Compiling Hume down to gates

J. Sérot[1] and G. Michaelson[2]

[1] LASMEA, Université Blaise Pascal/CNRS, France,
Jocelyn.Serot@lasmea.univ-bpclermont.fr
[2] Heriot-Watt University, Edinburgh, Scotland
G.Michaelson@hw.ac.uk

Abstract. We describe the implementation of a subset of the Hume programming language on a FPGA architecture at the gate level. Hume is a domain specific language for developping multi-process systems requiring strong static guarantees that resource bounds are met. The compiler produces RT-level, synthetizable VHDL code that can be processed by a standard tool chain to program FPGAs at the gate level. Preliminary results suggest that this compilation route offers substantial opportunities for exploiting fine-grain parallelism in Hume programs. The approach also offers an significantly higher abstraction level than that offered by traditional hardware description languages such as VHDL or Verilog.

1 Introduction

1.1 Overview

Field Programmable Gate Areas (FPGAs) have long been promoted as a solution to the direct realisation of software in hardware. While CPUs offer very fast execution in hardware of low level instructions, a specific CPU design may not be optimal for individual programs. In contrast, in principle, an FPGA may be directly configured to realise a specific program.

And FPGAs have long been promoted as a way out of the restrictions of specific CPU designs on system scalability. While fabrication technology is rapidly increasing the number of processing elements in multi-core CPUS, nonetheless such cores are necessarily in some fixed configuration which may not be optimal for an arbitrary problem. In contrast, in principle, an FPGA of sufficient size may implement an arbitrary number of processing elements with arbitrary interconnections.

Nonetheless, there are immense practical problems in realising the full potential of FPGAs. In particular, FPGAs are very low level devices requiring expert understanding of hardware concerns to gain best performance. Thus, there has been considerable research into developing both languages for describing FPGA configurations at considerably higher levels of abstraction, and tool chains for seamlessly realising such abstracted configurations in hardware.

1.2 High-level imperative FPGA programming

High level hardware description languages (HDLs) are now very widely used for programming FPGAs. Such HDLs typically expose very low level hardware concerns within higher level type and control constructs, and have tool chains to generate gate-level descriptions from register-transfer level descriptions (RTL), for realisation ultimately in an Application Specific Integrated Circuit (ASIC) or configured FPGA.

The widely used Verilog[10] was strongly influenced by C. Similarly, the widely used and somewhat higher level VHDLwas strongly influenced by Ada. Both languages offer tool chains to RTL but have very restricted data types and control constructs.

At a higher level again, SystemCis based on library extensions to C++ and Handel-C[4] augments a C subset with hardware-oriented constructs. Both SystemC and Handel-C may be compiled to lower level HDLs for hardware realisation.

Microsoft's Kiwi project[9] is exploring the use of high level languages for hardware description within the .Net framework. The main focus is on C# with translation to VHDL.

HDLs are substantial improvements on traditional low level diagrammatic hardware design techniques. However, like system programming languages, they offer uneasy compromises between the higher and lower levels. First of all, HDLs usually build from bit or wire level components requiring expert understanding for effective use. Furthermore, they typically offer a subset of a full high level language, for example restricting types and control constructs, while nonetheless retaining high level constructs which may be simulated but not realised in hardware.

Finally, the marriage of a conventional imperative language with hardware oriented extensions is not always amicable. As the anonymous "bob" comments on the Kiwi project[9]:

> bob 14 Nov 2010 9:49 PM: *the vhdl code was cleaner and easier to understand than the C# code...*

1.3 Functional FPGA programming

There are also a number of functional approaches to FPGA programming, drawing on the classic FP strength of higher order abstraction to compose components for hardware realisation.

Lava[3] augments Haskell with modules for hardware description. The Lava tool chain generates VHDL. Sheeran[8] provides a useful reflection on Lava's origins. Several groups are actively developing Lava, for example Kansas Lava[5].

The Kiwi project has recently been complemented with the use of F#[9], a Standard ML derivation. Here, common middleware for all .Net compliant languages eases the route to VHDL.

Gannet[11] is a functional approach for configuring Systems on a Chip components. Gannet is in the Scheme tradition of dynamically typed, syntax-light languages and is realised in a SystemC tool chain.

All these functional approaches share the same problems of high level imperative languages for FPGA programming, that is the requirement for deep understanding of hardware architectures and restrictions on which high level constructs can be realised in hardware. Here, the inability to realise arbitrary recursion is particularly problematic given the statelessness of functional languages. Furthermore, functional languages bring the long standing difficulty of lack of familiarity for mainstream programmers and hardware designers.

1.4 Hume and FPGAs

Hume[6] is a contemporary language for developing multi-process systems requiring strong static guarantees that resource bounds are met. With roots in polymorphic functional languages, Hume is distinguished by an explicit separation of *coordination and expression layer*. The coordination layer, for configuring independent communicating processes, is based on concurrent finite state boxes connected by single-buffered *wires*. The expression layer defines control within boxes and is based on pattern matching on input values to enable general recursive actions to generate output values.

