Simulator ModelSim (VHDL93 = 0, Explicit = 1)

Synthesizer Synplify

Architecture Architecture alternatives are realised by setting constants in constants.vhd. You have to take care of editing your own load/make files for both the simulator and synthesizer, which tend to be tool specific. The source files have been listed in alphabetic order.

1 VHDL Sources

Filename	**Description**
alu.vhd	Arithmetic-Logic Unit
bench.vhd debug_bench.vhd uCore100_bench.vhd	Generic test bench for the simulator Generic test bench for simulating the umbilical debug interface. Test bench for simulating the uCore100 prototyping board instantiation.
clocks.vhd	clock generation for the tri-state bus and synchronous RAMs
constants.vhd uCore100_constants.vhd	Global configuration, busses and records Configuration for the uCore100 prototyping board instantiation.
core.vhd debug_core.vhd uCore100.vhd	Generic top level entity. Here pin assignments will be specified for a concrete implementation. Generic top level entity with umbilical debug interface. Top level entity for the uCore100 prototyping board instantiation.
debug_centronics.vhd	Umbilical interface using a centronics port.
dstack.vhd	data stack entity
functions.vhd	Packages and sub-entities used throughout the project.
interrupt.vhd	Interrupt conditioning
peripherals.vhd	Memory mapped registers for the uCore100 prototyping board instantiation.
program.vhd	Program memory generated by the cross-compiler
rstack.vhd	Return stack entity
sequencer.vhd	Instruction address generator
uBus.vhd	Internal microcore tri-state bus
uCore.vhd	Technology independent top level entity.

2 Simulation File Order

```
vlib work
vcom ../functions.vhd
vcom ../constants.vhd
vcom ../dstack.vhd
vcom ../alu.vhd
vcom ../rstack.vhd
vcom ../program.vhd
vcom ../sequencer.vhd
vcom ../interrupt.vhd
vcom ../uBus.vhd
vcom ../uCore.vhd
vcom ../clocks.vhd
vcom ../core.vhd
vcom ../bench.vhd
```

3 Synthesis for the uCore100 Prototyping Board

```
#add_file options
add_file -vhdl -lib work "../functions.vhd"
add_file -vhdl -lib work "../uCore100_constants.vhd"
add_file -vhdl -lib work "../dstack.vhd"
add_file -vhdl -lib work "../alu.vhd"
add_file -vhdl -lib work "../rstack.vhd"
add_file -vhdl -lib work "../sequencer.vhd"
add_file -vhdl -lib work "../interrupt.vhd"
add_file -vhdl -lib work "../uBus.vhd"
add_file -vhdl -lib work "../uCore.vhd"
add_file -vhdl -lib work "../clocks.vhd"
add_file -vhdl -lib work "../debug_centronics.vhd"
add_file -vhdl -lib work "../Peripherals.vhd"
add_file -vhdl -lib work "../uCore100.vhd"
```

4 MicroCore Scaling

Setting constants in **constants.vhd** allows to instantiate different configurations, design alternatives, and computational capabilities. The values assigned to these VHDL constants in the following description are the ones used for the uCore100 prototyping board.

Important notice: Most of these settings have to be "ported" to the cross-compiler by setting Constants in the **load_<application>.f** file appropriately.

4.1 Semantic Switches

CONSTANT syn_stackram : STD_LOGIC := '1';

When set to '1', the stack_ram will be relised as synchronous blockRAM. Otherwise, it will be realised as asynchronous RAM, which may be internal or external of the the FPGA.

CONSTANT with_locals : STD_LOGIC := '1';

When set to '1', the Instantiates the LOCAL addressing mode relative to the return stack pointer (RSP+TOS).

CONSTANT with_tasks : STD_LOGIC := '1';

When set to '1', the TASK addressing mode relative to the TASK register (TASK+TOS) will be instantiated. For multi-tasking, tasks_addr_width (see below) has to be set appropriately as well.

CONSTANT with_nos : STD_LOGIC := '1';

When set to '1', the NOS (Next-Of-Stack) register will be instantiated. This is needed for the single cycle SWAP instruction and the complex math step instructions.

