First experiences with Microcore

N.J. Nelson, C. Williams

Abstract

Following the convincing demonstrations at EuroForth 2004, we decided to use the "Microcore" VHDL Forth processor in the design of three new products. This paper will describe our progress in expanding the core design with additional peripherals, performing simulation, board implementation, and early experiments in writing code on the Microcore.

N.J. Nelson B.Sc., C.Eng., M.I.E.E.
Micross Electronics Ltd.,
Units 4-5, Great Western Court,
Ross-on-Wye, Herefordshire.
HR9 7XP U.K.
Tel. +44 1989 768080
Fax. +44 1989 768163
Email. njn@micross.co.uk

C. Williams B.Sc., C.Eng., M.I.E.E.
Chrysalis Design,
Craig-y-don,
Llandinam,
Powys
SY17 5BG
Tel. / Fax. +44 1686 688065
Email. chris@chrydesn.demon.co.uk

1. Overview of Microcore

Microcore is a VHDL description of a microcontroller, which can be implemented in an FPGA. It is highly configurable, and in particular, the external data path width is a compilation variable, so that various different "sizes" of the same processor may be constructed, with different performance / cost balances. The code for Microcore is available under a licence which is similar to open source software, and which encourages other to contribute to the project development while retaining compatibility and openness.

Microcore was first described by Klaus Schleisiek at the 17th EuroForth at Dagstuhl, and he also described an implementation of the device at the 20th conference, where he gave a convincing demonstration of the technology.

Microcore has its own website from which the code may be downloaded.

2. Reasons for choosing Microcore

Advantages unique to Microcore

a) It's free. This is a serious consideration for a small company where development budgets are tight.
b) It comes from a known and trusted developer. Klaus also designed the IX1 microcontroller which has given us years of completely trouble-free service.
c) You can actually talk to the designer, who even answers email and telephone calls. This is in marked contrast to other offerings of standard cores.
d) Genuine futureproofing
Even if the hardware goes obsolete, the software won't. There should be no difficulty in moving a Microcore project to a future FPGA technology. The struggle to buy "one careful previous owner" RTX chips will be over.
e) Control
We have all the code to produce versions of Microcore for as long as we need to.
f) No black boxes
If there is a problem, nothing is hidden. We can analyse the problem to whatever depth is required.
g) Simple and inexpensive design tools
We have used Xilinx and Mentor Graphics tools.
h) Different sizes, same code
We can use exactly the same software on both 8 bit and 32 bit external data bus width versions.
j) Simplicity
We almost understand quite a bit of it.
k) It runs Forth
All of us understand it, and with careful core design it should be possible to port large chunks of our existing code straight in.

Advantages of FPGA microcontrollers over fixed hardware

l) Potential for future performance enhancement
As the speed of FPGAs increases, so will the speed of Microcore.

m) Extensibility
On-chip peripherals can be added relatively easily.
n) Pinout flexibility
Pinouts can be matched to the PCB layout requirements, enabling a simpler and less expensive 4 layer PCB to be used. Without this, a 6 layer PCB would almost certainly be needed.

3. Our particular requirements

We needed to replace and upgrade three products.
a) The Virtual Programmable Logic Controller
This is a high integrity device which provides the central control functions of a distributed automation system. We described this card at EuroForth 97. It uses the RTX2001 as its CPU, and has a PCI interface with a PC but is otherwise completely autonomous. This has been a very satisfactory design with excellent reliability.
b) The Rapid Automated Bacterial Impedance Technique (RABIT), also a PCI card but this time designed as a centralised data logger for a large number of distributed microbiological tests cells.
c) The RABIT block module, which provides ultra-accurate temperature control and digitisation of a group of 32 microbiological test cells.

Both RABIT circuits used the Intel 251 microcontroller, which is possibly the worst microcontroller ever produced. We shall be very glad to replace it.
The new versions of both a) and b) are very similar, using 32 bit data bus widths and an Ethernet connection to the PC instead of a PCI connection. They have differing power supply, communication and memory requirements.
The new version of c) will be an 8 bit implementation.

