Smart Stacks in VHDL

Ulrich Hoffmann and Andrew Read

EuroForth 2020

Introduction

This work originates from our development of seedbed, a minimal VHDL implementation of the seedForth [1] target. A softcore stack processor needs hardware implementations of stacks. *Smart Stacks* are an approach based on abstracted stack operations rather than register level control . An innovation of smart stacks is that they offer built-in exception handling.

Stacks have been implemented countless times before in VHDL and whilst we are not aware of similar ideas in publication, we do not claim originiality.

A stack as memory

The simplest stack is memory with a stack pointer. An illustrative VDHL entity is shown below:

(During this paper we present code snippets in VDHL which are intended to be self-explanatory. For readers not familiar with VHDL, it suffices to note that VHDL design units are known as entities, and that each entity comprises an interface plus one or more architectures. VHDL designs are hierachical, so that entities may instantiate other entities within their design.)

```
entity stack_1 is

    generic(width : natural;

        depth : natural );

    port(    clk : in std_logic;

        rst : in std_logic;

        input : in std_logic_vector(width - 1 downto 0);

        stack_pointer_n : in integer range 0 to depth - 1;

        write_enable : in std_logic;

        output : out std_logic_vector(width - 1 downto 0);

        stack_pointer : out integer range 0 to depth - 1

    );

end entity;
```

A VDHL module which instantiates this stack will be responsible for providing the new value of the stack pointer `stack_pointer_n` and setting `write-enable` on each clock cycle. If `stack_1` is coded with good-practice VHDL, then the vendor synthesis tools will instantiate it efficiently as FPGA block RAM.

A stack with operations

The example above is simple and efficient, but perhaps not very scalable. We would prefer just to instruct the stack on each clock cycle, and let it take care of write enable and the stack pointer by itself. `stack_2` accomplishes just that by making the `stack_pointer` an output only and adding a `stack_op` input:

```
entity stack_2 is

    generic(width : natural;

        depth : natural );

    port(    clk : in std_logic;

        rst : in std_logic;

        input : in std_logic_vector(width - 1 downto 0);

        stack_op : in stack_op_type;

        output : out std_logic_vector(width - 1 downto 0);

        stack_pointer : out integer range 0 to depth - 1;

        err_under : out std_logic;

        err_over : out std_logic

    );

end entity;
```

We have introduced a user-defined VHDL type

```
type stack_op_type is

    (s_nop, s_push, s_drop, s_replace, s_reset);
```

`stack_2` knows what is should do with the stack pointer and write enable depending on the operation.

Beyond the expected `s_push` and `s_drop` , `s_nop` is necessary because synchronous hardware logic updates registers every clock cycle. If no update is desired this must be specified. The `s_replace` operation completes the orthogonal set: register write is enabled but the stack pointer remains unchanged. `s_reset` allows the stack to be reset (meaning the stack pointer will be returned to its initialization value) without triggering `rst <= '1'` across the whole design.

Now that `stack_2` is managing its own stack pointer, it must also handle stack overflow or underflow conditions and raise exception signals. These are included in the port interface.

Implementing the stack with operations

`stack_2` is implemented in a straightforward manner around a VHDL `case` statement, as the following except illustrates:

```
case stack_op is

    when s_push =>

        we <= '1';    sp_n <= sp_inc;

    -- other cases



    when s_nop =>

        we <= '0';    sp_n <= sp;

end case;
```

This VHDL coding format is very scalable. As we introduce further complexity into our stacks, we need just set the appropriate signals for each operation in the `case` statement. VHDL synthesis tools handle `case` statements well and

produce optimized and efficient logic from them.

A stack with smart operations

Consider the Forth word `dup`. We could implement `dup` by routing the output of `stack_2` to the input and instructing `s_push`. Only a little more logic is requried for `?dup`, and `drop` is trivial. This stack seems to be quite useful!

But a moment's thought suggests that `swap` is going to be more difficult. We'll need a temporary register in the instantiating entity and the operation will take two clock cycles... and we don't want to even think about `rot`. The problem is that the underlying stack entity, which is built from simple memory, only outputs and inputs the top-of-stack item.

That entity can be modified such that it becomes possible to update the top *three* stack locations in the same clock cycle. Writing behavioral VHDL code to accomplish this is not difficult, but it may not be straightforward (and is beyond the scope of this memo) to write the VDHL code in such a way that the synthesis tools translate the design into an efficient combination of registers and block RAM .

