Simulator ModelSim 5.3b (VHDL93 = 0, Explicit = 1)

Synthesizer Synplify 5.14

Architecture Alternative architectures have been realised using different vhdl-Files, which have to be compiled/loaded depending on the design choices due to synthesiser limitations. You have to take care of editing your own load/make file which tends to be tool specific. The source files have been listed in compilation order.

1 VHDL Source files

Filename	Description
functions.vhd constants.vhd	Packages used throughout the project. Architecture parameters, general types and Instruction set
dstack.vhd	Data Stack Entity including RAM model for stack RAM
alu.vhd alu3state	Arithmetic-Logic Unit with pre-incrementing data memory access ALU using 3state logic for the ALU input multiplexer. This may be more efficient for some FPGA families.
data_ram.vhd syn_data_ram.vhd	Asynchronous Data and Return Stack memory Synchronous Data and Return Stack memory. Added in order to support RAM blocks inside FPGAs. These tend to be synchronous RAMs.
rstack.vhd	Return Stack Entity
program.vhd debug_program.vhd	Generated by the cross-compiler from Forth source code. Depending on the settings in the cross-compiler load file, this "program ROM" may either be modeled as a ROM-array or as a big CASE construct.
sequencer.vhd	Instruction address generator
interrupt.vhd	Interrupt conditioning
uBus.vhd uBus3state.vhd	Multiplexed uCore Bus uCore Bus using three-state buffers. This version may be more efficient to implement depending on the FPGA family. Please note the "bus-holdoff" signal TRI_EN that has to be generated.
io.vhd	uCore's memory mapped input and output. This entity also holds internal registers which are addressed via memory mapped I/O, e.g. the TASK register
uCore.vhd	Structure of uCore, technology independent
core.vhd bench.vhd	Version with both program memory ROM and data memory RAM inside the FPGA. CORE holds all technology and project dependent information as e.g. pin assignments. BENCH is the matching test bench.
debug_centronics.vhd debug_core.vhd debug_rom.vhd debug_bench.vhd	Debug version of MicroCore. See text below for explanations on the debug philosophy.
rom_ram_core.vhd rom_ram_bench.vhd	Version with external program ROM and external, asynchronous data stack RAM.
rom_syn_core.vhd rom_syn_bench.vhd	Version with external program ROM and internal synchronous data memory RAM for compilation into RAM blocks.
syn_core.vhd syn_bench.vhd	Version with both program memory ROM and synchronous data memory RAM inside the FPGA. The latter uses RAM blocks if available.

2 Using synchronous block RAMs

Some FPGA families do have synchronous RAM blocks as a resource. SYN_DATA_RAM.VHD can be used to model the RAM appropriately. There is a drawback though: Microcore has been realised with asynchronous RAM in mind and no pipelining whatsoever, for the sake of simplicity. Therefore, if you want to use synchronous RAM, you have to generate a clock signal that will register the memory read address before the end of the uCore cycle proper. SYN_CORE.VHD and ROM_SYN_CORE.VHD take care of that by using an external clock twice the uCore clock frequency, operating the synchronous RAM blocks at twice the uCore frequency.

3 Using three-state drivers for busses/multiplexers

Three state busses can be used in FPGAs that have 3-state buffers available as a resource. A three state implementation usually consumes considerably fewer resources. As a drawback, you have to generate the TRI_EN signal (in the CORE entity), which in general can only be generated from the delayed clock. TRI_EN is used to disable all 3-state buffers at the beginning of a new cycle in order to make sure that no short circuit situations will occur due to different delays of the enable signals of the 3-state buffers. Neither the synthesiser, nor the place&route tools will be able to realise this delay precisely. In the present implementation, it is generated by routing the clock to an un-used pad DELAY, whose input signal is used as the delayed clock signal. In real implementations this may need some tweaking before final sign-off. During debugging, you may set TRI_EN permanently true.

