# Low-Level Programming in Hume: An Exploration of the HW-Hume Level

Kevin Hammond[1], Gudmund Grov[2], Greg Michaelson[2], and Andrew Ireland[2]

[1] School of Computer Science,
University of St Andrews, St Andrews, Scotland
Tel.: +44-1334-463241
kh@dcs.st-and.ac.uk
[2] Dept. of Mathematics and Computer Science,
Heriot-Watt University, Edinburgh, Scotland
Tel.: +44-131-451-3422
{gudmund,air,greg}@macs.hw.ac.uk

**Abstract.** This paper describes the HW-Hume level of the novel Hume language. HW-Hume is the simplest subset of Hume that we have identified. It provides strong formal properties but posseses limited abstraction capabilities. In this paper, we introduce HW-Hume, show some simple example programs, describe an efficient software implementation, and demonstrate how important properties can be exposed as part of an integrated formally-based verification approach.

## 1 Introduction

The novel Hume language embeds a strict, purely functional *expression layer*, that describes computations, within a *process layer*, that describes a system of asynchronous communicating processes. By varying the structure of the Hume expression layer, a number of distinct Hume levels can be identified, where each level fully contains the level below, but increases the difficulty of providing accurate cost information and other properties. *Full-Hume*, or *Hume*, is a Turing-Complete language based on concurrent finite state automata whose transitions are controlled by pattern matching over rich types to initiate actions described by general recursive expressions. *PR-Hume*, restricts repetition to primitive recursion, enabling decidable termination. *Template-Hume* only permits repetition through pre-defined higher-order operators. *FSM-Hume* is a finite-state language with fixed size types and first order functions. Finally, *HW-Hume*, aimed at hardware realisation, is a relatively impoverished language for manipulating tuples and vectors of bits, with exact time and space use prediction.

We have previously introduced the Hume language [22], defining the different levels of Hume, as outlined above, and shown how translations may be made between levels [21]. We have also demonstrated that it is possible to construct bounded space cost models for FSM-Hume [23], and for time and space up to PR-Hume [34,35]. We are in the process of constructing automatic analyses to provide bounds on amortised time and space cost information on levels up to PR-Hume. This paper considers HW-Hume in considerably more depth than in the general papers mentioned above [21,22]. Section 2 introduces HW-Hume and provides some simple examples; Section 3 discusses formal verification of safety, liveness and real-time properties using model-checking; Section 4

Z. Horváth, V. Zsók, and A. Butterfield (Eds.): IFL 2006, LNCS 4449, pp. 91–107, 2007.

describes a software implementation of HW-Hume and provides some performance results; Section 5 discusses possible hardware implementations; Section 6 describes related work; and finally, Section 7 concludes.

## 2 HW-Hume

HW-Hume programs (Figure 1) are built from a series of *box* declarations linked using static *wires*. Multiple identical instances of a box may be defined using a *template* for subsequent *instantiation*. A single HW-Hume box comprises a set of pattern-directed rules, rewriting a set of inputs to a set of outputs, plus appropriate type information for each input/output. The most primitive type of value is a bit, which may be grouped into fixed-size vectors or tuples in either a pattern or an expression. Patterns and expressions may be formed from bit literals, variables, the wildcard pattern _, vector or tuple structures, or (at the top level) the asynchronous * construct, which ignores its input and produces no output. $\tau$ defines the valid HW-Hume types: bit types, **word** 1; the unit type, (); tuple types $\tau_1 \times \ldots \times \tau_n$; bounded vector types, **vector** $n$ **of** $\tau$, where $n$ is the bound; and named types, *typeid*.

