Harnessing Parallelism in FPGAs Using the Hume Language

J. Sérot
Institut Pascal, UMR 6602 CNRS / U. Blaise Pascal,
Clermont-Ferrand, France
Jocelyn.Serot@univ-bpclermont.fr

G. Michaelson
Heriot-Watt University, Edinburgh, Scotland
G.Michaelson@hw.ac.uk

Abstract

We propose to use Hume, a general purpose, functionally inspired, programming language, initially oriented to resource-aware embedded applications, to implement fine-grain parallel applications on FPGAs. We show that the Hume description of programs as a set of asynchronous boxes connected by wires has a very natural interpretation in terms of register-transfer level hardware description, hence leading to efficient implementations on FPGAs. The paper describes the basic compilation process from a subset of Hume to synthetisable RTL VHDL and show preliminary experimental results obtained with a very simple perceptron application.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications–Functional Language; D.3.4 [*Programming Languages*]: Processors–compilers; B.7 [*Integrated Circuits*]: Types and Design Styles–gate arrays

General Terms Languages

Keywords Functional programming, reconfigurable hardware, FPGA, Hume, VHDL

1. Introduction

Digital circuits based upon reconfigurable logic (FPGAs) offer large opportunities for exploiting massive, fine grain parallelism. In many application domains, FPGAs are now 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.

FHPC'12, September 15, 2012, Copenhagen, Denmark.

In this paper, we propose to use Hume, a domain-specific language (DSL), initially developed for programming resource aware, software embedded systems, to program FPGAs.

Our motivations are two-fold. First is our perception that while the strong capabilities for performance prediction and resource usage certification have already been clearly demonstrated [10, 11], Hume's ability to naturally describe fine-grain parallel computations – such as those supported by FPGA targets – has been left largely unexplored. Second is the very pragmatic concern suggesting that re-inventing yet another programming language for this kind of task was not a good idea and that in the domain of embedded programming – in which Hume has already been promoted – a real need for high-level programming languages for exploiting FPGAs existed.

The paper is organised as follows : Sec. 2 is a brief presentation of the Hume programming language. In Sec. 3 we describe how a subset of Hume (called mHume) can be compiled down to synthetizable VHDL. Some preliminary experimental results are given in Sec. 4. Sec. 5 makes a brief review of related work.

2. Hume

Hume [9] 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.

The simple example in Figures 1 and 2, from [2], generates the squares of a sequence of integers. The box `inc` generates successive integers starting from `0`. These are fed to the box `square` which finds their squares by repeated addition.

- Line 1 introduces `integer` as an alias for `int 32`, that is a 32-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.

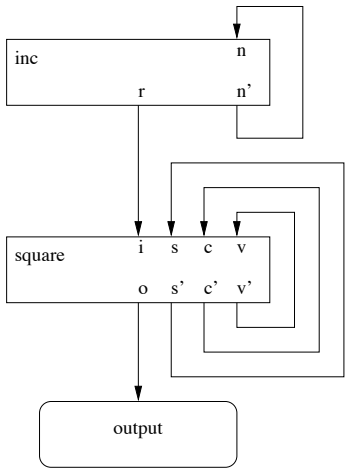

Figure 1. Square program.

```
 1 type integer = int 32;

 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);
```

Figure 2. Square program Code

- 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`.
- 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`.

2.1 Hume for hardware

Hume was initially designed to program software systems running on sequential hardware, that is CPUs running embedded applications, with a stress on predictable resource consumption. We think that the language offers interesting opportunities for programming FPGAs and in particular to exploit the massive fine grain parallelism that these devices offer without requiring deep knowledge of underlying hardware.

First, Hume's explicit separation of coordination and control layers offers an appropriate degree of abstraction for going from software-based specification to hardware realisation. We think that this explicit separation of coordination and computation makes Hume particularly well suited for reasoning about, as well as constructing, parallel systems. Formalisms for parallelism, like the π calculus, tend to focus on coordination, while those for functional languages, like BMF, focus on recursive and compositional reasoning. However, reasoning about coordination invariably has implications for computation, and vice versa, and neither considerations take account of pragmatic aspects of parallelism like time and space behaviour. In contrast, Hume is supported by the integrated box calculus [7]. This provides a small set of base transformations for introducing and eliminating boxes and wires, and for moving activities between coordination and computation. From this base set, richer transformations have been elaborated and proved correct, for example to realise function composition as vertical pipeline parallelism [8], and map [8] and fold [13] over lists as divide and conquer parallelism. Thus, an initial pure functional expression of a program may be systematically refined into interconnected boxes for potential parallel realisation. Furthermore, the box calculus may, in principle, be used for resource directed program transformation, as transformations have predictable effects on construct costs.

