1 is 1.

```
INIT
LD     IN1
STO    RAM0
LDC    IN2
XNOR   RAM0
OEN    RR     If 1 then OUT0 changes
LD     RAM0
STO    OUT0
```

```
JMP    X      Resets Program Counter
```

The XNOR function is used to detect if one of the input is 1 and the other 0. If so RR will be 1 and OUT1 might change its status. If both inputs are equal, RR is 0 and OEN 0 disables any write access to OUT0. RAM0 holds the set input IN1 and can be written if IN1 is different from IN2 to OUT1.

The OEN RR line acts as an IF condition.

The command **./mc14500.py --html rs.asm** produces the `rs.html` file that contains the link https://www.linurs.org/mc14500sim/index.html?rom=60A0B011882278B01880C0&program=rs&ilsb=False&iowidth=4&pcwidth=8&endian=little that starts the simulator with the program.

An other approach than run over the instructions that can not modify the outputs is using SKZ and jump over a JMP instruction. If SKZ find RR=1 then the JMP instruction is executed and the loop aborted. If RR=0 the the JMP instruction is skipped and the following instruction updating OUT are processed.

```
INIT
LD     IN1
STO    RAM0
LDC    IN2
XNOR   RAM0
LDC    RR
SKZ    X     If 0 then OUT0 changes
JMP    X
LD     RAM0
STO    OUT0
JMP    X     Resets Program Counter
```

The command **./mc14500.py --html skz.asm** produces the `skz.html` file that contains the link https://www.linurs.org/mc14500sim/index.html?rom=60A0B01188227820E0C01880C0&program=skz&ilsb=False&iowidth=4&pcwidth=8&endian=little that starts the simulator with the program.

ILSB

The 4bit instruction is placed per default to the msb (most significant bits) in the command

```
00 60 ORC    RR       Set RR to 1
01 A0 IEN    RR       Enable inputs
02 B0 OEN    RR       Enable outputs
```

Calling the assembler with the **-i** (or **--ilsb**) option or putting

```
OPT ILSB
```

to the assembly file moves the instruction to the lsb (least significant bits) in the command

```
00 06 ORC    RR       Set RR to 1
01 0A IEN    RR       Enable inputs
02 0B OEN    RR       Enable outputs
```

PCwidth

The program counter width (PCwidth) has a minor role in the assembly. It defines the width of the first column in the `<file>.lst`. Additionally it lets the simulator know when the program counter overruns (gore back to zero).

Example 7.1. PCwidth 16bit

```
0000 60 ORC    RR       Set RR to 1
0001 A0 IEN    RR       Enable inputs
0002 B0 OEN    RR       Enable outputs
```

The PCwidth can be set with the **-p** (**--pcwidth**) option or putting

```
OPT PCWIDTH 8
```

to the assembly file.

Important

PCwidth is a decimal value

IOwidth

IOwidth is the number of input and output address lines (used for input and outputs but also for RAM).

The IOwidth can be set with the **-w** (**--iowidth**) option or putting

```
OPT IOWIDTH 12
```

Common IOwith values are 4 (to have a 8 bit wide program memory) or 12 (to have a 16 bit wide program memory)

Important

IOwidth is a decimal value

Big Endian

Since the program memory width is not required for the assembler and disassembler the endian types (little or big) are not required to be known. However if the resulting binary needs to be converted into a 8 bit wide output format (as bin, motorolas, intelhex, hexdump), then this must be known. Consequently if the output format is in respect to the program memory flexible (as mif, js, html), FPGA, then endian is no issue and can be ignored.

As default the assembler and disassembler assume it is little endian.

If not then the assembler or disassembler can be called with the **-g** (or **--bigendian**) option.

Note

Since little and big endian is just required for certain binary formats, no option exist to add this to the assembly file.