### 2.1 Boxes and Coordination

HW-Hume *boxes* are abstractions of processes that correspond to (usually finite) state machines. The left-hand-side (pattern part) of each rule defines the situations in which that rule may be *active*, i.e. could be executed. The right-hand-side of each rule is an expression specifying the results of the box when the rule is activated and matches the corresponding pattern. A box may become active when any of its rules are active, i.e.

| | | |
|---|---|---|
| *program* ::= | $decl_1$ ; ... ; $decl_n$ ; | $n \geq 1$ |
| *decl* ::= | *box* \| *wire* \| *type* \| *template* \| *instantiation* | |
| | | |
| *box* ::= | **box** boxid *ins outs* ( **match** \| **fair** ) *matches* | |
| *ins/outs* ::= | ( $\text{ioid}_1$ :: $\tau_1$ , ... , $\text{ioid}_n$ :: $\tau_n$ ) | $n \geq 0$ |
| $\tau$ ::= | **word** 1 \| () \| ($\tau_1$ , ... , $\tau_m$) \| **vector** $n$ **of** $\tau$ \| typeid | $m \geq 2, n \geq 1$ |
| | | |
| *matches* ::= | $match_1$ \| ... \| $match_n$ | $n \geq 1$ |
| *match* ::= | ( $pat_1$ , ... , $pat_n$ ) → *expr* | $n \geq 0$ |
| *expr*/*pat* ::= | 0 \| 1 \| varid \| _ \| * \| () \| ( $expr_1/pat_1$ , ... , $expr_n/pat_n$ ) | $n \geq 2$ |
| | \| **vector** $expr_1/pat_1$ ... $expr_n/pat_n$ | $n \geq 1$ |
| | | |
| *wire* ::= | **wire** $link_1$ **to** $link_2$ [ **initially** *expr* ] | |
| *link* ::= | boxid . ioid \| deviceid | |
| | | |
| *type* ::= | **type** typeid = $\tau$ | |
| *template* ::= | **template** templateid *ins outs* ( **match** \| **fair** ) *matches* | |
| *instantiation* ::= | **instantiate** templateid **as** boxid [ * nat ] | |

**Fig. 1.** HW-Hume Syntax

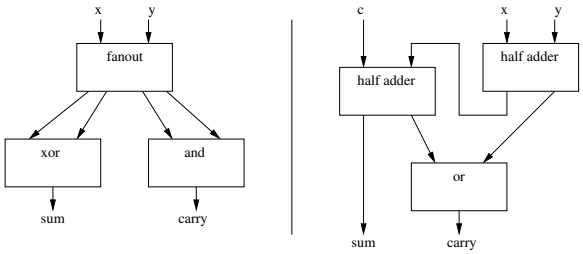


**Fig. 2.** a) Half-adder b) Full-adder

they may match the inputs that have been provided. In this case, the box runs to completion, producing any required outputs. For example, we can define boxes to implement *xor*, *and*, and a two-in to four-out *fanout* as:

```
box xor
in (a,b::Bit)
out (x::Bit)
match
  (1,1) -> 0
| (0,0) -> 0
| (_,_) -> 1;
```

```
box and
in (a,b::Bit)
out (x::Bit)
match
  (1,1) -> 1
| (_,_) -> 0;
```

```
box fanout
in (x,y::Bit)
out (x1,y1,x2,y2::Bit)
match
  (x,y) -> (x,y,x,y);
```

For each box, we first specify the names and types of the inputs and outputs – here all single bits. Note that boxes may use the same names: these are always qualified externally by the box name. We then specify the pattern-matching rules that take the given inputs and produce the correct output. The final rule in the first two cases uses "anonymous" variable patterns, defined using `_`. HW-Hume boxes are connected into a static process network using wires to connect one specific output to one specific input. For example, Figure 2 shows: (a) a *half adder* built from a `fanout` box, an `and` box and an `xor` box; (b) a *full adder* built from two half adders and an `or`, where the half adder is either built from simpler components as in (a) or defined in its own right from a truth table:

```
box or
in (a,b::Bit)
out (x::Bit)
match
  (0,0) -> 0
| (_,_) -> 1;
```

```
box half-adder
in (x,y::Bit)
out(s,c::Bit)
match
  (0,0) -> (0,0)
| (0,1) -> (1,0)
| (1,0) -> (1,0)
| (1,1) -> (0,1);
```

In either case, the boxes are then wired into a static process network using the obvious wiring declarations. The use of a static process network allows strong program properties to be obtained, as discussed in Section 3.