4. Experiences with the tools

Design philosophy

Our basic design philosophy is to make it simple and to use as much of the Microcore design as possible. We don't want to have to dig deep into the VHDL to understand it and by doing a conservative design where we keep a respectful distance from the limits we hope to reduce our problems. We also need to remember that the number of boards that we will make is quite small, and that the cost of the development tools must be kept to a minimum.

The chips used in the design also affect the tools. Each chip vendor has its own tool set for which it is optimised, but limited to its own ICs, this includes both Lattice and Xilinx . By choosing a class of chip that has already been used to implement Microcore other potential pitfalls may be reduced. Microcore has already been implemented in the Xilinx Spartan series of chips. These are currently cheaper than the Lattice parts, but they do require an external flash memory to initialise them. We want to reduce manufacturing problems so we don't want ball grid array packages. We also want a part that will give room for experimentation in the future.

The XC3S400 PQ208 is a Xilinx Spartan 3 device in a 208 pin plastic quad flat pack and it will accept the Microcore with room for expansion and the additional peripherals that we need. A cheap programmer is available for transferring the compiled output on the computer to the flash memory on the board and modifying the design as often as required.

Choice of tools

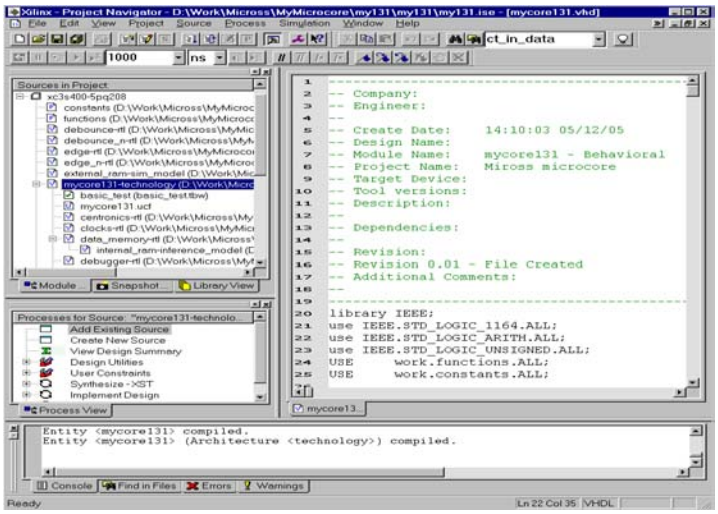
There are two possible tool sets we could use, the Xilinx ISE (Integrated Software Environment), or the Synplify system from Synplicity. The pros and cons of each are as follows:

Xilinx ISE	Synplify
Free	~£10,000
Complete design from beginning to end	Works with Xilinx tools
Limited optimisation increases chip area used.	Advanced optimisation gives smallest possible design
Design may not give maximum possible speed.	Advanced optimisation may give fastest design

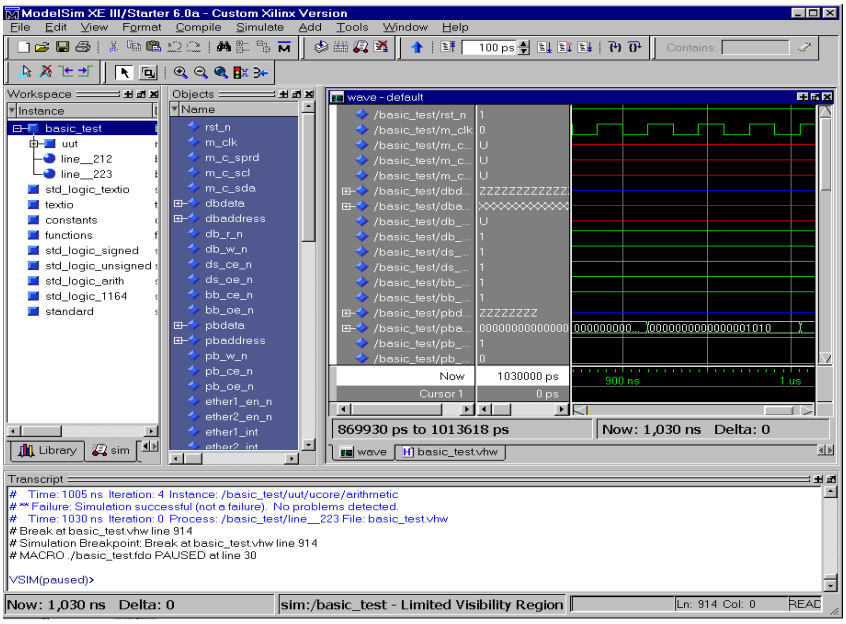
The ModelSim simulator from Mentor Graphics is provided to simulate the designs at all levels. This accepts a VHDL description that can be functional, i.e. no timing information, and allows the VHDL to be checked for accuracy, right up to a full post layout description that gives detailed operations and timings.