A crucial feature of Hume is that the expression layer is state free, with all local variable instantiations lost between execution cycles. However, in the coordination layer state is retained on wires. In particular, feedback wires from a box's outputs to its inputs enable individual boxes to retain state between execution cycles, and are the basis of box iteration.

Hume's tool chain is strongly based on the Hume Abstract Machine (HAM) which provides a unitary locus for consistent implementation and resource analysis. Thus, a standard compiler generates HAM code from Hume which may be:

- interpreted directly on the HAM;
- further compiled to native code, for example via C;
- analysed to identify resource bounds, for example via an amortised type system implemented within Isabelle.

A number of routes have been explored for implementing Hume on FPGAs. These include executing:

- HAM code on the HAM on single Power PC core[1];
- HAM code compiled to native code on single Power PC and microBlaze cores[1];
- Hume compiled directly via C to native code on multiple microBlaze cores[12].

All routes offer consistent, scalable speedup, but are nonetheless markedly slow compared with the equivalent routes on proprietary CPUs.

The Reduceron[7] is a soft core for an FPGA, implementing an abstract graph reduction machine for a minimal non-strict, higher order functional programming

language. It would be interesting to evaluate the Reduceron against the Hume abstract machine on FPGA: it is likely that the performance limitations of lazy evaluation are substantially outweighed by direct implementation of the abstract machine rather than via another soft core.

Nonetheless, we think that Hume's explicit separation of coordination and control addresses the problems encountered in other high level approaches to FPGA programming as discussed above, and offers a basis for direct realisation via a HDL.

Hume was explicitly intended for use as a multi-level language sharing a common coordination form. Each level reflects different restrictions on expressivity, in particular in the allowed use of types and functional forms, from Hardware Hume (HW-Hume), restricted to pattern matching on bit patterns, to full Hume which is Turing complete. Each level has different formal properties, so HW-Hume has decidable time and space behaviour and full Hume shares all the undecidability restrictions of Turing completeness.

Thus, given a base FPGA realisation of box coordination alone, then the expressivity at the control level might also be varied to reflect the sophistication of hardware compilation. We anticipate that Finite State Hume (FSM-Hume), which augments HW-Hume with fixed size types and arithmetic/logic operations, will be an excellent starting point for direct HDL implementation. Subsequently, Template Hume, which provides a fixed repertoire of higher order functions, offers a framework for exploring functional abstraction in composing hardware components, drawing on the experiences of the pure functional approaches discussed above.

Furthermore, we think that Hume coordination offers an appropriate degree of abstraction from hardware realisation, enabling efficient implementation without requiring deep knowledge of underlying hardware.

Finally, Hume encourages a box-based approach to software design. This, along with the notion of state machines, is very familiar to hardware designers which may ease acceptability beyond our perfectly formed but nonetheless very small community.

In subsequent sections we: introduce Hume and a running example in slightly more detail; give a short presentation of the VHDL programming language; describe our compilation route from Hume to VHDL; explore compilation of the example; and evaluate its performance on an FPGA.

2 mHume

mHume[12] is an evolving experimental version of Hume. Based around the full coordination layer, it provides a platform for exploring the direct compilation of different expression layer instantiations without going through the HAM. The mHume syntax is summarised in Figure 1. [12] describes direct compilation to C. The mHume syntax is summarised in Figure 1.

program	→ [*component*`;`]$^+$	*patt*	→ *int* | *var* | ∗
component	→ *box* | *wire* |	*exps*	→ *exp* [`,` *exps*]*
	stream | *typedef*	*exp*	→ *int* | *var* | `(` *exp* `)` | *exp op exp* | ∗
box	→ `box` *id*	*op*	→ `+` | `-` | `*` | `/`
	`in` (*links*)	*wire*	→ `wire` *id* (*inwires*) (*outwires*)
	`out` (*links*)	*inwires*	→ *inwire*[`,` *inwire*]*
	`match` *matches*	*inwire*	→ *id*[.*var*[`initially` *int*]]
links	→ *link* [`,` *links*]*	*outwires*	→ *outwire*[`,` *outwire*]*
link	→ *var*`::`*type*	*outwire*	→ *id*[.*var*]
matches	→ *match* [`|` *matches*]*	*stream*	→ `stream` *id* { `from` | `to` } `"` *path* `"`
match	→ *pattern* `->` *exps*	*typedef*	→ `type` *var* `=` *type*
pattern	→ *patt* [`,` *pattern*]*	*type*	→ *var* | `int` *int*

Fig. 1. mHume syntax.

The example in Figures 2 and 3, from [12], finds the squares of successive integers by repeated addition. Here a minimal expression layer for integers and integer operations is used.

