μForth

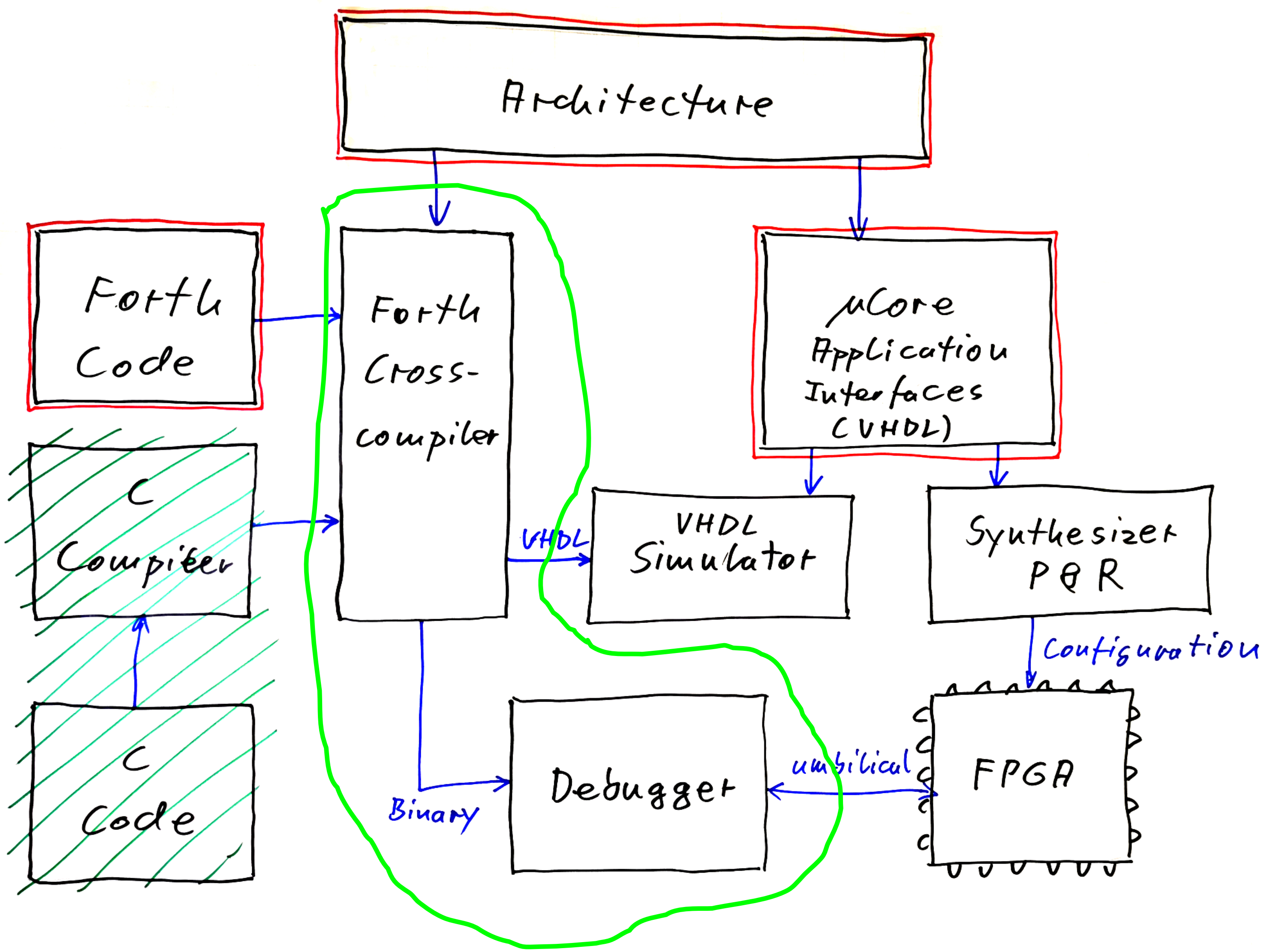
microCore's Assembler

and

Interactive Development Environment

Klaus Schleisiek
kschleisiek at freenet.de

Design Flow


Architecture
Forth Code
Forth Cross-compiler
μCore Application Interfaces (VHDL)
C Compiler
C Code
VHDL
VHDL Simulator
Synthesizer P&R
Configuration
Binary
Debugger
umbilical
FPGA

Characteristics

Command line oriented IDE for microCore consisting of

- µForth cross-compiler
- Compiling forward branches
- Libraries
- Umbilical debugger (dis-assembler, single step tracer)
- Co-operative multitasker and semaphores
- Math instructions
- Floating point

Please refer to uForth.pdf for a full description of the µForth and debugging wordsets.