Second, the expression layer is state free, with all local variable instantiations lost between execution cycles, and the coordination layer state is only retained on wires. In particular, explicit 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. This addresses many problems encountered in other high level approaches to FPGA programming, in which complex synchronisation protocols must be made explicit at the expression level.

Third, 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. In particular, Finite State Hume (FSM-Hume)[12] – which augments HW-Hume with fixed size types and arithmetic/logic operations – is likely to be an excellent starting point for expressing massively parallel applications to be implemented on FPGAs. 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.

2.2 mHume

In this paper, we used a restricted version of the full Hume language, named mHume [2]. mHume is based around the full coordination layer but provides a minimal expression layer with integer types and operations. This restriction provides more flexibility for exploring the direct compilation of the language on the target hardware, without interfering with the essential issues of parallelism and coordination.

A simplified version of the mHume syntax is summarised in Figure 3.

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

Figure 3. mHume syntax.

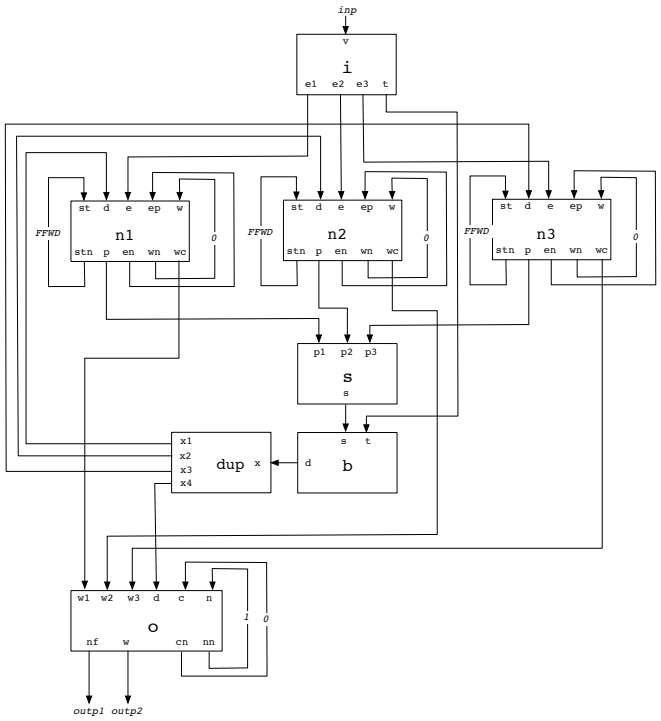

Figure 4. Perceptron program.

Figure 4 and listings 1 and 2 give the description in mHume of a very basic single-layer *perceptron*[14] which can learn how to compute any linearly separable two-inputs binary function.

Basically, the goal of this application is to compute a set $W = \{w_j\}_{j=0\ldots M}$ of factors (called *weights*) such that a given binary function $f(e_1, \ldots, e_M)$ (where $e_i \in \{0, 1\}$) can be computed as $H(\sum_{j=0}^{M} w_j\ e_j)$, where H is the Heaviside function and $e_0 = 1$. This set W is obtained by *learning*, using a *training set* $D = \{x_i, t_i\}_{i=1\ldots N}$, where

- $x_i = \{e_{i,j}\}_{j=1\ldots M}$ is an *input vector*,
- t_i is the desired (expected) output of the perceptron for that input vector.

Learning operates by successive steps. At step i :

1. the i^{th} element of the training set is read,
2. the output of the perceptron is computed, as $s_i = H(W.x_i) = H(\sum_{j=0}^{M} w_j\ e_{i,j})$ (*feed-forward* phase),
3. the computed output s_i is compared to the expected output t_i, giving a *correcting factor* $\Delta = s_i - t_i$,
4. the weights in W are updated accordingly : $w_j \leftarrow w_j + \Delta.e_{i,j}$.

This proceeds until stabilisation, which is detected when the weights are not modified ($\Delta = 0$) for at least M successive steps. It has been shown that this algorithm converges in a finite number of steps if the data set is linearly separable. In our case, the initial set of weight is set (arbitrarily) to $\{0, \ldots, 0\}$. If the learning set is exhausted before stabilisation, it is repeated as needed.