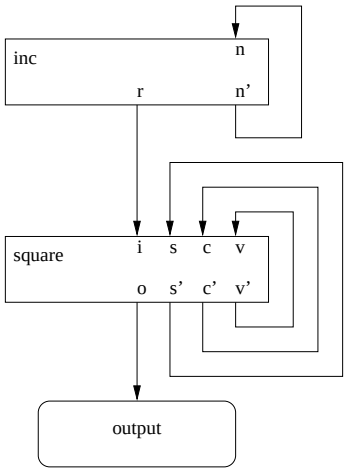

Fig. 2. Square program.

```
 1 type integer = int 64;

 2 box inc
 3 in (n::integer)
 4 out (r::integer,n'::integer)
 5 match (n) -> (n,n+1);

 6 box square
 7 in (i::integer,s::integer,
 8     c::integer,v::integer)
 9 out (o'::integer,s'::integer,
10      c'::integer,v'::integer)
11 match
12  (*,s,0,v) -> (s,*,*,*) |
13  (*,s,c,v) -> (*,s+v,c-1,v) |
14  (i,*,*,*) -> (*,0,i,i);

15 stream output to "std_out";

16 wire inc (inc.n' initially 0)
17          (square.i,inc.n);

18 wire square
19 (inc.r,square.s',square.c',square.v')
20 (output,square.s,square.c,square.v);
```

Fig. 3. Square program Code

Line 1 introduces `integer` as an alias for `int 64`, that is a 64-bit integer. Lines 2 to 5 define a box `inc` (2) with integer input wire `n` (3) and integer output wires `r` and `n'` (4). In line 5, an input is matched with variable `n` to output the value of `n` on wire `r` and `n+1` on wire `n'`. As we shall see, `n` is wired to `n'`. Essentially, `r` is the current and `n` is the next value for squaring Lines 6 to 14 define a box `square` (6) with integer inputs `i`, `s`, `c` and `v` (7 and 8), and integer outputs `o`, `s'`, `c'` and `v'` (9 and 10). In line 12, regardless of the input on `i` (`*`), if `c` is `0` then the (final) value from `s` is output on `o`. In line 13, regardless of the value on `i`, `v` is added to `s` and `c` is decremented. In line 14, with a new initial value for `i`, `s` is initialised to `0`, and `c` and `v` are initialised to `i`. As we shall see, `s` is wired to `s'`, `c` to `c'` and `v` to `v'`. Essentially, `i` is the value to be squared, `s` is the partial square, `c` counts how often `i` has been added to `s`, and `v` retains the initial value from `i` for repeated addition to `s`. Line 15 associates stream `output` with standard output. Lines 16 and 17 wire `inc`'s `n` to it's `n`, and `r` to `square`'s `i`. Finally, lines 18 to 20 wire `square`'s `i` to `inc`'s `r`, `s` to `s'`, `c` to `c'`, `v` to `v'` and `o` to `output`.

3 VHDL in a nutshell

In this section we introduce the basic concepts of VHDL, focusing on the features which are essential for explaining the principles upon which our compiler is built.

In VHDL, designs are described using a number of *modules*. Each module consists of a *entity* and at least one *architecture*. An entity describes the *interface* of the module : names and types of the input and output ports in particular. An architecture describes the *implementation* of the module : how it works. There are basically three ways to write an architecture : *structurally* (by instantiating other modules and connect them using *signals*), *concurrently* (by using a set of concurrent assignments that are re-executed every time signals they depend on change) and *sequentially* (by encapsulating sequential code within a *process*). Consider for example, the design depicted in Fig. 4, composed of two simple `and` gates. The corresponding entity and structural descriptions are given in Fig 5.

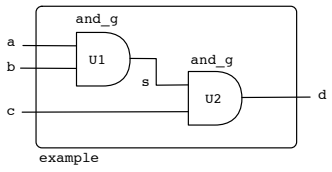

Fig. 4. VHDL example

```
ENTITY example IS
  PORT (a,b,c: IN BIT; d: OUT BIT);
END example

ARCHITECTURE structural OF example IS
  SIGNAL s : BIT;
BEGIN
  U1 : entity and_g(beh) PORT MAP (a,b,s);
  U2 : entity and_g(beh) PORT MAP (s,c,d);
END ARCHITECTURE
```

Fig. 5. VHDL structural description of the example

Two modules, named `U1` and `U2` are instantiated and explicitly connected using signal `s`. In VHDL, signals correspond to physical wires. Now, the `and_g` modules can be described by the following entity and *behavioral* description :

```
ENTITY and_g IS
  PORT (i1, i2 : IN BIT;
        o : OUT BIT);
END example
```

```
ARCHITECTURE beh OF and_g IS
BEGIN
  PROCESS (i1,i2)
  BEGIN
     o <= i1 and i2;
  END PROCESS
END ARCHITECTURE
```

The behavior of the module is here specified as a (sequential) *process*. The *sensitivity list* of this process contains the two inputs, `i1` and `i2`. This means that the process will be executed each time the signal connected to one of those inputs changes. Upon execution, the value `(i1 and i2)` will be computed and the signal connected to the output `o` will be updated with this value.