**Asynchronous Language Constructs.** Unlike the widely-used synchronous languages for real-time systems, such as Lustre [10], Signal [17] or Esterel [8], HW-Hume is an

asynchronous language, allowing the expression of hardware/software systems that are not explicitly clocked, and where individual boxes may produce outputs without synchronising on their inputs. The two main mechanisms for asynchronicity in Hume are to allow some or all inputs/outputs to be *ignored*, using $*$, and to allow *fair matching* on rules, where on each box cycle the first rule considered is that after the one that succeeded on the previous cycle. The *-pattern indicates that the corresponding input position should be ignored, i.e. the match always succeeds without demanding any input. * can also be used in a top-level expression position. For example, a multiplexer can be described by the rules below, where the fourth rule will discard the selector if no other input is available.

```
type Bit = word 1;
type Byte = vector 8 of Bit;
type Selector  = (Bit,Bit);

box multiplexer
in  (b1, b2, b3 :: Byte, sel :: Selector)   out  (b :: Byte)
fair
    (b,  *,  *,          (0,0)) -> b
|   (*,  b,  *,          (0,1)) -> b
|   (*,  *,  b,          (1,0)) -> b
|   (*,  *,  *,          _    ) -> *;
```

Note that, in this example, although there is no explict input clock signal, the selector input acts as a trigger, effectively requiring synchronisation between the selector and the corresponding input. A more asynchronous version can be produced, if required, by simply eliminating the selector input.

```
box multiplexer2
in  (b1, b2, b3 :: Byte)   out  (b :: Byte)
fair
    (b,  *,  *) -> b
|   (*,  b,  *) -> b
|   (*,  *,  b) -> b;
```

Now each input is immediately mapped to the output without waiting for some selector to be present. Outputs are chosen from the three possibilities fairly [5,22]. Multiplexing is an example of an operation that cannot easily be expressed in a single-layer purely functional notation, since it is non-deterministic at the box level. Despite this local idea of non-determinacy (an essential part of the problem specification), it is important to realise that *the system as a whole* is still deterministic in that it will respond identically to the same inputs received at the same relative times [22].

### 2.2 A Simple Traffic Lights Example in HW-Hume

As a more detailed example, we consider a set of traffic lights, as used in the UK, which displays a sequence of red (stop), red and amber (prepare to go), green (go) and amber (prepare to stop) lights[1]. We might encode these state changes as:

[1] A variant of this example has also been used in [21] to illustrate inter-level transformations.

| light(s) | state | meaning | red | amber | green |
|---|---|---|---|---|---|
| red | 0 | stop | 1 | 0 | 0 |
| red/amber | 1 | prepare to go | 1 | 1 | 0 |
| green | 2 | go | 0 | 0 | 1 |
| amber | 3 | prepare to stop | 0 | 1 | 0 |

where a `1` indicates that the corresponding light is on and a `0` that it is off. In HW-Hume, we could model a traffic light as a box which changes state when it receives a signal. We encode the state as a two-bit binary number, and the light settings as a tuple of bits. So that we can reuse the lights definition later, we will use a template definition:

```
template trafficlights
in  (signal::Bit, state::(Bit,Bit))
out (state'::(Bit,Bit),lights::(Bit,Bit,Bit))
match
  (1,(0,0)) -> ((0,1),(1,1,0))
| (1,(0,1)) -> ((1,0),(0,0,1))
| (1,(1,0)) -> ((1,1),(0,1,0))
| (1,(1,1)) -> ((0,0),(1,0,0));

instantiate trafficlights as lights;

wire change              to  lights.signal;
wire lights.state        to  lights.state';
wire lights.lights       to  display;
```

where `change` and `display` are unspecified external connections. On each box cycle, if the `signal` on `change` is `1` then, for the current `state`, a new setting on `lights` is sent to `display` and a new state is produced on `state'`. Unlike the earlier, *combinational* examples we have seen, this is an example of *sequential logic*: it is necessary to record the `state` value as feedback between box iterations.

## 3 Verifying HW-Hume Programs

Because of the cost and difficulty involved in applying bug fixes, low-level system designs often possess strong correctness criteria. This is especially true for hardware, where there is a long tradition of using automated verification and formal methods to enhance confidence in the correctness of such systems. In particular *model checking* [12] has been successfully applied to many hardware systems. In this approach, a property is specified in a temporal logic, and its correctness against a given model (program) is verified algorithmically by exploring the complete state space of the model.