CONSTANT with_tor : STD_LOGIC := '1';

When set to '1', the TOR (Top-Of-Return_stack) register will be instantiated. This is needed for the decrement_and_branch instruction NEXT and the complex math step instructions.

CONSTANT with_ip : STD_LOGIC := '1';

When set to '1', the IP (Instruction Pointer) register will be instantiated. This is needed for the THREAD and TOKEN instructions for interpreting threaded code.

CONSTANT with_tokens : STD_LOGIC := '1';

When set to '1', the TOKEN instruction will be instantiated, which allows rapid token threaded code interpretation.

4.2 Vector Widths

CONSTANT data_width : NATURAL := 32;

This defines the data path width and therefore, the magnitude of the numbers that may be processed. Please note that the object code will not change as long as the magnitude of the largest number to be processed fits the data path width.

CONSTANT data_addr_width : NATURAL := 21;

This sets the address range of the data memory, which can at most be data_width-1 wide because the "upper" half of the address range is used for external memory mapped I/O.

CONSTANT dcache_addr_width : NATURAL := 0;

Number of address bits of the data memory space that is realised as block-RAM inside the FPGA.

CONSTANT prog_addr_width : NATURAL := 19;

Program memory address width sets the size of the program memory. It can be at most data_width wide because all program addresses have to fit on the return stack.

CONSTANT pcache_addr_width : NATURAL := 0;

Number of address bits of the program memory space that is realised as block-RAM inside the FPGA. When pcache_addr_width=0, no internal RAM is used; when pcache_addr_width=prog_addr_width, no external RAM is used at all.

CONSTANT prog_ram_width : NATURAL := 16;

Number of address bits that may be used to modify the program memory van Neumann style. If set to zero, the program memory operates as a pure ROM of a Harvard Architecture.

CONSTANT ds_addr_width : NATURAL := 6;
Number of address bits for the data stack memory.

CONSTANT rs_addr_width : NATURAL := 8;
Number of address bits for the return stack memory.

CONSTANT tasks_addr_width : NATURAL := 3;
Number of address bits for the task address. 2**tasks_addr_width copies of the data and the return stack will be allocated. The task address is added to the left of both the ds_address and the rs_address.

CONSTANT usr_vect_width : NATURAL := 3;
The implicit call destination addresses for two adjacent USR instructions will be 2**usr_vect_width apart from each other.

CONSTANT reg_addr_width : NATURAL := 3;
Number of address bits reserved for internal memory mapped registers that reside at the upper end of the address space.

CONSTANT interrupts : NATURAL := 2;
Number of interrupt inputs and their associated FLAGS and Interrupt-Enable bits.

CONSTANT token_width : NATURAL := 8;
Number of bits for a token address of a token threaded system.

5 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 debugger. Alternatively, MicroCore may be reset after program load using control signals of the debug interface.

6 The Forth cross-compiler

6.1 For Microsoft Windows

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.

Modify **`config.f`** to load **win32for-config.f** as configuration file.

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.

6.2 For Linux

Install gforth (e.g. from http://www.gnu.org/software/gforth/gforth.html)

Modify **`config.f`** to load **gforth-config.f** as configuration file.

Proceed as described for win32for above.

6.3 Cross-Compiler Source files

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

Filename	**Description**
config.f gforth-config.f win32for-config.f	Configuration file in order to adapt to different forth systems gforth specific adaptation file. win32for specific adaptation file.
constants.f	Constants used globally
microcore.f	the MicroCore cross-compiler
disasm.f	the MicroCore dis-assembler
forth.f	Basic Forth words not implemented in the cross-compiler
coretest.f	test program for the simple kernel
debugger.f	test program for the debugger, including debug mirco kernel
load_coretest.f	load file for coretest.f
load_debug.f	load file for debugger.f
runlight.f	A simple application using the LED bar on the uCore100 prototyping board.
prolog.vhd epilog.vhd	prolog and epilog files for generating vhdl code to model the cross compiled executable as a big case statement