VHDL processes can also make use of *variables*. In contrast to signals, which are updated concurrently and globally at the end of the execution cycle, a variable is updated as soon as the sequential statement affecting it is executed. They therefore generally do not correspond to physical wires but to registers. Here's a possible description in VHDL of a 4-bit synchronous counter :

```
ENTITY counter IS
  port (
    val: OUT UNSIGNED(3 downto 0);
    clk: STD_LOGIC)
END counter;
```

```
ARCHITECTURE beh OF counter IS
  SIGNAL v: UNSIGNED(3 downto 0);
BEGIN
  PROCESS (clk)
    IF rising_edge (clk) THEN
      v := v + 1;
    END IF;
  END PROCESS;
  val <= v;
end beh;
```

Whenever a rising edge occurs on input `clk`, the internal variable `v` in incremented. A *concurrent* statement `val <= v` takes care of updating the counter output accordingly.

Synthesis. Synthesis is the process where a VHDL program is compiled and mapped into an implementation technology such as an FPGA or an ASIC. Not all constructs in VHDL are suitable for synthesis. While different synthesis tools have different capabilities, there exists a common synthetizable subset of VHDL that defines what language constructs and idioms map into common hardware for many synthesis tools. In the current state of the art, programs written at the *register transfer level* are synthetizable. Register transfer level (RTL) is a level of abstraction in which the circuit's behavior is defined in terms of data transfers between synchronous registers, all synchronized by the same clock, and the logical operations performed on those data.

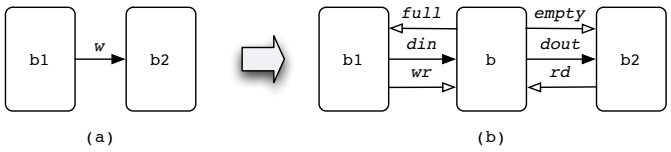

Fig. 6. Network generation. (a) Initial box structure (b) After buffer insertion

4 Compiling Hume to VHDL

Compiling a Hume program to a RT-level VHDL description involves three phases : network generation box translation and VHDL transcription.

4.1 Network generation

In this phase, we derive a *structural* description of the program as a network of components, where a component represent either a box or a wire of the original program. The process is sketched on Fig. 6. The key issue here is that Hume *wires* are not mapped to physical wires (VHDL signals) but to a dedicated component that we call a *buffer*. A buffer has one input and one output corresponding to the initial wire and four extra control signals : `full`, `empty`, `rd` and `wr`. The `full` (resp. `empty`) signal tells whether the buffer is ready for reading (resp. writing); it will be used by the box connected to its output (resp. input). The `rd` (resp. `wr`) signal, when asserted to `1`, actually pops (resp. pushes) the value from (resp. to) the buffer, passing it from the full (resp. empty) to the empty (resp. full) state.

4.2 Box translation

In this phase, each box of the original Hume program is translated into a finite state machine (FSM). This translation process closely follows the dynamic semantics of the language, in which a box can be in two different states : *Ready* (awaiting input) or *BlockedOut* (output pending).

Since we are targeting a RT-level description, all transitions will be triggered by a global `clock` signal. This means that all boxes will actually change state simultaneously. This dramatically simplifies[3] the scheduling algorithm, which can be rewritten as follows :

```
At each clock cycle
  For each box b , in parallel, do
    if b.state = Ready then
```

[3] Often, and as pointed out by G. Berry in [2] for instance, complex software solutions become trivial when described in hardware, because parallelism comes for free at this level.

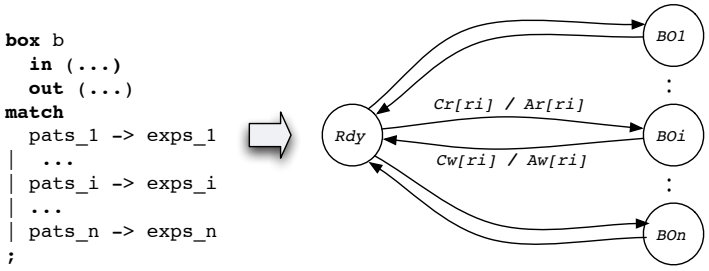

Fig. 7. Translation of a box into a FSM

```
    if a fireable rule r can be found in b.rules then
       read inputs for rule r;
       b.state <- BlockedOut
    end if
  else if b.state = BlockedOut then
    if outputs for the selected rule r are writable then
       write outputs for rule r;
       b.state <- Ready
    end if
  end if
end for
```

Each box can be therefore be described as a finite state machine (FSM) having $nrules + 1$ states : one state corresponding to the *Ready* state in the previous algorithm and one state per rule, corresponding to the *BlockedOut* state for the corresponding rule. This transformation is illustrated on Fig. 7. Each transition in the resulting FSM is labeled with a set of *conditions* and a set of *actions* (denoted *Conditions/Actions* on the diagram).