We exploit TLA$^+$ [29] which combines TLA (Temporal Logic of Actions [28]) with a variant of ZF set-theory and which allows both system (model) and properties to be specified in the same logic. The validity of a program property can therefore be

expressed by logical implication: *Program* ⇒ *Property*. This validity can then be checked by the TLC model checker [29] for TLA$^+$. TLA($^+$) also have a proof system, meaning we can give deductive proofs of properties, which will be required in the higher levels of Hume. It has a similar layering to Hume, and this together with both the algorithmic and deductive proofs support, made TLA fit really well into our work. In HW-Hume, individual box definitions are fairly simple, and the most interesting properties (and errors!) consequently arise when combining two or more boxes. We therefore illustrate our approach using a slightly extended version of the traffic light example, where two instances of the `trafficlights` template are connected to model a complete road junction under the control of the `controller` box below.

```
instantiate trafficlights as lights * 2;

box controller
in  (state ::  (Bit,Bit,Bit))
out (state' :: (Bit,Bit,Bit), lights1,lights2 :: Bit)
match
  (0,0,0) -> ((0,0,1),1,*)  -- lights1: Red -> Red-amber
| (0,0,1) -> ((0,1,0),1,*)  -- lights1: Red-amber -> Green
| (0,1,0) -> ((0,1,1),1,*)  -- lights1: Green -> Amber
| (0,1,1) -> ((1,0,0),1,*)  -- lights1: Amber -> Red
| (1,0,0) -> ((1,0,1),*,1)  -- lights2: Red -> Red-amber
| (1,0,1) -> ((1,1,0),*,1)  -- lights2: Red-amber -> Green
| (1,1,0) -> ((1,1,1),*,1)  -- lights2: Green -> Amber
| (1,1,1) -> ((0,0,0),*,1); -- lights2: Amber -> Red
```

In the remainder of this section we will discuss both safety and liveness properties of this program. We also show how time analysis of the expression layer can be combined with TLA$^+$ to verify bounded real-time properties of the coordination layer.

### 3.1 Safety Properties

A *safety property* specifies that certain undesired behaviour never occurs [4]. The safety-part of a specification therefore specifies what a good behaviour is, but does not require that something actually happens. We formalise the safety part of the traffic light example as follows. Let *Prog* denote the safety part of our program. The state space consists of $\bar{\mathsf{i}}$, all the internal variables used in any box, and $\overline{w}$, all the wires used in the program. These are given an initial value by *Init*. For each box, we define actions $\mathcal{N}_{l1}$, $\mathcal{N}_{l2}$ and $\mathcal{N}_{ctl}$ which update the state space. Since TLA$^+$ is a logic rather than a programming language, these actions are defined as predicates on a before-step and an after-step, where all variables in the after-step are primed. For example, if $l1$ can be executed then $\mathcal{N}_{l1}$ will match the (unprimed) input wires (unfairly). If it succeeds, then the (primed) input wires are set to empty since there are no $*$ in the pattern, and the primed outputs are updated with the result of the computation. A *next-action* $\mathcal{N}$ updates the complete state space, and is defined in terms of the execution of all the boxes

in the program ($\mathcal{N}_{l1} \wedge \mathcal{N}_{l2} \wedge \mathcal{N}_{ctl}$). It must have the form $[\mathcal{N}]_{(\overline{i},\overline{w})}$, which abbreviates $\mathcal{N} \vee (\overline{i},\overline{w})' = (\overline{i},\overline{w})$. This is an important feature since it allows "internal actions" that do not alter the state space. Since we are working in a temporal logic, this next-action is required to hold throughout execution. We therefore prefix the action with the temporal *always* operator ($\Box$) to give $\Box[\mathcal{N}]_{(\overline{i},\overline{w})}$. In the traffic lights program, only state transitions are specified: at any given time/state we do not know which lights are on and which are off. Since we are interested in the current colour of the lights, we introduce two auxiliary variables, $tl_1$ and $tl_2$, to expose this information in the model. These variables emulate the actual lights, allowing us to formalise the required safety properties much more naturally. We assume that the lights are initially red. The action $\mathcal{N}_{tl_1,tl_2}$ updates these variables if (and only if) the corresponding light changes colour. *Prog2* extends *Prog* with these definitions. Note that these auxiliary definition do not change the behaviour of *Prog*:

$$Prog2 \triangleq \exists\!\!\exists\, \overline{i} : Init_{tl_1,tl_2} \wedge Init \wedge \Box[\mathcal{N}_{l1} \wedge \mathcal{N}_{l2} \wedge \mathcal{N}_{ctl} \wedge \mathcal{N}_{tl_1,tl_2}]_{(\overline{i},\overline{w},tl_1,tl_2)}$$