We chose to use the Xilinx ISE foundation pack that can be downloaded free from the Xilinx web site. By not pushing the design to its limits we hope that the reduced optimisation will not cause a problem. The huge reduction in cost is also more in line with the number of chips we are likely to produce.

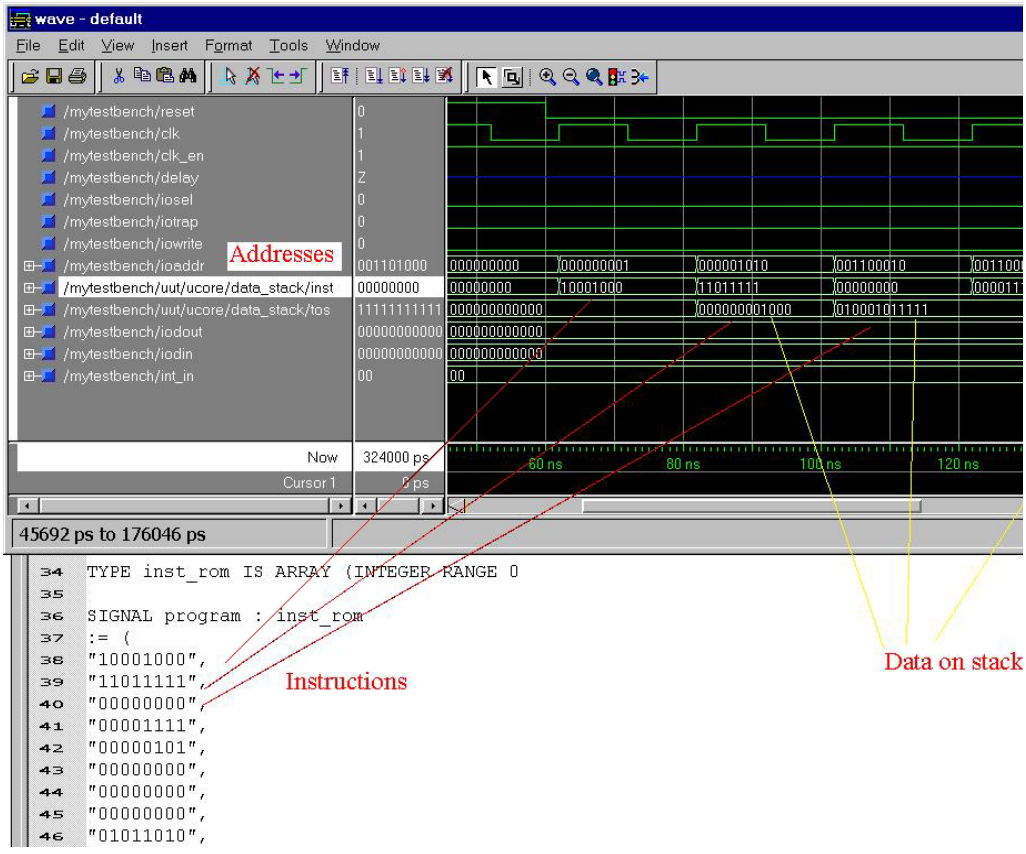
Here is a typical screen for the ISE version 7:
This shows a project window where all the files can be entered, a process window where for each file all the possible actions are listed, an edit window where all the files can be viewed and edited and a console where progress and errors are displayed.