Chapter 8. Physical MC14500 implementation

Hardware issues

Something not very well documented is a possible bus contention (fight) on the bidirectional data line of the MC14500 or its attached peripherals during the change of the write signal. Bus contention occurs when more than one chip drives the data line at the same time with probably a different signal level. It can be safely assumed that the MC14500 chip designers had the timing inside the chip under control. So it is just an chip external issue. Different solution exist or are possible:

1. Some designs simply ignore it

2. A serial resistor could be added to limit the current to the specified level in case a bus fight occurs

3. Some designs use a OR-gate feed by X1 and Write and use then the output for the OE~ of the peripheral chips. This might be a good solution, especially if the OR-gate adds some delay.

The datasheet gives values for "X1 to Data" and "X1 to Write". It is not clear if this means X1 falling or rising or if it is valid for both edges. Additionally no minimum values are defined. Looking at the typical values for Vcc = 5V shows that the Write line rises before the Data is valid.

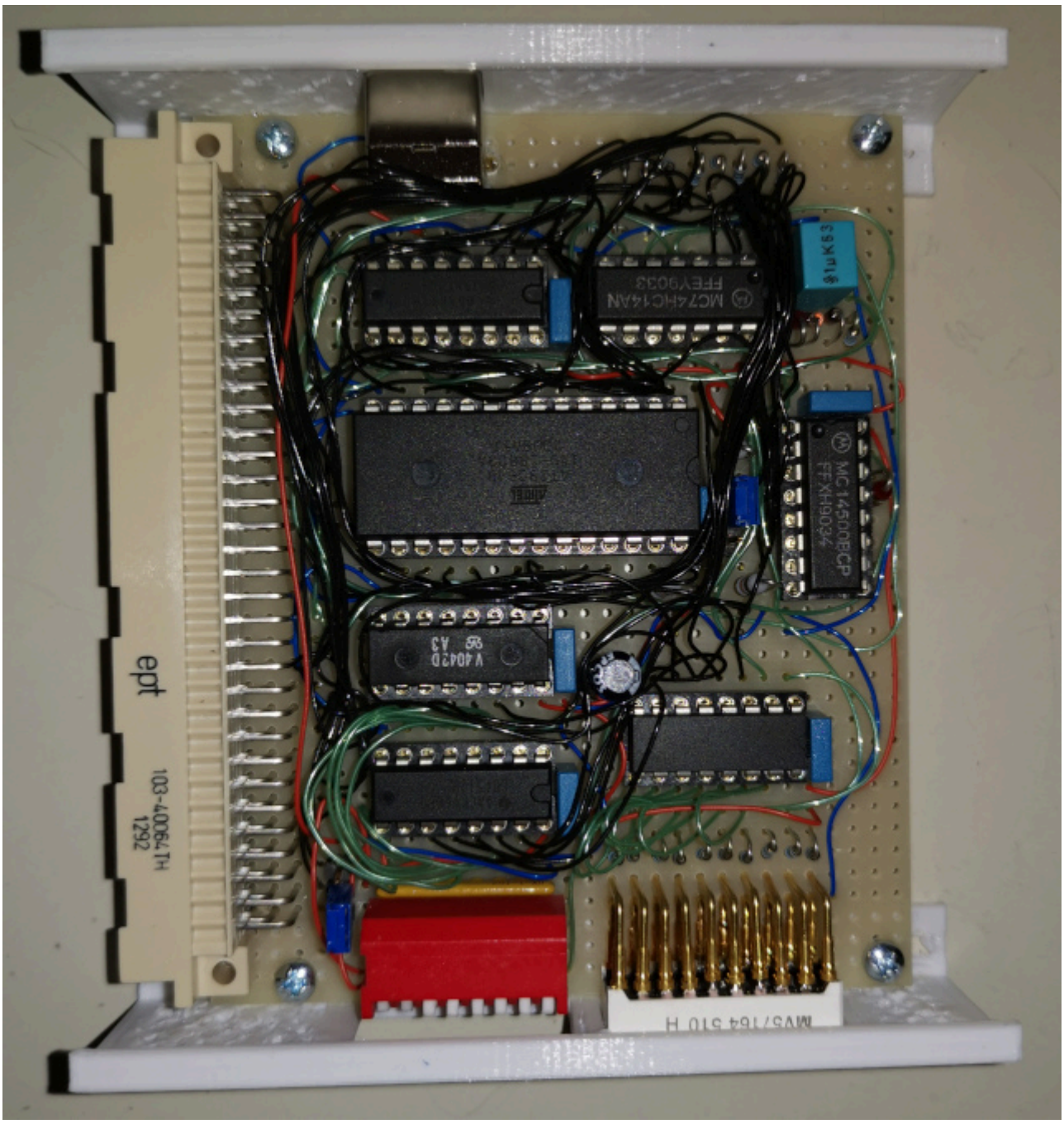
MC14500 CPU board

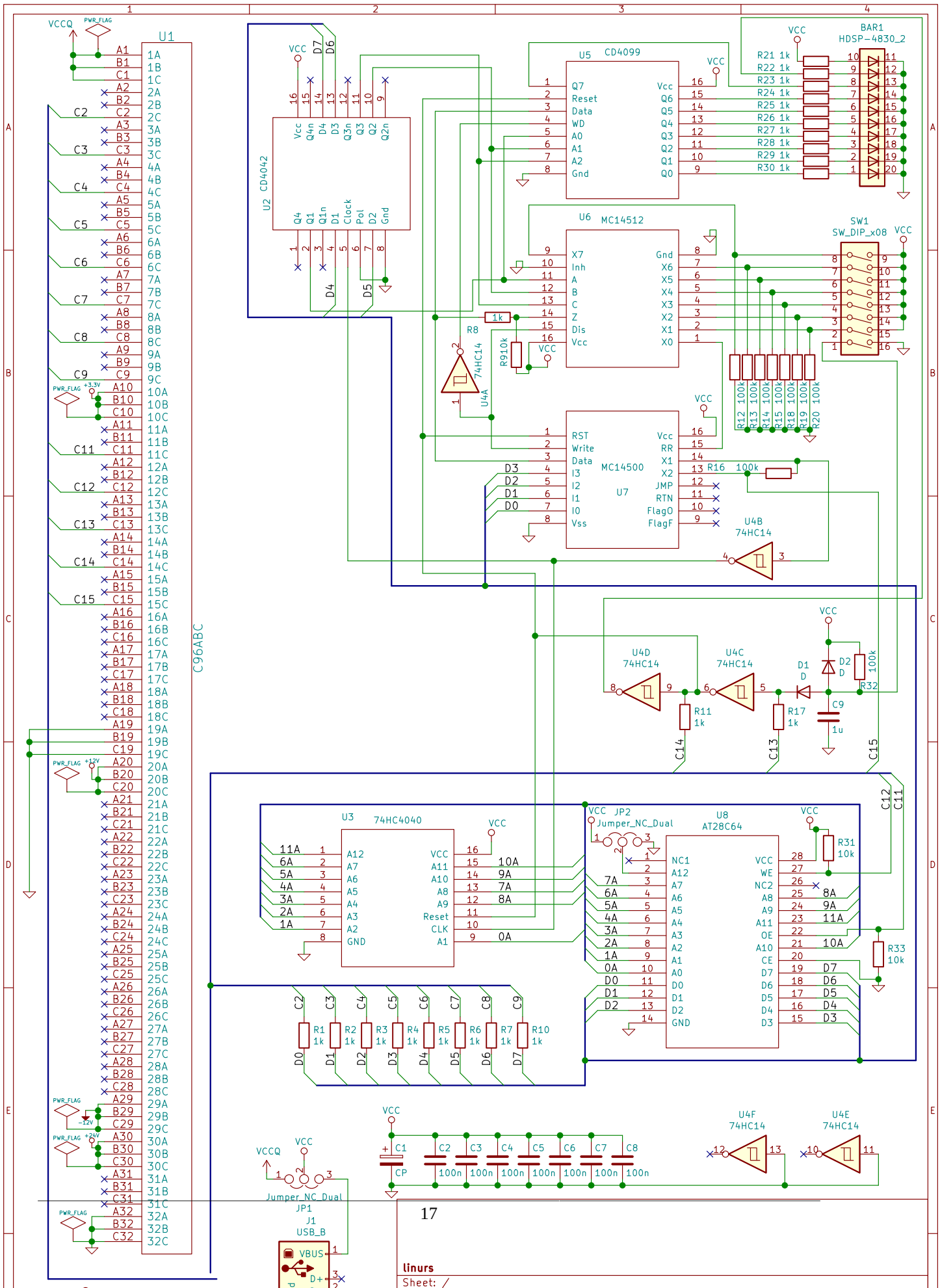
The implementation uses synchronous logic and avoids dependencies on asynchronous delays. Therefore all is based on the clock. The MC14500 latches the 4 instruction bits on the falling clock edge. The rising clock is used by the MC14500 to read and write the IO's. So the program counter can update the ROM on the rising clock edge. The IO-address from the ROM changes also on the rising clock edge and a IO-address is required to select the IO to be read or written. To have a synchronous design and not run into timing issues the IO-address from the ROM is latched on the falling edge into D-Flip-Flops. So the IO-address applied to the IO's does not change while the MC14500 reads or writes the IO's.