In the corresponding program (listings 1 and 2) :

- box `i` reads the training set on the input stream as a sequence of vectors and outputs the input vector and the expected output. For instance, the training set for a perceptron learning a two-input OR function will be given as

```
<< 0 0 0 >>
<< 0 1 1 >>
<< 1 0 1 >>
<< 1 1 1 >>
<< 0 0 0 >>
<< 0 1 1 >>
...
```

- The `n1`, `n2` and `n3` boxes implement the three neurons composing the single layer of this very simple perceptron[1]. Each of these boxes alternate between two modes of behavior, governed by the value `st`. If `st=FFWD` (feed-forward), the product $p_j = w_j * e_j$ is computed and output. If `st=UPDW`, the current weight w_j is updated using the $Delta$ value fed back by the `b` box and the saved value of the input e_i.
- The `s` box just sums and thresholds the products p_i computed by the neurons in the feed-forward phase.
- The `b` box computes the Δ correcting factor by comparing the computed output to the expected output.
- The `o` box controls the iterations of the algorithm. It counts (`c`) the number of steps for which Δ (`d`) is 0. As soon as this count reaches a predefined value `L` – which is normally set to the effective length of the learning set, 4 in our case –, it outputs the number `n` of steps performed so far along with the final value of the weights.

[1] In listing. 1, the code of `n2` and `n3` is identical to that of `n1` and has been omitted.

Listing 1. Perceptron program Code

```
stream inp from "or2_training.dat";
stream outp1 to "out1.dat";
stream outp2 to "out2.dat";

type dint = int 6;
type state = int 1;
constant FFWD = 0; -- Feed-forward
constant UPDW = 1; -- Update weights
constant L=4; -- learning_set_length

box i
 in (v::vector 3 of dint)
out (e1::dint, e2::dint, e3::dint, t::dint)
match
 (v) -> (1, v@0, v@1, v@2);

box n1
 in (st:: state,  -- state
     d:: dint,    -- DeltaW
     e:: dint,    -- input
     ep:: dint,   -- saved input
     w:: dint)    -- current weight
out (stn:: state, -- next state
     p:: dint,    -- output
     en:: dint,   -- saved input
     wn:: dint,   -- recirculated weight
     wc:: dint)   -- copy for output
match
  (FFWD, *, e, *, w) -> (UPDW, e*w, e,      w, w)
| (UPDW, d, *, e, w) -> (FFWD,    *, *, w+d*e, *);

box n2, n3 ... -- idem n1

box s
 in (p1::dint, p2::dint, p3::dint)
out (s::dint)
match
  (p1, p2, p3) -> if p1+p2+p3 > 0 then 1 else 0;

box b
 in (s::dint,  -- computed response
     t::dint)  -- expected response
out (d::dint)  -- DeltaW
match
  (s,t) -> t-s;

box dup
 in (x::dint)
out (x1::dint, x2::dint, x3::dint, x4::dint)
match
 (x) -> (x, x, x, x);
```

- The `dup` box simply broadcasts the value Δ computed by `b` to `n1`, `n2`, `n3` and `o` for updating the weights and potential output respectively[2].

3. Compiling Hume for FPGAs

The "classical" Hume's tool chain for implementing Hume on CPUs is 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;

[2] The dynamic semantics of Hume does not allow a wire to connect one output to several inputs.

Listing 2. Perceptron program Code (continued)

```
box o
 in (w1::dint, w2::dint, w3::dint,
     d::dint, c::int 8, n::int 8)
out (nf::int 8, w::vector 3 of dint,
     cn::int 8, nn::int 8)
match
  (w1, w2, w3, d,   c, 0) -> (*, *, *, *) -- done
| (w1, w2, w3, 0,   0, n) -> (*, *, 1, n+1)
| (w1, w2, w3, 0, L, n) -> (n, <<w1,w2,w3>>, L, 0)
| (w1, w2, w3, 0,   c, n) -> (*, *, c+1, n+1)
| (w1, w2, w3, d,   c, n) -> (*, *, 0, n+1);

wire i (inp) (n1.e, n2.e, n3.e, b.t);
wire n1 (n1.stn initially 0, dup.x1, i.e1,
         n1.en, n1.wn initially 0)
        (n1.st, s.p1, n1.ep, n1.w, o.w1);
wire n2 (n2.stn initially 0, dup.x2, i.e2,
         n2.en, n2.wn initially 0)
        (n2.st, s.p2, n2.ep, n2.w, o.w2);
wire n3 (n3.stn initially 0, dup.x3, i.e3,
         n3.en, n3.wn initially 0)
        (n3.st, s.p3, n3.ep, n3.w, o.w3);
wire s (n1.p, n2.p, n3.p) (b.s);
wire b (s.s, i.t) (dup.x);
wire dup (b.d) (n1.d, n2.d, n3.d, o.d);
wire o (n1.wc, n2.wc, n3.wc, dup.x4,
        o.cn initially 0, o.nn initially 1)
       (outp1, outp2, o.c, o.n);
```

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

The first and easier way for executing Hume program on a FPGA is to use a *soft-core*-based approach [1, 2] and have the CPU core(s) implemented on the FPGA and executing either

- the HAM interpreter, itself executing HAM code;
- HAM code compiled to native code;
- Hume programs compiled directly via C to native code.