From here, all the different operations needed to build a design are managed. One obvious operation is to run the simulator. By selecting the test bench file, which contains waveforms, you have the option to run the simulator directly. A typical screen shot:



This again shows a multi-window screen with the waveform result on the right.
The first job in evaluating the tools and the microcore design was to try and run a functional simulation.



This shows the first instructions in the boot memory being run at the end of reset and loading immediate data onto the stack. The first 2 instructions have the top bit set that then loads the following 7 bits onto the stack.

5. Peripherals we have developed

At this stage of using Microcore we need two additional peripherals:

- Watchdog counter
- SPI serial interface

Watchdog counter.

The watchdog counter counts a period of time, using the master input clock as its reference and if it is not reset in that period by the software it causes a processor reset. We require a timeout of 1ms and can set this directly into the hardware based on the processor master clock.

The VHDL code is as follows:

```
watchdog_reload <= '1' when sel_io = '1' AND
(addr(watchdog_select_address_bit) = '1') else '0';

watchdog_control : process(m_clk,rst_n,watchdog_reload,watchdog_div)
BEGIN
      if(rst_n = '0') then
            --On reset set a slightly longer watchdog time
            watchdog_div <= (OTHERS => '1');
      else
            if (rising_edge(m_clk)) then
                  if watchdog_reload = '1' then
                        watchdog_div <= "11000011010100000000"; --200000
                  else if two_meg_div = "0000" then
                              watchdog_div <= watchdog_div - 1;
                        end if;
                  end if;
            end if;
      end if;

END PROCESS watchdog_control;
```

This describes a simple down counter watchdog_div that is decremented on every rising edge clock with the code:

```
      watchdog_div <= watchdog_div - 1;
```

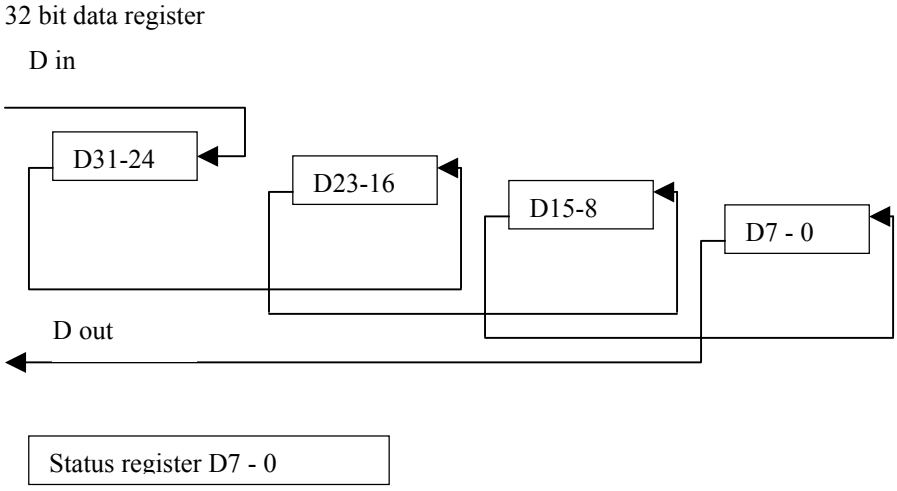
This is modified if we have a reset signal or a watchdog_reload signal. The watchdog reload signal comes from a memory access instruction from the processor to the watchdog address.

The microcore reset generator then looks at both the external reset signal and the value of watchdog_div. If watchdog_div ever gets to a value of ‘0’, the processor is reset, and can start again.