Here the $\exists\!\!\exists$ operator is a form of existential quantification that is used to hide the state, that is, the internal variables of the boxes are hidden. The first safety property we define is an invariant asserting that both lights cannot be green at the same time:

$$Prog2 \Rightarrow \Box(tl_1 \neq (0,0,1) \vee tl_2 \neq (0,0,1))$$

The $\Box$-prefix ensures that the property holds throughout execution. This can be strengthened to show, e.g., that if one of the lights is not red, then the other light is red:

$$Prog2 \Rightarrow \Box\Big((tl_1 \neq (1,0,0) \Rightarrow tl_2 = (1,0,0)) \wedge (tl_2 \neq (1,0,0) \Rightarrow tl_1 = (1,0,0))\Big)$$

The final safety property we define is that the *order* of the light changes is correct. This is no longer a state invariant, since we need to compare two states: that before the change and that after the change. We define a pseudo-function `Next` as follows:

```
Next l = case l of (1,0,0) -> (1,1,0)
                 |  (1,1,0) -> (0,0,1)
                 |  (0,0,1) -> (0,1,0)
                 |  (0,1,0) -> (1,0,0);
```

where `Next` is a meta-level definition used in the reasoning process and not part of the HW-Hume program. We can then verify that the next-action of *Prog2* implies this change. We use `Next` $tl_1/tl_2$ to ensure that there exists a correspondence between changes in the action and the associated value.

$$Prog2 \Rightarrow \Box[tl_1' = \texttt{Next}\ tl_1]_{tl_1}$$
$$Prog2 \Rightarrow \Box[tl_2' = \texttt{Next}\ tl_2]_{tl_2}$$

The subscripts to the actions ensure that only those steps where the lights actually change value are considered. This is necessary if the formula is to be valid.

### 3.2 Liveness Properties

*Liveness properties* assert that something good will eventually occur [4]. The specification must therefore be constrained to remove non-progress behaviours. We constrain it with a type of liveness called *fairness*. There are two types of fairness, both building on the *enabled* predicate: An action is enabled when it could successfully execute. *Weak* fairness asserts that if an action remains enabled, then it will eventually execute, while *strong* fairness asserts that if an action is enabled infinitely often then it will eventually execute. The scheduling of Hume guarantees both strong and weak fairness of boxes. This is because all boxes that can be executed are always executed. Further, since the only way an executable (enabled) box can become non-executable (disabled) is by executing it, weak and strong fairness are both equivalent for Hume. Note that this notion of fairness is distinct from the notion of fair matching introduced earlier. We only require weak fairness for our proofs, extending *Prog2* with the fairness predicate for all the boxes:

$$Prog3 \triangleq Prog2 \wedge WF_{(\bar{i},\bar{w})}(\mathcal{N}_{l1} \wedge \mathcal{N}_{l2} \wedge \mathcal{N}_{ctl})$$

The first liveness property we show is that at any given time, there will always be a time in the future when the lights are green:

$$Prog3 \Rightarrow \Box\Diamond(tl_1 = (0,0,1)) \wedge \Box\Diamond(tl_2 = (0,0,1))$$

$\Box\Diamond$ is read as "always eventually". One kind of liveness property which is very important for HW-Hume programs is a so-called *leads-to* property. For example, we can specify that if $tl_1$ is red then $tl_2$ will eventually become green:

$$Prog3 \Rightarrow tl_1 = (1,0,0) \leadsto tl_2 = (0,0,1) \wedge tl_2 = (1,0,0) \leadsto tl_1 = (0,0,1)$$

where $A \leadsto B$ means that (always) when $A$ is *True* then eventually $B$ will be $True$. Note that this property is strictly weaker than the previous property, which can be specified simply as $True \leadsto tl_1 = (0,0,1)$. Note that $\Box\Diamond T$ specifies termination, i.e. the production of some result. This termination property can be strengthened to only check for termination under certain condition $P$: This is formalised as a leads-to property $P \leadsto T$.