All these routes can offer consistent, scalable speedup but the scalability is ultimately limited by the number of cores that can implemented on a FPGA (typically a few dozens on a high-end FPGA with the current technology). This coarse-grained approach therefore cannot exploit the full potential of massive fine grain parallelism offered by FPGAs. It also generally leads to a considerable waste of hardware resources since it frequently happens that not all the computational units of the instantiated CPU cores are required to run a specific application. Finally, because of the relatively limited clock frequencies, the solutions are, in most cases, markedly slow compared with the equivalent routes on proprietary CPUs.

Fully exploiting the huge amount of fine grain parallelism offered by FPGAs requires a more radical approach. In the current state-of-the-art, what is needed is a *register transfer level* description of the application. 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. RTL descriptions are typically written using hardware description languages such as VHDL or Verilog and are accepted by hardware synthetizers provided by FPGA vendors to produce optimized, target-specific, gate-level netlists.

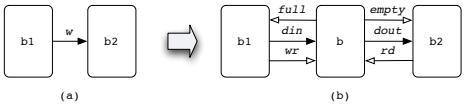

Figure 5. Network generation. (a) Initial box structure (b) After buffer insertion

In the sequel, we therefore describe a compilation process, transforming mHume programs into synthetizable, RT-level VHDL. This process basically involves three phases : network generation, box translation and VHDL transcription.

3.1 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. 5. 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.

3.2 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. Often, and as pointed out by G. Berry in [4] for instance, complex software solutions become trivial when described in hardware, because parallelism comes for free at this level. Here, this dramatically simplifies 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
      if a fireable rule r can be found in b.rules
         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
         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. 6. Each transition in the resulting FSM is labelled 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 :

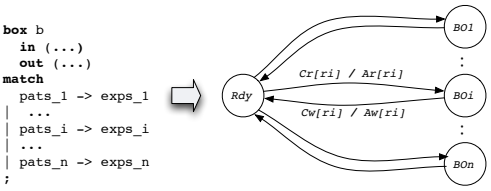

Figure 6. Translation of a box into a FSM

- 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 expression exp and write the corresponding value on the j^{th} output"[3].

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. 2.2 is given in Fig. 7.

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

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

[3] 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.

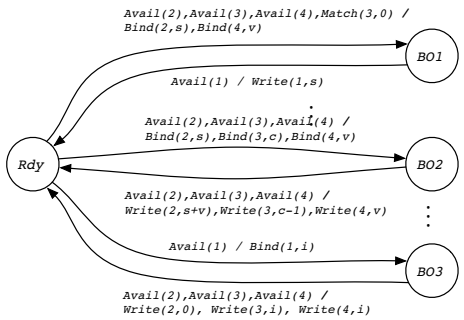

Figure 7. FSM for the `square` box

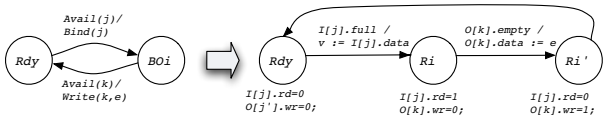

Figure 8. Transformation of the FSM to generate the `rd` and `wr` signals (I[j] and O[k] respectively refer to the j^{th} input and k^{th} output of the box)

3.3 Transcription to VHDL

The transcription in VHDL of the network derived in Sec. 3.1 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 outputs of this component correspond to the I/O streams declared in this program. This makes it possible to automatically generate a *testbench* 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. 8 on a simple, mono-rule, example.