SPI serial interface

The SPI (Serial Peripheral Interface) is more complex than this. It uses a clock line, two data lines (one input and one output) and a chip enable to provide bidirectional data transfer, and can be used to talk to a wide range of chips. We currently need to communicate with a serial flash memory to store our programs. Normally, data transfers are in 8 bit bytes, but we have made good use of the 32 bit data path to allow up to 4 bytes to be transferred at a time without processor intervention.

The general hardware arrangement is as follows:



Bit	Function	Status register	
0	Int	Read, cleared when written	
1	Start	Write, cleared when done	
2	B0	Write	Byte count to transfer
3	B1	Write	
4	CE1	Write	Chip enable 1
5	CE2	Write	Chip enable 2

The 32 bit shift register sends data out from the low order byte, and reads data in through the high order byte. The status register, a memory location in the I/O memory area holds 5 bits to control the operation:

- An interrupt bit to indicate when a transfer is complete.
- A start bit, set by the user to start a transfer and cleared automatically when the transfer is complete.
- A two bit count of the number of bytes to be transferred, set by the processor.
- A pair of chip enables, passed directly to the devices, set by the processor.

The VHDL for this has been developed as a separate module. This contains the status register, all the shift registers and counters. It is controlled by a hardware state machine implemented as follows:

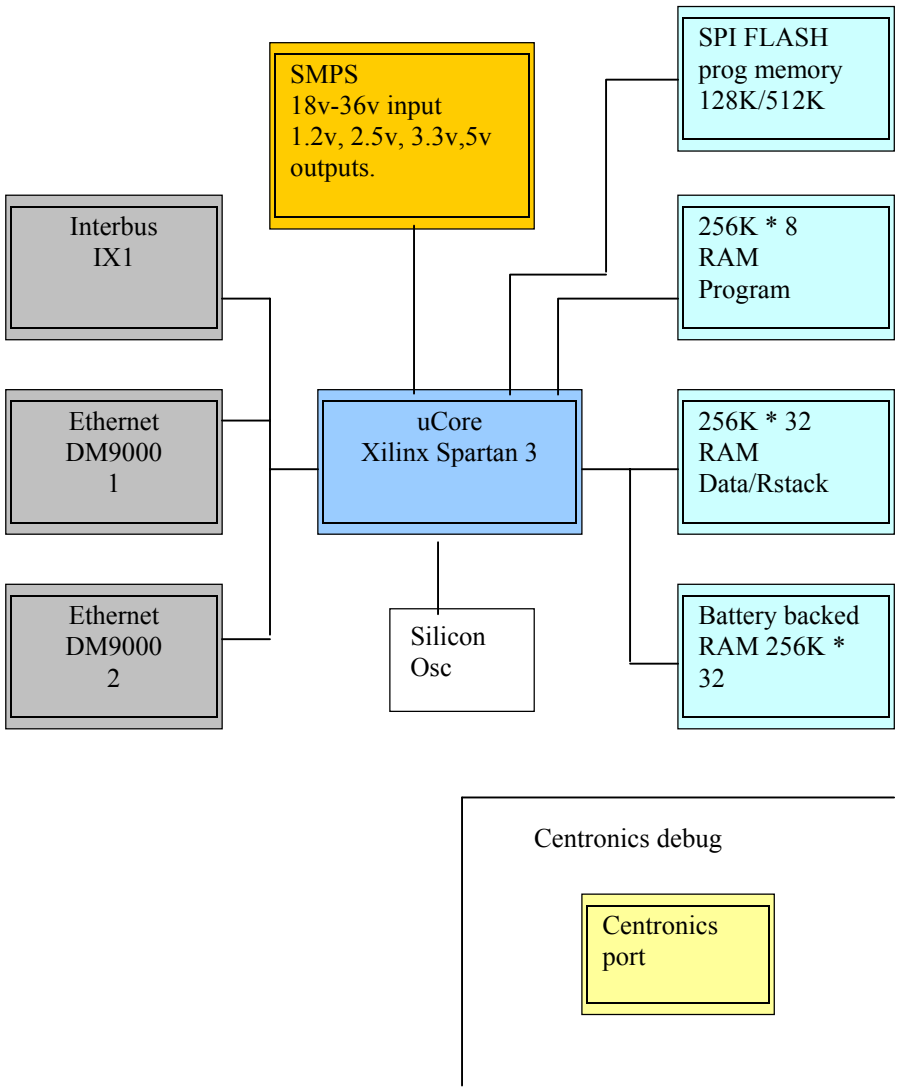
```
State_machine: process(reset,state,status_reg,shift_clock)
BEGIN
   if reset = '1' then
            state <= waiting;
            bit_counter <= "000000";
       elsif falling_edge(shift_clock) then
            case state is
                  when waiting => if status_reg(start_bit) = '1' then
                        state <= running;
                        if status_reg(byte_count_1_bit downto
byte_count_0_bit) = "00" then
                              bit_counter <= "000111";
                        elsif status_reg(byte_count_1_bit downto
byte_count_0_bit) = "01" then
                              bit_counter <= "001111";
                        elsif status_reg(byte_count_1_bit downto
byte_count_0_bit) = "10" then
                              bit_counter <= "010111";
                        else
                              bit_counter <= "011111";
                        end if;
                  end if;
                  when running => bit_counter <= bit_counter - 1;
                        if bit_counter = 0 then
                              state <= waiting;
                        end if;
                  when others => state <= waiting;
            end case;
       end if;
END PROCESS state_machine;
```