### 3.3 Real-Time Properties

One of the novel aspects of, and indeed a prime motivation for, the design of Hume is that upper bounds on time and space can be guaranteed for the expression layer. Since HW-Hume is a language of bits, tuples and vectors, it is straightforward to produce precise models of both space and time usage. For brevity, we omit formal definitions of these models here (definitions for FSM-Hume can be found in [23]), but will show how time bounds for the expression layer obtained from such a model can be combined with TLA$^+$ to give time bounds for the Hume coordination layer.

We are interested in properties of the form "if $tl_1 = (1,0,0)$ then $tl_2 = (0,0,1)$ within time *Bound*", that is where *Bound* represents an upper bound on the time usage. Let $T_{l1/l2/ctl}$ be the time bounds guaranteed from the analysis of the expression layer. Further, let $T_{con}$ and $T_{write}$ be respectively the upper bounds on the time it takes to consume

and write all values. *Error* indicates that *Bound* has been exceeded and *Disabled* indicates that we are not between $tl_1 = (1,0,0)$ and $tl_2 = (0,0,1)$. Let $t$ be a variable representing time, and $\mathcal{N}^2$ be the conjunction of all next-actions. We can then define the real-time specification:

$$Prog4 \triangleq \exists\, \overline{i} : Init_{tl_1,tl_2} \wedge Init \wedge t = Disabled \wedge \Box[\mathcal{N}^2 \wedge t' = NextTime(t)]_{(\overline{i},\overline{w},tl_1,tl_2,t)}$$

where $t$ is initially *Disabled*. For each step *NextTime(t)* calculates the new value of $t$ as follows: if $tl'_1 = (1,0,0)$ then $t$ is set to *Bound*. For all the following steps $t$ is decremented with either $T_{l1} + T_{l2} + T_{ctl} + T_{con}$, if boxes are executed sequentially; or $Max(T_{l1}, T_{l2}, T_{ctl}) + T_{con} + T_{coord}$, if execution is concurrent. $T_{coord}$ is the coordination cost, if applicable. If $tl'_2 = (0,0,1)$ then $t$ is reset to *Disabled*. Finally, if $t \leq 0$ then $t'$ is set to *Error*. Since we want to verify that our specified time bound is never exceeded, we must prove the property:

$$Prog4 \Rightarrow \Box(t \neq Error)$$

We use *explicit-time* model checking [30] to verify this property. This obviates the use of a special real-time logic or model checker, and may not be much less efficient than such a checker in practice [30]. In our experiment we used the value $T_{l1} = 2, T_{l2} = 2, T_{ctl} = 1, T_{con} = 1, T_{write} = 1$. When executing the boxes sequentially, we found we were unable to guarantee a *Bound* of 30 or 50, but were able to guarantee a *Bound* of 60. In general, concrete time values, such as $T_{l1}, T_{l2}$ and $T_{ctl}$ above, can be obtained by using a worst-case execution time analysis on the expression layer; while $T_{con}$, $T_{write}$ and, if present, $T_{coord}$ will be platform-dependent, but should be easy to determine. A companion paper [7] shows how this could be done, giving concrete values for a simple architecture, and describes the construction of a worst-case execution time analysis for Full-Hume. Note that these values are not fixed in the TLA$^+$ specification, but can be supplied to TLC as part of the configuration of the model that must be checked.

## 4 A Software Implementation of HW-Hume

This section describes a high-performance software implementation of HW-Hume that can be used as the basis for software/hardware codesigns (where some HW-Hume boxes are implemented as described and others are replaced by hardware equivalents). The implementation also serves as a low-memory, high-performance implementation of Hume, where the source program is either restricted to the HW-Hume level, or can be transformed from a higher level of Hume into HW-Hume, for example as shown in [21]. We discuss hardware/software integration at the end of the section.