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. 3 gives the VHDL code generated for the `inc` box of the example introduced in figures 1 and 2:

- Lines 1-13 give the interface of the component. Hume integers are translated to VHDL `std_logic_vectors`. As explained in Fig. 5, the `n`, `n_empty` and `n_rd` signals correspond to the `n` original input. Similarly, the `r`, `r_full` and `r_wr` (resp. `nn`, `nn_full` and `nn_wr`) signals correspond to the `r` (resp. `n'`) original output. The two other input signals are the global clock and a reset for hardware initialization.
- The behavior of the box is made explicit in its architecture, lines 15-51. This architecture describes a synchronous FSM.
- The state variable is declared in line 17, its type being declared in line 16. Here the box has only one rule, so there are three states. The behavior itself is made explicit as a *process* sensitive to the `clock` and `reset` signals (line 19). This process uses a internal variable `r1_n`.
- This variable memorizes the value obtained when the pattern of rule *r1* is bound (line 31)[4].
- The core of the process – which, according to VHDL execution model, is executed whenever the signal `clock` or `reset` changes value – is between line 21 and 50.
- Lines 22-26 handles asynchronous reset : the process state is reset to *Ready* and read/write signals are set to 0.
- Lines 28-48 describe what happens when a rising edge occurs on the `clock` input signal. This part is written in a 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 30-33 require that if process (box) is in the *Ready* state and a value is available on input `n` (line 30), then this value is copied (line 31), the read signal is asserted (line 32) and the next state will be `R1a` (line 33).
- In state `R1a` (lines 36-42), the read signal is reset to 0 and the availability of the output link is tested (line 37). If yes, the outputs are written (line 38-41) and the next state will be `R1b`.

4. Experimental results

Evaluation of the generated VHDL code has been carried out using the Altera Quartus II v9.0 tool chain, first by simulating the generated RTL code and then by synthetizing it on a target FPGA.

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 `perceptron` example introduced in Sec. 2.2, are displayed in Fig. 9 for the training set of a two-input OR function mentioned in Sec. 2.2. Trace names match those given in Fig. 4 and listing 1; for example `n1.w` shows the evolution of the `w` output of box `n1`. The clock period has been arbitrarily fixed to 10 ns and input vectors are input every 16 clock cycles. The program correctly terminates with the following outputs : `outp1=14`, `outp2=«0 1 1»`.

We performed the synthesis of this example on a *Stratix* EP1S80 FPGA. This is a medium-sized device, embedding 79040 logic cells and 7 Mbits of RAM. The default parameters for the synthetizer were used. The synthetized solution occupies 687 cells (less than 1%) and runs at a maximum clock frequency of 142 MHz. Fig. 10 is a top level view of the synthetized network. The four bigger boxes implement the `n1`, `n2`, `n3` and `o` boxes. The mid size boxes correspond to the `s` and `b` boxes and the smaller boxes represent buffers. Fig. 11 shows the gate-level implementation of a buffer for a 1-bit wide wire. Fig. 12 shows the hardware architecture inferred by the synthetizer for the `s` box. The most easily recognizable elements are the collection of registers (memorizing the inputs `p1`, `p2` and `p3` and the output `s`) and the two adders and comparator (drawn as small circles) which perform the basic function of the box. The rectangular box at center left implements the FSM control.

5. Related work

There have been a number of functional approaches to parallel and/or FPGA programming, drawing on the classic FP strength of higher order abstraction to compose components for hardware realisation.

[4] 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.

Listing 3. VHDL code generated for the `inc` box

```
1  entity inc_box is
2     port ( n_empty: in std_logic;
3            n: in std_logic_vector(31 downto 0);
4            n_rd: out std_logic;
5            r_full: in std_logic;
6            r: out std_logic_vector(31 downto 0);
7            r_wr: out std_logic;
8            nn_full: in std_logic;
9            nn: out std_logic_vector(31 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(31 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;
```

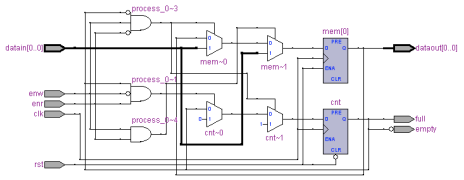

Figure 11. Gate-level architecture of a 1-bit buffer

Lava [5] augments Haskell with modules for hardware description. The Lava tool chain generates VHDL. Sheeran [16] provides a useful reflection on Lava's origins. Several groups are actively developing Lava, most noticeably Kansas Lava [6].

CλaSH [3] is another language/toolchain for translating a subset of Haskell into synthetizable VHDL.

The Kiwi project has recently been complemented with the use of F# [17], a Standard ML derivation. Here, common middleware for all .Net compliant languages eases the route to VHDL.

Gannet [18] 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.