To reduces the number of chips, the program counter is free running and can not be reseted by the MC14500. This gives a constant cycle time for the program. Constant cycle times are common on SPS. The cycle time is small enough so a human being will not notice a reaction delay.

The timing regarding the data output is not well specified in the MC14500 datasheet. To prevent an eventually bus fight that could damage the MC14500 a 1k resistor is added between MC14500 and the input chip. If a bus fight would occur then the current will be limited.

The design allows too be in-circuit programmed and controlled by a daughter board. Again there could be some collision between outputs so all potential dangerous signals are feed via 1k resistors to the connector.


MC74HC14AN
MC14500BCP
ept
103-400641
1292


U1
C96ABC
U5 CD4099
U2 CD4042
U6 MC14512
U7 MC14500
U3 74HC4040
U8 AT28C64
JP2 Jumper_NC_Dual
JP1 Jumper_NC_Dual
J1 USB_B
BAR1 HDSP-4830_2
SW1 SW_DIP_x08
U4A 74HC14
U4B 74HC14
U4C 74HC14
U4D 74HC14
U4E 74HC14
U4F 74HC14
VCC
VCCQ
PWR_FLAG
linurs
Sheet: /

Avr board

The MC14500 CPU board runs autonomously and its EEPROM chip could be removed and programmed. The AVR board allows more modern features as in circuit programming of the EEPROM via a virtual COM connection on USB.

The commands are pure ASCII, therefore it will work with a standard terminal program as **minicom**. Sending the **cat** command lists all commands implemented.

Relevant commands are:

Table 8.1. Avr Commands

rst	rst 1 puts the board in reset condition. rst 0 clears it and rst returns the rst status
data	data reads the data from the EEPROM. data <0..255> writes the data onto the EEPROM
clk_cpu	sets clock to low, so instruction data is latched to the CPU
clk_pc	sets clock to high and increases the programm counter
latch_mem	write pulse to the EEPROM to store the data
dir_read	The EEPROM puts its data on its pins
dir_write	The EEPROM reads data from its pins
clk_run	The AVR board toggles the clock line, so the program runs. clk_cpu or clk_pc will stop this.

Chapter 9. FPGA implementation

The FPGA implementation of the MC14500 CPU used a Altera Cyclone III FPGA and the Quartus II development environment. Unfortunately both the Cyclone III FPGA and Quartus II are now outdated. Cyclone III are no more supported by newest Altera development environment.

Features of the MC14500 CPU are:

1. Cyclone FPGA EP3C5E144C8 in the rather small QFP 144 case
2. 16 inputs plus 16 outputs that can be read back
3. 16 bit wide ROM (2kByte) containing the program
4. High speed
5. All important signals available on the pins
6. Embarrassing how little of the chips internal logic is used, leaving space for much more
7. 1 bit wide RAM to store temporary data
8. All CPU signals are fed to pins of the FPGA