µForth cross-compiler

- µCore's assembler is a Forth dialect - microForth
- The cross-compiler runs on a host PC.

  Only run time code will be produced on the target. The dictionary and code that is only needed during compilation will remain on the host.
- Peephole optimization, e.g. tail calls replaced by branches.
- When a load file is included (e.g. load_core.fs), µForth is always loaded from source before compiling the application code.
- µForth loads on top of gforth_0.6.2, which is available as a docker image ('docker pull microcore/gforth_062').

e.g. load_core.fs

```
Only Forth also definitions hex

include extensions.fs           \ Some System word (re)definitions
include ../vhdl/architecture_pkg.vhd
include microcross.fs           \ the cross-compiler

Target new initialized          \ go into target compilation mode and initialize

9 trap-addr code-origin
          0 data-origin

include constants.fs            \ MicroCore Register addresses and bits
include debugger.fs
library forth_lib.fs
include coretest.fs

init: init-leds ( -- )  0 Leds ! ;

: boot  ( -- )  0 #cache erase   CALL initialization   debug-service ;

#reset TRAP: rst    ( -- )            boot                 ;  \ compile branch to boot
#isr   TRAP: isr    ( -- )            interrupt IRET       ;
#psr   TRAP: psr    ( -- )            pause                ;  \ call the scheduler
#break TRAP: break  ( -- )            debugger             ;  \ Debugger
#does> TRAP: dodoes ( addr -- addr' ) ld cell+ swap BRANCH ;  \ the DOES> runtime primitive
#data! TRAP: data!  ( dp n -- dp+1 )  swap st cell+        ;  \ Data memory initialization

end
```

Compiling Forward Branches

This is tricky. The branch offset may require multiple LIT instructions preceding the branch instruction itself.

When an IF or WHILE is compiled, an offset that fits into a single LIT instruction is assumed. Not only the offset's address that has to be filled is pushed on the stack as usual, but also its source code location.

When the closing ELSE, THEN, or REPEAT is encountered, its offset can be computed.

- If it is less than 64, it fits into a single LIT and we are done.
- Otherwise, we now know how many LITs will be needed and the source code will be re-compiled with the proper number of LIT instructions in front of the branch.

Libraries

library forth_lib.fs will pre-compile forth_lib.fs as a library

- This produces dictionary entries for each word definition in the library compiling pointers to the word's source code.
- In this step, no code will be produced for the target.
- When a pre-compiled word is used later on during compilation or interactive interpretation via the umbilical, the word's source code will be loaded producing actual target code.
- Therefore, no dead code will be compiled.
- These libraries exist so far: forth_lib.fs, task_lib.fs, and float_lib.fs.
- Alternatively, libraries can be loaded using e.g. include forth_lib.fs, which will immediately compile all of the target code as usual.

Umbilical Debugger

Host and Target communicate via a two-wire RS232 umbilical link in order to load µCore's program memory and to control µCore interactively via a terminal program on the host. In the latter case one has the look and feel of interactively using a Forth system on the target itself.