As you can see, the naming conventions and appearance are much closer to ‘C’ than to Forth, but you can also see that it is using its current ‘state’ which can be ‘waiting’ or ‘running’ along with the start_bit in the status register and the bit counter to control its operation.

In use you load the data to be transferred into the shift registers, set the start_bit, the byte count and the chip enable, and wait either for the interrupt or by poling the start bit for the end of the transfer. Any data read back from the device can then be read into the program from the shift registers.

6. The first hardware design

This is the block diagram of the complete system that we are building:



The most important question to ask at the start of the design is at what voltage the chips will run. The Xilinx chip requires 1.2V and 2.5V for its internal operation but will interface to the outside world at any voltage up to 3.3V. Looking at the chips around it, some are available at 3V, some at 3.3V and others at 5V.

In this case most chips are available for 3.3V operation, except for the IX1 chip that is only available at 5V. This meant providing a separate 5V supply using voltage converters on the signals to and from the Microcore.

The next question to ask is how the memory is to be organised. We need 32 bit wide RAM for stack and data, but we also need an area of battery backed ram for long term storage. The memory needs to be 10ns to run at full speed and we could not find memory that fast that was also low power enough to be battery backed. Our solution is to have both kinds of memory, with the 55ns battery backed ram requiring 2 cycles for access.

The program memory is only 8 bits wide but requires the same compromises. It needs to be fast RAM and non-volatile. We could not find anything to do this. We compromised with a fast RAM chip and a slow serial flash memory. The Microcore can be made to write to its program memory, so at boot time, running the internal boot loader, the code in the flash memory can be read out and written to the RAM. The program then jumps to the start of the RAM. The flash memory can be written by the program as well.

The external peripherals that we need for the application are placed on the memory bus. The Centronics debug port comes straight from the Microcore design and is used in initial development for programme load and debugging. The master clock is a silicon oscillator, which is an alternative to a crystal. This has the advantage both of size and its ability to 'jitter' slightly. This does not affect the operation of the Microcore, but it does reduce the electromagnetic interference and that helps with technical approvals.

The design was started using version 1.30 of the Microcore. Part way through the process 1.31 became available. This has some significant differences and required some changes to our designs. Then 1.32 became available. The rapid changes can cause problems in the design. It is better to stay with a version until its limitations cause real problems rather than change every time a new version is available.

7. First steps in software development

We expect to have some hardware to show in time for the conference. With luck, a little software might even have been written.

8. Conclusions

We'll tell you next year!