Figure 9.1. FPGA implementation

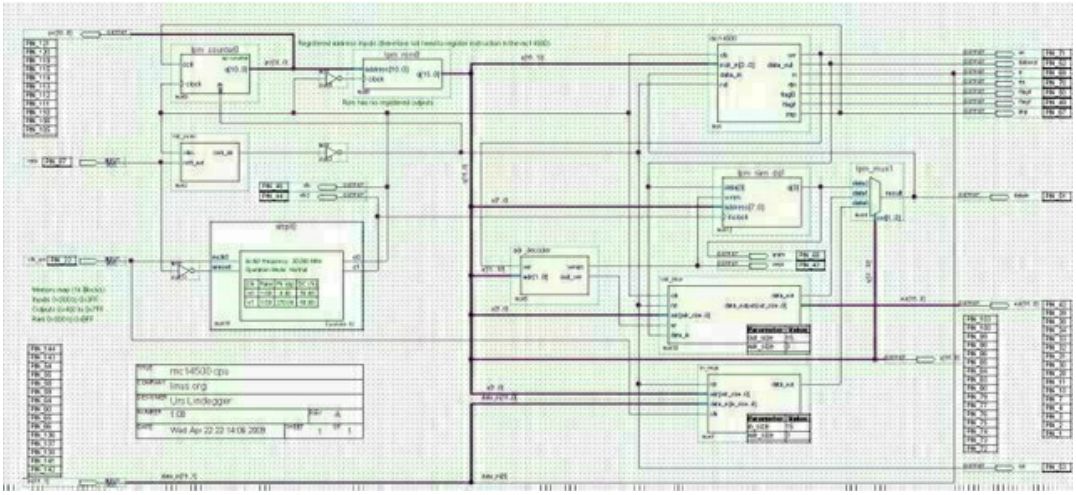


Figure 9.2. Simulation of a program

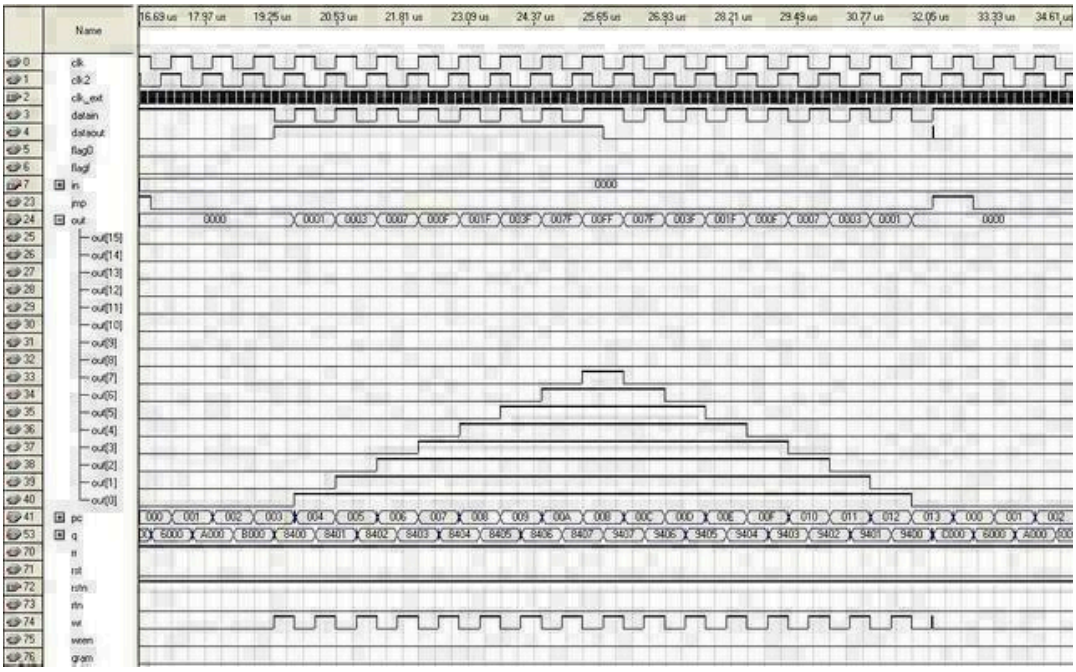
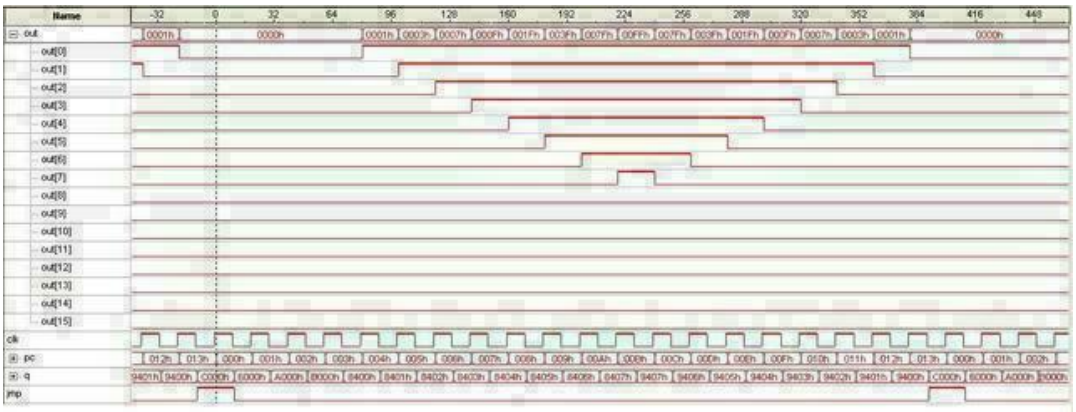


