uCore progress

with remarks on arithmetic overflow

Klaus Schleisiek

SEND Off-Shore Electronics GmbH, Hamburg

ks @ send.de

State of uCore affairs

- **uCore 1.x is finished.**
- **Version 1.65 defines a very rich instruction set. Because of its co-design environment, it can be safely reduced to match application needs.**

Progress during the past year:

- **New application using Lattice LFXP2-17E as a "single chip controller"**
- **Cross-compiler interprets CONSTANTS.VHD and therefore, uCore is a co-design environment building upon one single source**
- **Step instructions for UM/MOD and FM/MOD executing in #bits+2 cycles**
- **Meticulous overhaul of all arithmetic overflow conditions**
- **Deterministic results on overflow**
- **Simplification of the bit-wise writable register mechanism**

Co-design environment

- **In a hardware / software co-design environment, both the hardware and the software can be simulated in a unified environment.**
- **Changes in the hardware should directly modify the software as well. Otherwise, hardware and software may deviate, working with inconsistent processors models.**
- **For uCore this means that the Forth cross-compiler must derive its "knowledge" about the architecture and the instruction set from uCore's VHDL specification.**
- **This is defined in CONSTANTS.VHD**

VHDL code interpretation

Therefore, CONSTANTS.VHD has to be loaded into Forth before loading the cross-compiler itself

```
include vhdl.fs
include ../uCore/constants.vhd
include microcore.fs
include disasm.fs
include constants.fs
```

VHDL code to be interpreted

Various constants that define compiler switches, bus widths, register addresses, and control values:

```
CONSTANT with_mult       : STD_LOGIC := '0';
CONSTANT data_width      : NATURAL := 24;
CONSTANT flag_reg        : INTEGER := -2;
CONSTANT mark_start      : byte := "00110011";
```

... and instructions:

```
--ALU POP \ and PUSH
CONSTANT op_ADD    : inst_group := "000";
CONSTANT op_ADC    : inst_group := "001";
CONSTANT op_SUB    : inst_group := "010";
CONSTANT op_SSUB   : inst_group := "011";
CONSTANT op_AND    : inst_group := "100";
CONSTANT op_OR     : inst_group := "101";
CONSTANT op_XOR    : inst_group := "110";
CONSTANT op_NIP    : inst_group := "111";
```

VHDL code interpretation

In addition, the VHDL source has to be beefed up by additional words, which control Forth interpretation. These words all start with `--` turning them into comments for the VHDL compiler.

```
--VHDL ---~ ----------------------------------------------------
-- SST100 - constants.vhd
-- ----------------------------------------------------------------

LIBRARY IEEE;
USE     IEEE.STD_LOGIC_1164.ALL;
USE     work.functions.ALL;

PACKAGE constants IS
---~--  \ when loading the Forth cross-compiler, code between "---~" up to "--~--" will be skipped.
CONSTANT version          : NATURAL := 1100;
CONSTANT with_mult        : STD_LOGIC := '1'; -- '1' when FPGA has hardware multiply resources
CONSTANT data_width       : NATURAL := 24;    -- data bus width
---~
... more "VHDL only" code
-----
--BRA POP
CONSTANT op_ALWAYS : inst_group := "000"; -- ELSE, REPEAT
CONSTANT op_QZERO  : inst_group := "001"; --      ?dup IF
CONSTANT op_SIGN   : inst_group := "010"; --     0< 0= IF
CONSTANT op_NSIGN  : inst_group := "011"; --        0< IF
CONSTANT op_ZERO   : inst_group := "100"; --           IF
CONSTANT op_NZERO  : inst_group := "101"; --        0= IF
CONSTANT op_NOVL   : inst_group := "110"; --     ovfl? IF
```

cross-compilation consequences

- Now e.g. `op_ZERO` has been compiled into the Forth dictionary as a constant holding the instruction's bit pattern and we can use it to define a code compiler for the target system

  ```
  op_ZERO Brn: 0=BRANCH ( f addr -- )
  ```

  which later on will be used appropriately by `IF`, `WHILE`, and `UNTIL`.