In a slightly different context, the CAPH dataflow language [15], for programming real-time stream-processing applications on FPGAs, share many ideas with Hume. Both are based between a clean distinction between an expression layer for expressing the behavior of individual boxes and a coordination layer. In both languages, behavior is expressed as a set of transition rules using pattern matching. But the execution models are different since CAPH models box interconnections as buffering, FIFO channels and boxes (actors) can hold state variables.

6. Conclusion

We have presented an approach to the automatic generation of FPGA configurations from mHume programs and have shown, for small examples, that it can achieve good silicon utilisation and performance.

The work presented should essentially be viewed as a proof-of-concept. We plan to extend and improve it in several ways.

First, by expanding the expressiveness of the computation layer to encompass a larger subset of the full Hume language (while keeping a tractable path down to RTL code).

Second, by integrating a macro-language for specifying complex networks in a modular fashion. Such a language could be derived, for instance, from the box template / instanciation mechanism introduced in the latest version of the Hume language.

Third, by trying to apply the Hume box calculus [7, 8, 13] to the (semi)-automatic derivation of the relatively low-level formulations of mHume programs from higher levels specifications (as a set of function calls for instance).

Fourth, by exploring the development of static analyses at the mHume level that will enable prediction of time and space behaviour, and of silicon occupancy, of FPGA implementations.

Fifth, and correlatively, by systematically evaluating larger and more complex examples, to assess how well the approach scales.

Most of these extensions will take advantage of Hume's explicit separation of coordination and computation. The coordination constructs described here are common to all Hume programming levels, providing a good foundation for further implementation. We also plan to exploit the strong similarities between the Hume and CAPH languages at the computation level.

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] A. Al Zain, G. Michaelson, and W. Vanderbauwhede. mHume for Parallel FPGA. In *Draft proceedings of 22nd International Symposium on Implementation and Application of Functional Languages, Amsterdam*, September 2010.

[3] C. Baaij, M. Kooijman, J. Kuper, W.A. Boeijink and M. Gerards. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In Proceedings of the 13th EUROMICRO Conference on

Digital System Design: Architectures, Methods and Tools (DSD 2010), Sep 2010, Lille, France.

[4] G. Berry. *Penser, modéliser et maîtriser le calcul informatique*. Fayard, 2009.

[5] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. *SIGPLAN Not.*, 34:174–184, September 1998.

[6] A.Gill, T. Bull, G. Kimmell, E.Perrins, E. 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.

[7] G. Grov. Reasoning about correctness properties of a coordination language. PhD, Heriot-Watt University, 2009.

[8] G. Grov and G. Michaelson. Hume box calculus: robust system development through software transformation. *Higher Order Symbolic Computing*, Vol 23, No 2, pp 191-226, July, 2012.

[9] 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 Comp. Eng., Erfurt, Germany*, pages 37–56. Springer-Verlag LNCS 2830, Sep. 2003.

[10] K. Hammond, C. Ferdinand, R. Heckmann, R. Dyckhoff, M. Hoffmann, S. Jost, H-W. Loidl, G. Michaelson, R. Pointon, N. Scaife, J. Sérot and A. Wallace. Towards Formally Verifiable Resource Bounds for Real-Time Embedded Systems. In *ACM SIGBED Review- Special issues on Workshop on Innovative Techniques for Certification of Embedded Systems 2006 (ITCES06)*, 3(4), October 2006, pp 27-36.

[11] C. A. Herrmann, A. Bonenfant, K. Hammond, S. Jost, H-W. Loidl and R. Pointon. Automatic Amortised Worst-Case Execution Time Analysis. In *7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis*, Pisa, Italy, July, 2007,pp 13-18.

[12] , G. Michaelson, K. Hammond and J. Serot. FSM-Hume is Finite State. In S. Gilmore (Ed), *Trends in Functional Programming 4*, Intellect, 2004, pp 19-28.

[13] G. Michaelson and G. Grov. Reasoning about multi-process systems with the box calculus. In *Proceedings of 4th Central European Summer Functional Programming School (CEFP 2011)*, Springer,2012.

[14] , F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. Republished in J.A. Anderson and E. Rosenfeld (Eds), *Neurocomputing. Foundations of Research*, MIT Press, 1988.

[15] J. Sérot, F. Berry and S. Ahmed. Implementing stream-processing applications on FPGAs : a DSL-based approach. In *21st International Conference on Field Programmable Logic and Applications, Chania, Crete, Sep 2011*

[16] M. Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

[17] S. Singh. Kiwi Synthesis of C# and F# Combinational Circuit Models into FPGA Circuits. *Satnam Singh's MSDN Blog*, April 2010.

[18] 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.