At each rule r_i we associate two sets of conditions and two sets of actions :

- the set $C_r(r_i)$ denotes the firing conditions for rule r_i, i.e. the conditions on the inputs that must be verified for the corresponding rule to be selected;
- the set $A_r(r_i)$ denotes the firing actions for rule r_i, i.e. the read operations that must be performed on the inputs when the corresponding rule is selected;
- the set $C_w(r_i)$ denotes the writing conditions for rule r_i, i.e. the conditions on the outputs that must be verified when the corresponding rule has been selected;
- the set $A_w(r_i)$ denotes the writing actions for rule r_i, i.e. the write operations that must be performed on the outputs when the corresponding rule has been selected.

There are

- two possible firing conditions : $Avail(j)$, meaning that the j^{th} input is ready for reading, and $Match(j, pat)$, meaning that the j^{th} input matches pattern pat;
- one firing action, $Bind(j, pat)$, meaning "read j^{th} input and match the corresponding pattern against pattern pat";
- one writing condition, $Avail(j)$ meaning that the j^{th} output is ready for writing;
- one writing action, $Write(j, exp)$, meaning "evaluate[4] expression exp and write the corresponding value on j^{th}.

Table 1 summarizes the rules for computing the sets C_r, A_r (resp. C_w and A_w) from the patterns (resp. expressions) composing a box rule. The FSM obtained for the `square` box introduced in Sec. 2 is given in Fig. 8.

$C_r[\![pat_1, ..., pat_n]\!] = \bigcup_{i=1}^{n} C'_r[\![i, pat_i]\!]$	$C'_r[\![i, *]\!] = \emptyset$
$C'_r[\![i, var]\!] = \{Avail(i)\}$	$C'_r[\![i, pat]\!] = \{Avail(i), Match(i, pat)\}$
$A_r[\![pat_1, ..., pat_n]\!] = \bigcup_{i=1}^{n} A'_r[\![i, pat_i]\!]$	$A'_r[\![i, *]\!] = \emptyset$
$A'_r[\![i, const]\!] = \emptyset$	$A'_r[\![i, pat]\!] = \{Bind(i, pat)\}$
$C_w[\![exp_1, ..., exp_n]\!] = \bigcup_{i=1}^{n} C'_w[\![i, exp_i]\!]$	$C'_w[\![i, *]\!] = \emptyset$
$C'_w[\![i, exp]\!] = \{Avail(i)\}$	$A_w[\![exp_1, ..., exp_n]\!] = \bigcup_{i=1}^{n} A'_w[\![i, exp_i]\!]$
$A'_w[\![i, *]\!] = \emptyset$	$A'_w[\![i, exp]\!] = \{Write(i, exp)\}$

Table 1. Rules for computing the sets C_r, A_r, C_w and A_w

4.3 Transcription to VHDL

The transcription in VHDL of the network derived in Sec. 4.1 boils down to instantiating the components forming this network and declaring the interconnection wires. The complete Hume program is turned into a VHDL component. The inputs and and outputs of this component correspond to the I/O streams declared in this program. This makes it possible to automatically generate a *test-bench* for the resulting VHDL design, in which the original input (resp. output) data streams are provided (resp. displayed) by specific VHDL processes reading samples from (resp. writing results to) to files for example.

Converting the FSM representation of boxes into VHDL is a little bit more involved. The *Avail* condition on an input (resp. output) is reflected directly into the value of the `full` (resp. `empty`) signal connected to this input (resp. output). But, because reading / writing is actually triggered by asserting the corresponding `rd` (resp `wr`) signals, an extra state must be added for each rule. This transformation is illustrated in Fig. 9 on a simple, mono-rule, example.

[4] This evaluation takes place in an environment augmented with the bindings resulting from the corresponding firing action; for the sake of readability environments have been left implicit here. A fully formalized account will be given in the final paper.