With that modification achieved our repetoire of stack operations expands greatly. `stack_3`, our new entity, also outputs both the top-of-stack and next-on-stack items `tos` and `nos`:

```
entity stack_3 is

    generic(width : natural;

        depth : natural );

    port(    clk : in std_logic;

        rst : in std_logic;

        input : in std_logic_vector(width - 1 downto 0);

        stack_op : in stack_op_type;

        tos : out std_logic_vector(width - 1 downto 0);

        nos : out std_logic_vector(width - 1 downto 0);

        stack_pointer : out integer range 0 to depth - 1;

        err_under : out std_logic;

        err_over : out std_logic

    );
end entity;
```

```
type stack_op_type is

    (s_nop, s_push, s_drop, s_replace, s_reset,

    s_nip, s_replaceAndNip, s_dup, s_ifDup,

    s_swap, s_rot, s_over,

    s_depth

    );
```

`stack 3` is responsible for its own stack maniputations and so we chose to call it a *smart stack*. Two further extensions:

1. `s_depth` writes the value of the stack pointer itself onto the stack
2. `s_replanceAndNip` supports arithmetic operations with the signature `( x1 x2 -- x3 )`

Exception handling

Forth's `catch` and `throw` (Milendorf [2]) necessitate some special stack handling. For example, when a exception is thrown the return stack should be appropriately restored so as to facilitate onward program flow after the exception. Implementations of exception handling in Forth typically rely on hooks provided by the Forth virtual machine to read and write stack pointers directly.

Such an approach is also possible in a softcore Forth processor, but there are reasons to hesitate:

1. If we allow software to update stack pointers we rupture the encapsulation that abstracts stacks as hardware entities which ought to manage themselves.
2. Thinking from a hardware perspective might identify a better-performing and more efficient way to accomplish exception handling.

A smart stack with an embedded exception stack

Let's bring two of our stacks together in a single module. We will instantiate `stack_2` in parallel with `stack_3` inside a new entity, `stack_4`, and expand our set of stack operations.

Here are the three new operations concerned with exception handling:

```
s_saveSP, s_restoreSP, s_dropSP
```

1. `s_saveSP` sets up a new exception frame. The stack pointer of `stack_3` is pushed onto `stack_2`.
2. `s_restoreSP` throws an exception. The stack pointer of `stack_3` is updated with the top-of-stack value from `stack_2`, which is simultaneously popped off the stack.
3. `s_dropSP` completes exception handling when a subroutine exits normally. The top-of-stack value of `stack_2` is dropped but the stack pointer of `stack_3` is not affected. In this way the exception frame is discarded.

The instantiation of `stack 2` has taken the role of an embedded exception stack. `stack_4` now encapsulates exception handling through appropriate stack operations.

Exception handling is therefore fast (the stack pointer can be updated in a single clock cycle) and atomic (exception handling is fundamental operation rather then being written in software which could itself be liable to exceptions).

Using smart stacks with exception handling

Our seedbed softcore processor utilizes a number of stacks, principally the parameter stack, the return stack and a subroutine stack. Each of these has an embedded exception stack. The processor implements global exception handling by passing a relevant exception instruction to all of the stacks, which in turn handle the exception locally.

Incidentally, mimicking the actual behavior of Forth's `catch` and `throw` requires a slightly expanded set of operations, for example:

```
s_saveSPAndPush, s_restoreSPAndPush, s_dropSPAndDrop
```

take care of requirements such as recyling the throw code to the top-of-stack after throwing an exception, or placing a zero on stack after dropping an exception. Implementing these additional operations in the VHDL `case` statement is straightforward since the instantiations of `stack_2` and `stack_3` are separate entities which can be controlled independently.

seedbed is both a vehicle to extend experimentation with seedForth into hardware, and a successor to the N.I.G.E. Machine [3]. The N.I.G.E. Machine incorporated hardware exception handling on a global level [4] but the approach described in this paper is certainly more elegant.

Conclusion

We have developed a *smart stack* approach to hardware stacks in VDHL which focuses on abstraction and scalability. Combining two smart stacks, encapsulated as a single entity, provides simple exception handling.

This work is a spin-off of our research and development in seedForth. We welcome correspondence.

Ulrich Hoffmann (FH Wedel University of Applied Sciences), uh@fh-wedel.de

Andrew Read, andrew81244@outlook.com

References

[1] http://www.complang.tuwien.ac.at/anton/euroforth/ef18/papers/

[2] http://www.euroforth.org/ef98.html/

[3] http://www.complang.tuwien.ac.at/anton/euroforth/ef12/papers/

[4] http://www.complang.tuwien.ac.at/anton/euroforth/ef14/papers/