4 Simple / extended Version

Depending on constants "tasks" and "locals" in CONSTANTS.VHD a "simple" or an "extended" version of MicroCore will be realised. This has to be reflected in the test program generated for the simulator using the cross-compiler. Different load files have been prepared: LOAD_CORETEST.F for the simple, LOAD_CORETEST_EXT.F for the extended version.

5 Observed realisability issues. XC4000 versus Virtex families.

First experiments indicate that the XC4000 as well as Spartan-families of Xilinx FPGAs are not well suited for MicroCore. These families have routing problems and they may benefit from a pipelined version of MircoCore, which would also better support synchronous RAM blocks. On the other hand, even the smallest member of the Virtex family may be used to create an efficient 24-bit version with ease.

6 Debug philosophy

The debug version of MicroCore uses a RAM as program memory. In addition, a debug interface needs to be added (e.g. the Centronics port in debug_centronics.vhd) for control of the processor and upload of the program. The debugger works in such a way that MicroCore is halted using the CLK_EN signal while the content of the program memory RAM is altered. Afterwards, CLK_EN is asserted again and MicroCore continues execution where it was halted without loss of state. That way, "breakpoint tokens" may be shifted through the executable code under control of a debugger on the host in order to realise a single-stepping emulator. Alternatively, MicroCore may be reset after program load using control signals of the debug interface.

7 The Forth cross-compiler

Install Win32For
(e.g. from ftp://ftp.taygeta.com/pub/Forth/Compilers/native/windows/Win32For/W32for42.exe).
This is the base 4th system, which I used to realise the cross-compiler. Its advantage is that it is a 32-bit system, which makes scalability of microcore up to 32-bits easy. And it is public domain.

Its disadvantage is its enormous complexity. But for the cross-compiler I only used Standard 4th words using Win32For as the development environment, which is pretty much obvious and self explanatory. But I took full advantage of its inherent 32-bitness.

For simulation code generation, several load files have been prepared. These load files produce code for a functional MicroCore test. These tests are comprehensive as far as the instruction repertoire is concerned, but they do not test the proper functionality of neither interrupts nor the trap mechanism. Modify the load files to adapt to different application needs.

When in Win32For, type

> *include <full_name_of_load_file>*

This loads all necessary code and produces a VHDL output file that can be used in simulation. After compilation, you will be in the Win32For system. Here you can call the dis-assembler e.g. to show compiled code starting at hex address $102:

> *hex 0102 disasm*

which disassembles the code starting at program memory location $102, one line at a time for every key stroke - you get out of it with either <cr> or <esc>. The variable EXPAND controls the dis-assembler behaviour. EXPAND OFF (default) will display macros with their macro name. EXPAND OFF will display every single opcode without reference to a generating macro.

But for the hardware development, optimisation is not needed. Please note that for testing purposes EVERY opcode can be synthesised by delving into the "Assembler" using "{" and "}":

```
.... something_and_other_code { LIT NONE MEM } more_4th_code ...
```

This includes the opcode for LIT NONE MEM in the instruction stream. These "Assembler" words are in a different context because otherwise, there would have been naming conflicts.

8 Cross-Compiler Source files

All cross compiler related files are in sub-directory "uForth"

Filename	Description
constants.f	Constants used globally
microcore.f	the MicroCore cross-compiler
disasm.f	the MicroCore dis-assembler
coretest.f	test program for the simple kernel
coretest_ext.f	test program for the extended kernel
debugger.f	test program for the debugger, including debug mirco kernel
load_coretest.f	load file for coretest.f
load_coretest_ext.f	load file for coretest_ext.f
load_debug.f	load file for debugger.f
prolog_case.vhd epilog_case.vhd	prolog and epilog files for generating vhdl code to model the cross compiled executable as a big case statement
prolog_rom.vhd epilog_rom.vhd	prolog and epilog files for generating vhdl code to model the cross compiled executable as a ROM array.