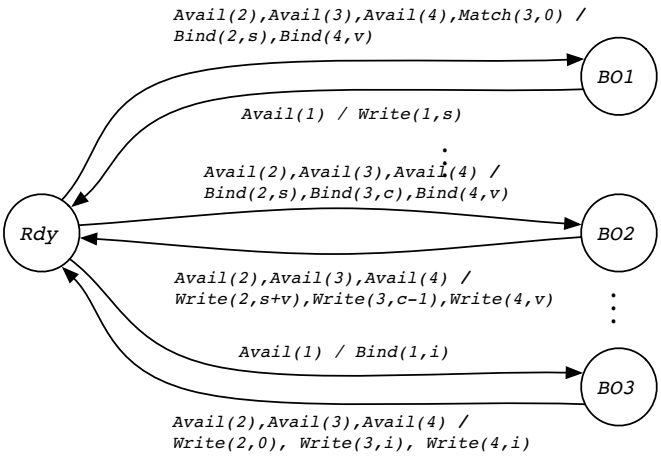

Fig. 8. FSM for the `square` box

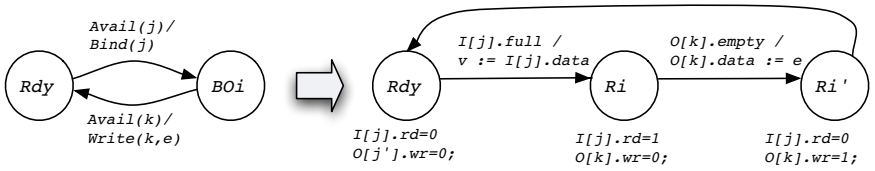

Fig. 9. Transformation of the FSM to generate the `rd` and `wr` signals (I[j] and O[k] respectively refer to the j^{th} input and k^{th} output of the box)

Since the syntax of the box-level expressions is very simple in mHume, the conversion of these expressions can be handled using a very simple syntax-directed function.

Listing. 1.1 gives the VHDL code generated for the `inc` box of the above example. Lines 1-15 give the interface of the component. Hume integers are translated to VHDL `std_logic_vector`s. As explained on Fig. 6, the `n`, `n_empty` and `n_rd` correspond to the `n` original input. Similarly, the `r`, `r_full` and `r_wr` (resp. `nn`, `nn_full` and `nn_wr`) correspond to the `r` (resp. `n'`) original output. The two other input signals are the global clock and a reset initial for hardware initialization. The behavior of the box is explicited in its architecture, lines 17-53. This architecture describes a synchronous FSM. The state variable is declared in line 19, its type being declared in line 18. Here the box has only one rule, so there are three states. The behavior itself is explicited as a *process* sensitive to the `clock` and `reset` signals (line 21). This process uses a internal variable `r1_n`. This variable memorizes the value obtained when the pattern of rule *r1* is bound

(line 33)[5]. The core of the process – which, according to VHDL execution model, is executed whenever the signal `clock` or `reset` changes value – is between line 23 and 52. Lines 24-28 handles asynchronous reset : the process state is reset to *Ready* and read/write signals are set to 0. Lines 30-50 describe what happens when a rising edge occurs on the `clock` input signal. This part is written in a very classical style, as a big `case` construct inspecting the value of the process state and, for each possible state, deciding on the actions to perform and the next state. For example, lines 32-35 tell that if process (box) is in the *Ready* state and a value is available on input `n` (line 32), then this value is copied (line 33), the read signal is asserted (line 34) and the next state will be `R1a` (line 35). In state `R1a` (lines 38-44), the read signal is reset to 0 and the availability of the output link is tested (line 39). If yes, the outputs are written (line 40-43) and the next state will be `R1b`.

5 Experimental results

Evaluation of the generated VHDL code has been carried out using the Altera Quartus II v9.0 tool chain.

For simulation, two specific, hand-written VHDL processes allow stream inputs and outputs to be read from and written to files.

Simulation results, for the `square` example introduced in Sec. 2 are displayed on Fig. 10. At this level, the clock period has been arbitrarily fixed to 10 ns. A close inspection of the chronograms shows that it takes four clock cycles to the `square` box to make an "iteration" (i.e. to increment the sum `s` and decrement the counter `c` by 1). Hence, computation of n^2 by the box will take $n \times 4$ clock cycles. The optimal version of such an operator, hand-crafted by a trained VHDL programmer, takes n cycles. The four to one ratio is indeed an very acceptable price to pay for abstraction here.

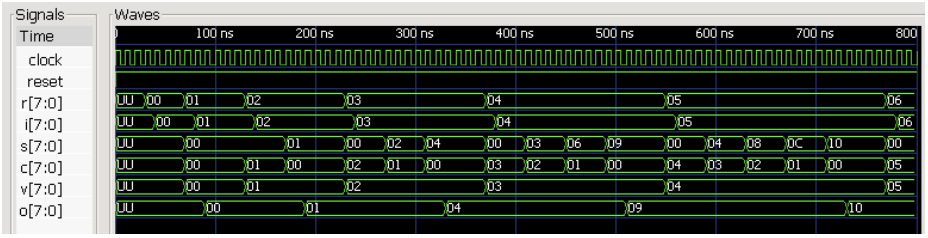

Fig. 10. Simulation results for the `squares` example

We performed the synthesis of this example on a *Stratix* EP1S80B956C6 FPGA. The default parameters for the synthetizer were used. Some views of

[5] Currently, a variable is introduced for each pattern appearing in each rule LHS. This can lead to redundancy and will be optimized in future versions of the compiler.