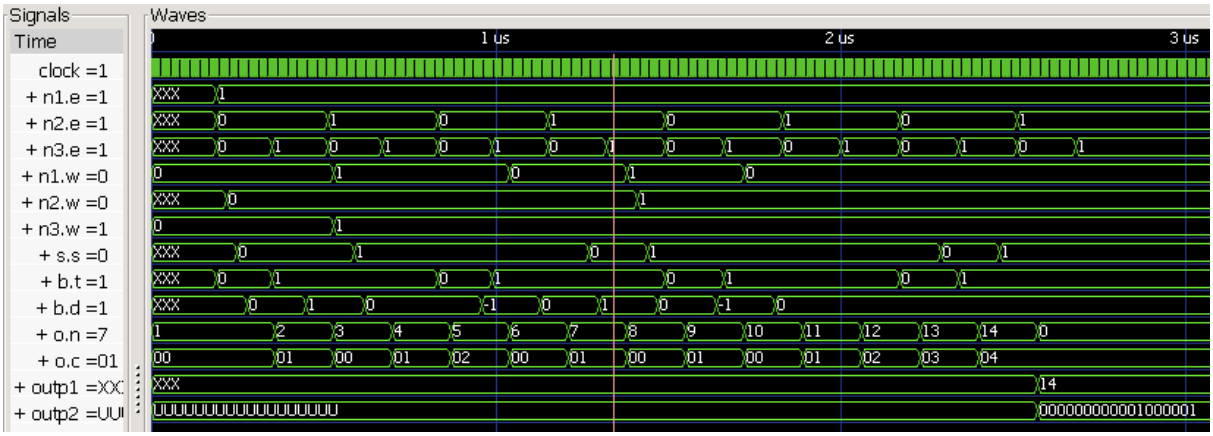

Figure 9. Simulation results for the `perceptron` example

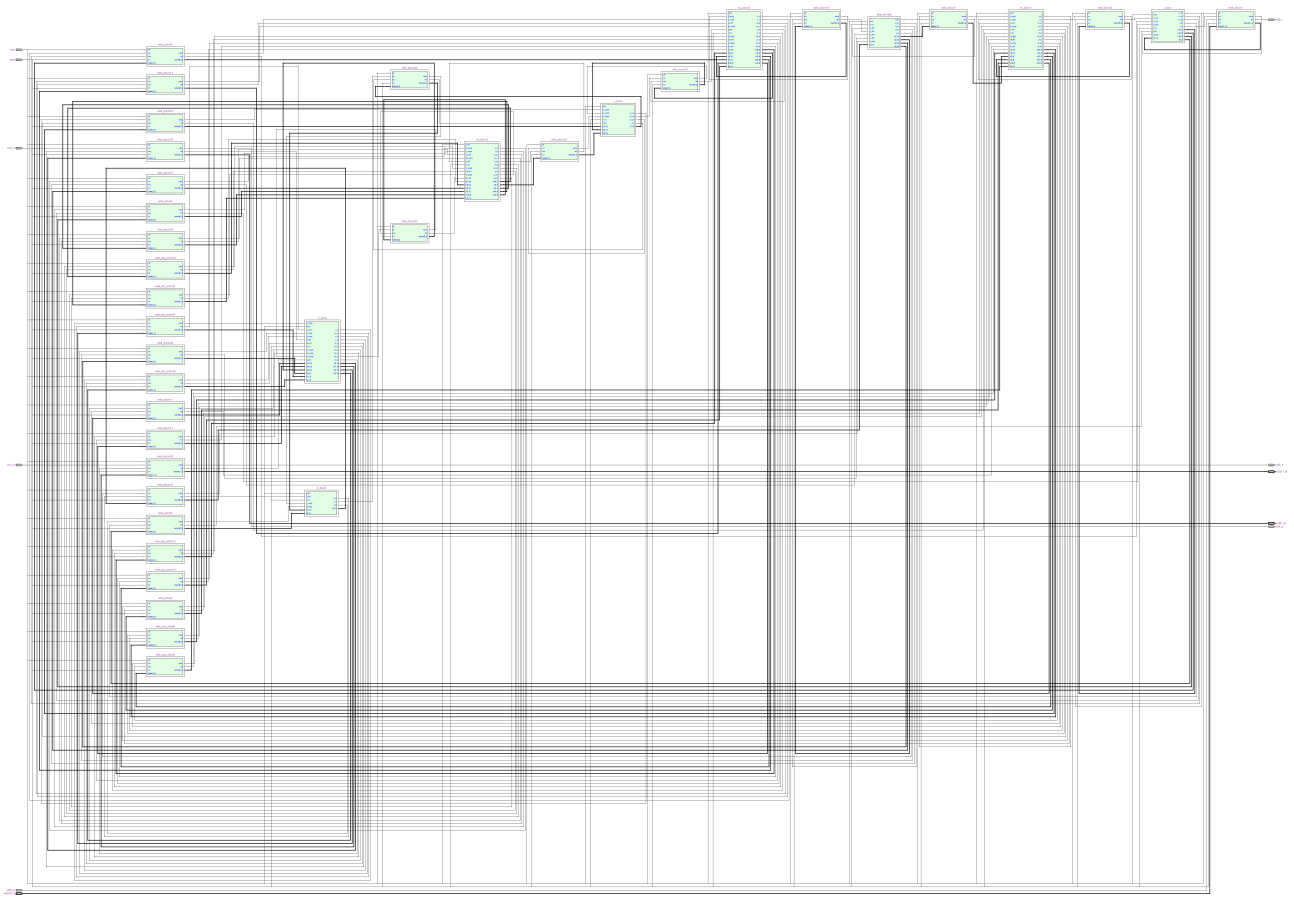