Figure 9.3. The FPGA hardware

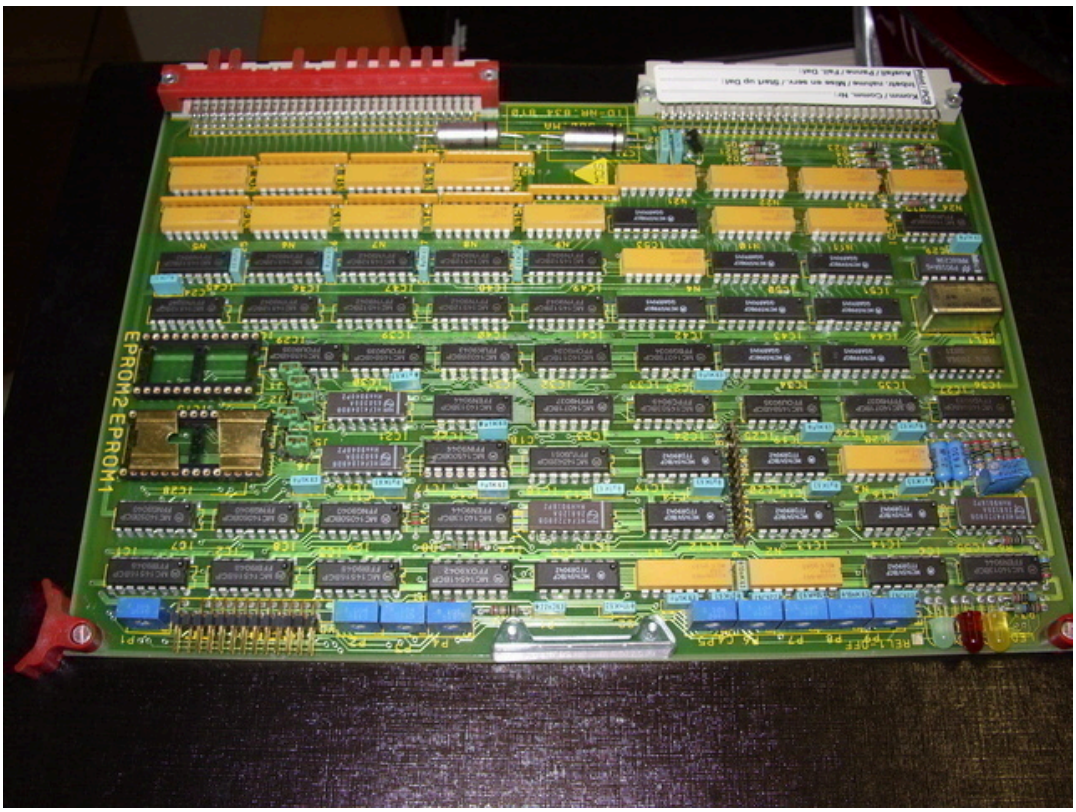
The program running in the real hardware can be analyzed with the FPGA embedded logic analyzer.