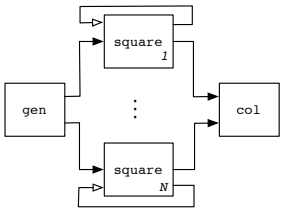

Fig. 11. Multi-square application

N	Occ (#LCs)	Occ (%)	Fmax (MHz)
1	693	<1	189
2	885	1	178
4	1263	2	178
8	2027	3	165
16	3550	4	164
32	6802	8	164
64	12702	16	139

Fig. 12. Synthesis results

the result are shown in Fig. 13 and Fig. 14. Fig. 13 is a top-level view of the synthetized network. Box labeled `box_wire` correspond to buffers. The two other boxes implement the `square` and `inc` original boxes. Fig. 14 shows the hardware architecture inferred by the synthetizer for the `inc` box. We can recognize a collection of registers[6] on the left, memorizing the internal outputs (`r` and `nn`) of the box and the adder performing the `n+1` operation (drawn as a small circle). The rectangular box at center left implements the FSM control.

For performance evaluation, we used a slightly modified application, depicted in Fig. 11. The first box, *gen*, generates an integer and passes it to the boxes $square_1$, ..., $square_N$. All these boxes are similar to the *square* box described above : they compute the square of the integer received by using iterative sum and count operations. The *col* box simply collects all the results.

Table 12 gives the occupation (number of logic cells used in the chip) and maximum clock frequency for different values of N, the number of *square* boxes running in parallel. The reported numbers show it should be possible to pack approximately 400 applications in a single FPGA of this category. This represent a vast improvement compared to the previous hardware implementation of mHUme on a FPGA [12] for which the number of parallel boxes was limited by the number of soft-core processors that could be instantiated on a chip (typically less than a dozen). This large amount of parallelism also largely compensate for the relatively small clock frequency, compared to a classical CPU. It takes 40000 cycles, i.e. approximately 270 μs at 150 MHz, on our FPGA to compute the square of 400 integers in the range 1...10000. Reaching the same throughput on a sequential CPU would require that this processor computes the square of an integer in less than 0.7 μs, that is, that it performs one sum-count iteration in less than $0.07ns$!

6 Conclusion

We have described a simple model and compilation route for implementing a subset of the Hume programming language on a FPGA directly at the gate level using state-of-the art synthesis technology. Preliminary results suggest that this

[6] For the sake of readability, the integer size has here been reduced to 8.