Figure 10. Top-level synthetized network for the `perceptron` example

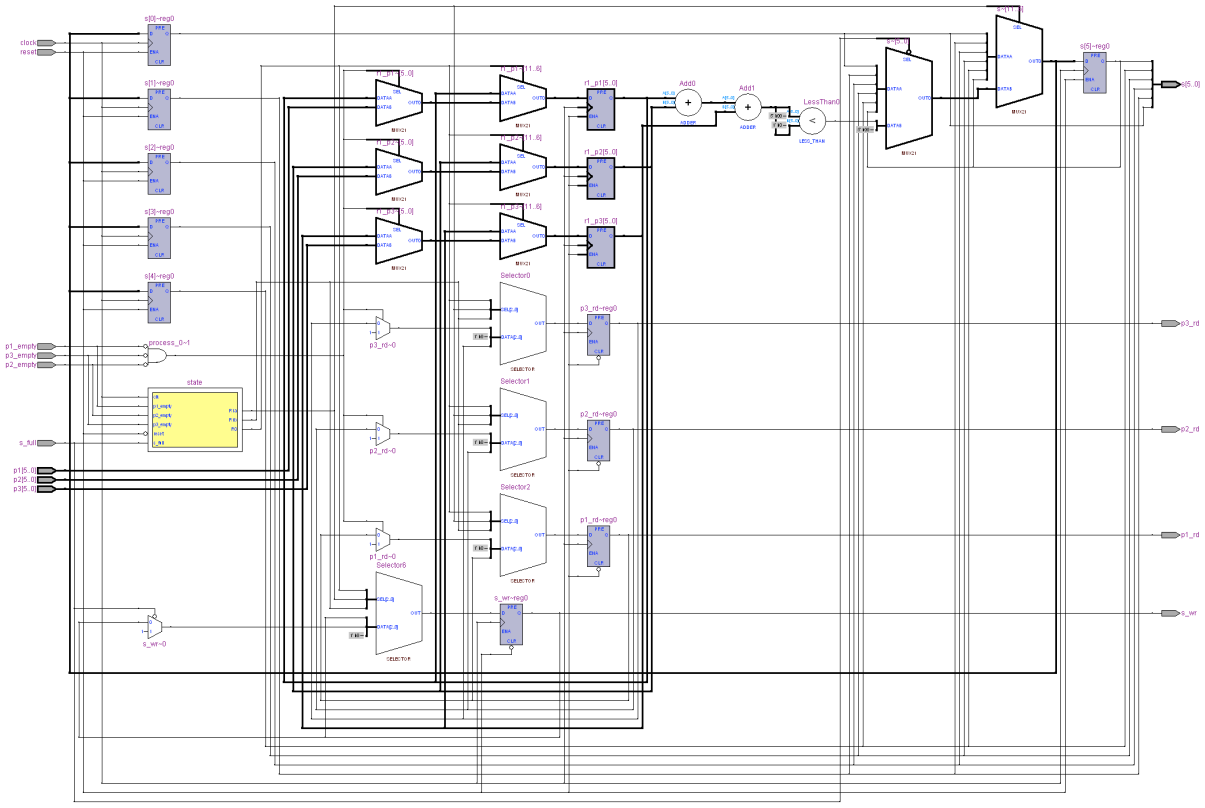

Figure 12. Synthetized RTL architecture for the `s` box