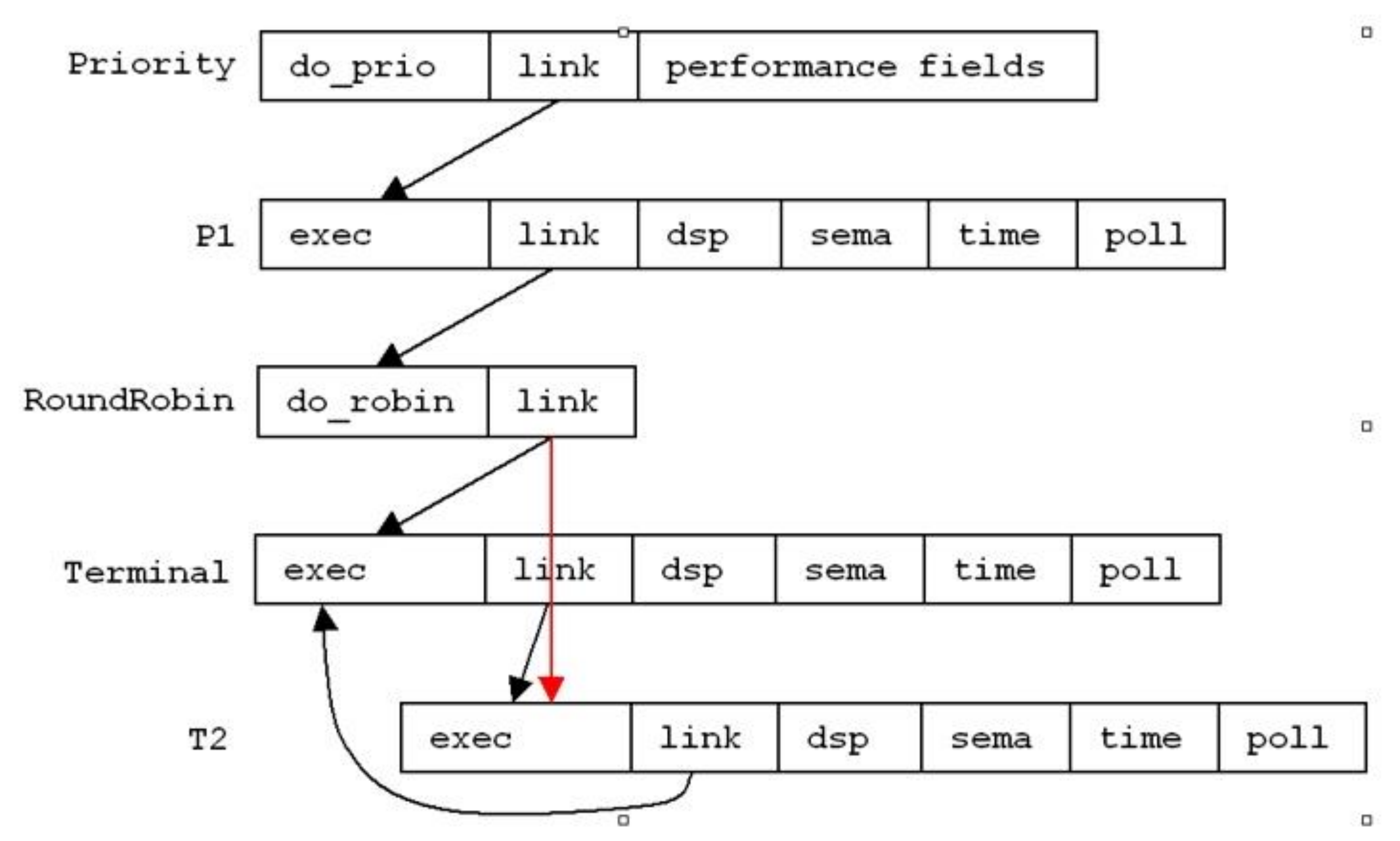
A dis-assembler allows to inspect the compiled code using show <wordname>.

A single step tracer allows to observe the stack while executing a colon definition using trace <wordname>. After every step, the stack can be manipulated or one of the following commands can be used:

- Nest to follow a call instruction. Unnest to fall back into the calling word.
- After to continue single stepping behind a backward branch after finishing the loop.
- Jump to skip the next instruction for debugging purposes.

Multitasker

The scheduler is a linked list of Task-Control-Blocks with a pointer to executable code that represents a task's state.



Tasks and Semaphors

µCore allows for 2**tasks_addr_width tasks using their own data- and return-stack areas.

- Task <name> creates a task.
  - pause, halt, wake, stop, activate, deactivate, schedule, spawn, cancel, poll, and poll_tmax are used for task control.
- Semaphore <name> creates a semaphore.
  - lock and unlock are used for mutual exclusion.
  - wait and signal are counting operators for synchronizing interrupts and tasks.

A full task switch takes 7 µsec on a 25 Mhz system.

Math Instructions

Several instructions allow for the following mathematical functions when a hardware multiplier is present:

Single/dual cycle: um*, m*, and * with overflow detection

Bit step instructions for: um/mod, m/mod, sqrt, and log2

Floating Point

Floating point numbers are data_width wide and therefore, they can be handled on the data stack just like integers.

Their exponent is exp_width wide and therefore, the mantissa is data_width - exp_width wide. In a 24 bit system, a 6 bit exponent and an 18 bit mantissa allows for meaningful floating point computations.

Floating point I/O usually is responsible for about 70% of the code. Therefore, I/O has been realized using scaling operators micro, milli, kilo, and mega to adjust integers for integer I/O.

The floating point package compiles to about 500 instructions including flog2 and fexp2 transcendental functions.

Several µCore instructions have been realized for floating point:
>float (man exp -- fp), float> (fp -- man exp), normalize (man exp -- man' exp'), *.

Links

microCore is available on git:

https://github.com/microCore-VHDL

and here is its documentation:

https://github.com/microCore-VHDL/microCore/tree/master/documents

uCore_overview.pdf
getting_started.pdf
uCore.pdf
uCore_instructions.pdf
uForth.pdf
uCore_Public_License.pdf