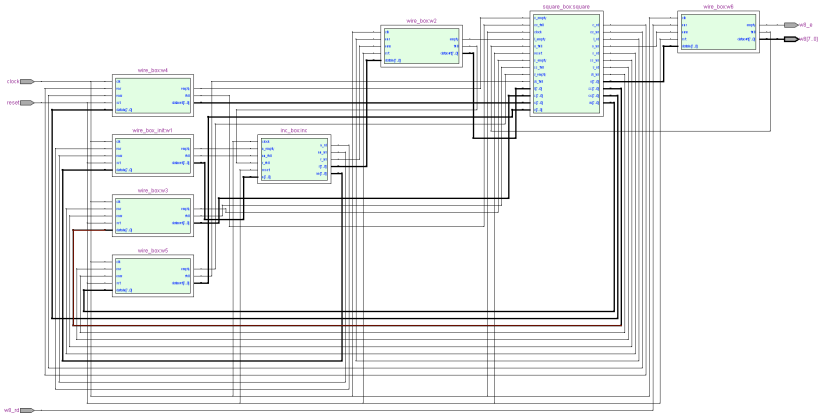

Fig. 13. Top-level synthetized network for the `squares` examples

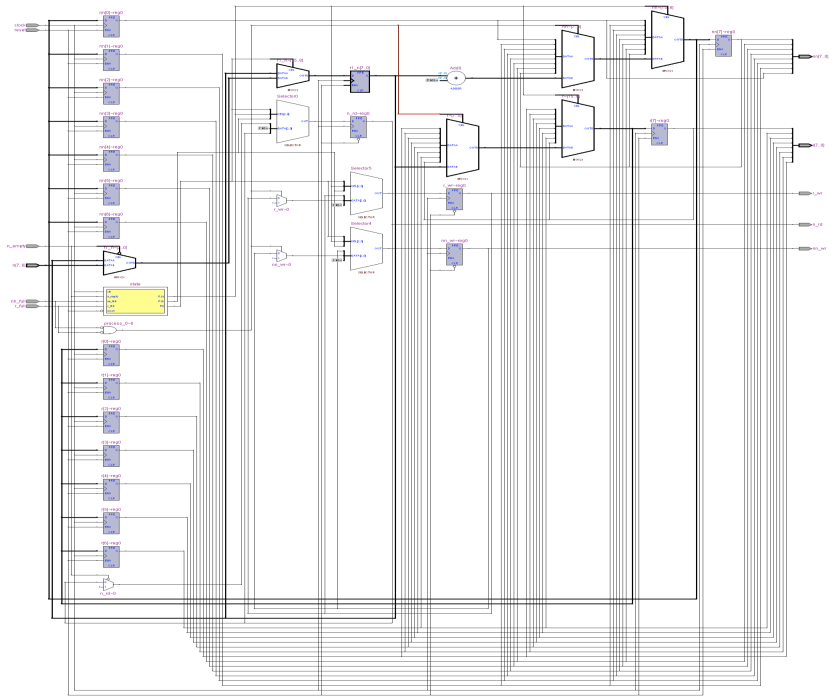

Fig. 14. Synthetized RTL architecture for the `inc` box

approachc route offers vast opportunities for exploiting fine-grain parallelism in Hume programs and a significantly higher abstraction level than that offered by traditionnal hardware description languages such as VHDL or Verilog.

The Hume subset currently supported is still limited, with data types and computation constructs restricted to integers and simple operations on these integers but it includes all the key features of Hume's coordination layer. So we are confident that significantly larger and more complex programs than that illustrated in this paper can be implemented on medium-sized FPGAs.

Among the many opportunities for further work, our two priorities are the extension of the expression language supported (integrating conditionnals, `let` expressions, *etc.*), support for global and external function declarations and optimisations of the generated VHDL code. It should be possible, in particular, to replace looping wires – connecting a box's output to one of its input – by internal process variables, and therefore minimize register allocation.

Acknowledgements. This work was partly supported by UK EPSRC project EP/F030592/1 'Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds'.

References

1. A. Al Zain, W. Vanderbauwhede, and G. Michaelson. Hume on fpga. In *Draft Proceedings of 10th International Symposium on Trends in Functional Programming (TFP10)*, University of Oklahoma, Oklahoma, USA, 2010.
2. G. Berry. *Penser, modéliser et maîtriser le calcul informatique*. Fayard, 2009.
3. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. *SIGPLAN Not.*, 34:174–184, September 1998.
4. Celoxica. *Handel-C Language Reference Manual*. Celoxica, 2005.
5. Andy Gill, T. Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and B. Werling. Introducing Kansas Lava. In *21st International Symposium on Implementation and Application of Functional Languages*. LNCS 6041, LNCS 6041, 11/2009 2009.
6. K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. GPCE 2003: Intl. Conf. on Generative Prog. and Component Eng., Erfurt, Germany*, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.
7. Matthew Naylor and Colin Runciman. The Reduceron Reconfigured. *SIGPLAN Not.*, 45:75–86, September 2010.
8. M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.
9. S. Singh. Kiwi Synthesis of C# and F# Combinational Circuit Models into FPGA Circuits. *Satnam Singh's MSDN Blog*, April 2010.
10. D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Springer, 1996.
11. W. Vanderbauwhede. Gannet: a Scheme for Task-level Reconfiguration of Service-based Systems-on-Chip. In *Proceedings of 8th ACM Workshop on Scheme and Functional Programming. Universite Laval, CA*. ACM, 2007.
12. A. Al Zain, Greg Michaelson, and Wim Vanderbauwhede. mHume for Parallel FPGA. In *Draft proceedings of 22nd International Symposium on Implementation and Application of Functional Languages, Amsterdam*, September 2010.

```
1  entity inc_box is
2     port ( n_empty: in std_logic;
3            n: in std_logic_vector(63 downto 0);
4            n_rd: out std_logic;
5            r_full: in std_logic;
6            r: out std_logic_vector(63 downto 0);
7            r_wr: out std_logic;
8            nn_full: in std_logic;
9            nn: out std_logic_vector(63 downto 0);
10           nn_wr: out std_logic;
11           clock: in std_logic;
12           reset: in std_logic );
13 end inc_box;
14
15 architecture FSM of inc_box is
16     type t_state is (R1a,R1b,Ready);
17     signal state: t_state;
18 begin
19   process(clock, reset)
20     variable r1_n : std_logic_vector(63 downto 0);
21   begin
22     if (reset='0') then
23       state <= Ready;
24       n_rd <= '0';
25       r_wr <= '0';
26       nn_wr <= '0';
27     elsif rising_edge(clock) then
28       case state is
29         when Ready =>
30           if n_empty='0' then
31             r1_n := n;
32             n_rd <= '1';
33             state <= R1a;
34           end if;
35         when R1a =>
36           n_rd <= '0';
37           if nn_full='0' and r_full='0' then
38             nn <= r1_n+1;
39             nn_wr <= '1';
40             r <= r1_n;
41             r_wr <= '1';
42             state <= R1b;
43           end if;
44         when R1b =>
45             nn_wr <= '0';
46             r_wr <= '0';
47           state <= Ready;
48       end case;
49     end if;
50   end process;
51 end FSM;
```

Listing 1.1. VHDL code generated for the `inc` box