- This way any change in the VHDL code will automatically be ported to the cross-compiler.
- In addition: When an instruction has been removed from the VHDL code, because it is not needed in the application, the cross-compiler will get a hiccup when it has not been removed there as well.

unsigned Division

At first I started with unsigned division `um/mod`, because it is easy. Three instructions are needed:

`op_UDIVS`	sets up the parameters
`op_DIVS`	the basic divison step executed once per bit
`op_UDIVL`	corrects the result and cleans up the stacks

`op_DIVS` holds its parameters in `TOS`, `NOS`, `TOR`, and the status register and therefore, it is fully interruptible.

```
: um/mod ( ud u -- urem uquot )
   udivs data_width times divs udivl ;
```

signed Division

Signed division was a problem for a long time. Until it ocurred to me that the obscure high level definition of `fm/mod` based on `um/mod` can be translated into VHDL quite easily.

```
: fm/mod  ( d n -- rem quot )
   dup >r   abs >r   dup 0< IF  r@ +  THEN
   r>  um/mod   r@ 0<
   IF  negate over IF  swap r@ + swap 1-  THEN
   THEN  rdrop ;
```

Therefore, signed division just needs two more instructions and, unfortunately, two more bits in the status register to remember the signs of the arguments

`op_SDIVS`	does the "intro" code
`op_DIVS`	identical to the "unsigned" step instruction
`op_SDIVL`	does the "correction" code

Overflow

- Once signed division was defined "in hardware" it was possible to set the overflow bit of the status register without run time penalty.
- Division overflow is quite complex adding a considerable amount of logic.
- Remains the `*` operation. Very often this is implemented as

```
: * ( n1 n2 -- n3 )   um* drop ;
```

  which delivers a misleading result in case of an overflow.

- Therefore, two more primitives have been added when multiply hardware is available implementing `m*` as a single cycle instruction

```
: * ( n1 n2 -- n3 )   m* multl ;
```

- Debugging was tricky.
  Reducing the data width to 8 bits allowed to do a complete test of all possible cases in about 15 minutes. This uncovered multiplication bugs as well.

? what to do on overflow ?

- Now the overflow bit of the status register is set/reset in a mathematically correct way.

So what?

The result is bogus nevertheless.

- We can branch depending on the overflow using

`ovfl? IF`	which could be a single cycle branch instruction
`?ovfl`	which is a conditional call to a fixed address on overflow

  but that adds runtime overhead and even if the program knows there was an overflow, the programmer may not know what to do.
  After all, the "division by 0" blue-screen of Windows is not really a meaningful result.

Returning a well defined result

- On overflow, we can return the "smallest" or the "largest" number that can be represented, i.e. $8000 or $7FFF in a 16 bit system instead of misleading bit patterns. E.g.:

`+n 0 /`	returns $7FFF
`-n 0 /`	returns $8000

- This leaves some cases, which are not so obvious:

`0 0 /`	I decided that it should return zero.
`$8000 negate`	If it returns $7FFF, e.g. the high level code for `fm/mod` does not work any more. So we better leave it at $8000, although it is intuitively as wrong as it can be.

- Does a commercial controller with "controlled overflow" exist?

Not much has been published about arithmetic overflow!

Setting register bits

- **A more efficient mechanism to realize "bit-wise writable registers" has been found.**

  `5 Ctrl-reg !` **sets bits 0 and 2 of the memory mapped control register without affecting the other bits of the register.**

  `5 invert Ctrl-reg !` **resets bits 0 and 2 without affecting the other bits.**

- **The sign-bit of the number stored into the register determines whether the number will be ored (sign-bit=0) or anded (sign-bit=1) with the content of the register.**
- **Compared to the previous mechanism, the code is more readable, more efficient, and bit_0 of the register can be used as a flag as well.**