Figure 9.4. FPGA Logic analyzer



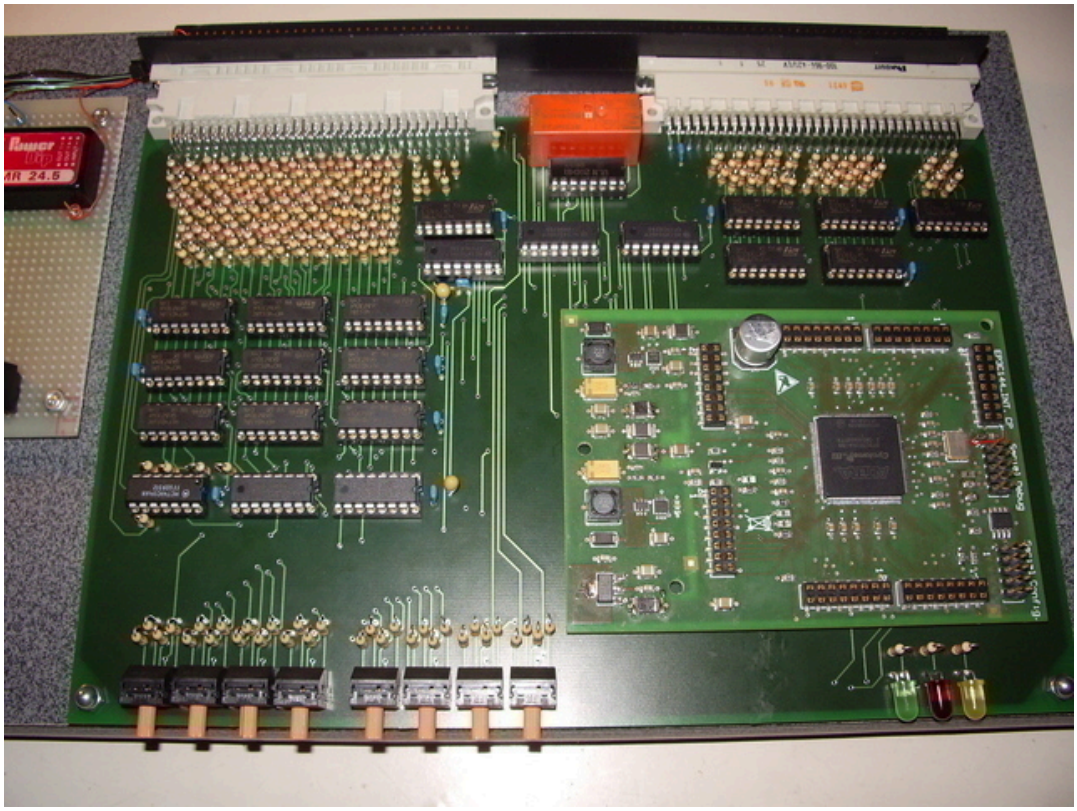
Chapter 10. Redesign

Figure 10.1. MC14500 CPU board



The redesign moved all the not 15V CMOS logic into the FPGA. The analog timers using potentiometers got replaced by hex switches controlling digital timers in the FPGA. The original design used two 8bit EPROM access cycles to get a command for the MC14500 and its IO-address. The FPGA implementation used a 16bit wide memory to get it with a single access. The big debug connector got not implemented, since the FPGA allows to implementing an internal logic analyzer that is obviously much more powerful.

Figure 10.2. Replacement CPU board



Chapter 11. Links

1. http://tinymicros.com/wiki/MC14500B A MC14500 wiki page containing links to original MC14500 data sheet and handbook
2. http://wdr-1-bit-computer.talentraspel.de/ a German site of a learning computer using the MC14500
3. http://www.6502.org/users/dieter/m14500/m14500.htm a rebuild of the MC14500
4. https://github.com/nicolacimmino/PLC-14500 nice and complete rebuild of a MC14500 PLC