### 4.1 HW-Hume Abstract Machine Instructions

HW-Hume programs are compiled to a simple abstract machine which has a single accumulator plus some temporary memory locations, and which is designed to be easily implementable using simple logical operations. Each box is compiled independently, with each rule compiled into a sequence of abstract machine instructions. For

the pattern-part the abstract machine first determines the availability of the required inputs, then if sufficient inputs are available, matches these inputs against the patterns, and finally consumes the inputs; and for the expression-part, it constructs each non-ignored output by selecting any necessary parts of the inputs (so binding variables) and combining these with any required literal values before writing the result to one of the output wires. Finally, rules may be reordered according to fairness criteria, and control then returned to the scheduler.

**Expression-Level Instructions:** There are two main instructions. **Load** *lit* loads literal *lit* into the accumulator. **Select** *i pos size shift* loads *size* bits into the accumulator from input *i* starting at bit *pos*, offsetting these in the accumulator by *shift* bits. So:

```
Load 4
Select 1 5 2 0
```

will load three bits into the accumulator, where the top bit is the constant 1 (specified by **Load 4**), and the first and second bits are selected from the fifth and sixth bits of input number 1, respectively (specified by the **Select** instruction). The result can then be written to the appropriate output wire using a **Write** instruction.

**Pattern-Matching Instructions:** The **Match** *fail i nlits lits nvars vars* instruction matches input *i* against literal pattern *lits* (whose size in bits is specified by *nlits*), disregarding any input positions that will be bound to variables according to *vars* (whose size in bits is specified by *nvars*). If the input doesn't match, execution continues at label *fail*, usually corresponding to the next rule.

```
Match next 2 3 5 3 2
```

requires the first and third bits of input 2 to be constant ones (as specified by the literal 5), but accepts any value for the second bit (as specified by the value 2 for *vars*). Both literals and variables are three bits wide. Each set of **Match** instructions is preceded by a **CanConsume** instruction which determines whether the necessary inputs are available, and followed by a **Consume** instruction which unlatches the corresponding input. For example,

```
CanConsume next 4 6
Consume 4 6
```

checks whether the second and third input can be consumed (specified by the bit pattern 6), consuming them if so, and otherwise branching to the label `next`. For example, we can compile the simple selector box below:

```
box sel in (s :: Bit, x1, x2 :: Byte) out (y :: Byte)
match
   (0,x1,_) -> x1
|  (1,_,x2) -> x2;
```

into the following sequence of instructions (which have beeen wrapped in a pair of **Box**/**EndBox** pseudo-instructions):

```
Box "sel" "sel" 3 1 2 "sel_init"
```

```
Label "sel"
CanConsume "sel_1" 3 7
Match "sel_1" 0 1 0 1 0
Match "sel_1" 1 8 255 8 255
Match "sel_1" 2 8 255 8 255
Consume 3 7
Load 0
Select 1 0 8 0
Write 0
Schedule
```

```
Label "sel_1"
CanConsume "sel_2" 3 7
Match "sel_2" 0 1 1 1 0
Match "sel_2" 1 8 255 8 255
Match "sel_2" 2 8 255 8 255
Consume 3 7
Load 0
Select 2 0 8 0
Write 0
Schedule
```

```
Label "sel_2"
EndBox "sel"
```

### 4.2 Compilation

We have produced a template-compiler that translates each abstract machine instruction into portable C source code. This can then be compiled to give a native implementation of HW-Hume. Each box is compiled as a `void` C function. We also define an associated set of output wire buffers and a set of pointers that define the box's inputs. Boxes are placed in a scheduler queue and scheduled using a simple round-robin scheduler, where `qrem` simply removes and returns the next function from the queue if there is one, or else returns `NULL`. Boxes are added to the scheduler queue when they have sufficient inputs to be able to execute. When no boxes can execute, termination occurs.

```
void runHume () { while((next=qrem()) != NULL) (void) (*next)(); }
```

The `main` function adds the `_initial` function to the scheduling queue. This ensures that wires are properly initialised with any required values. It then adds the `checkavails` function, which will check input availability for each box and add it to the scheduler queue. Finally the `main` function enters the scheduler `runHume`, shown above.

Individual abstract machine instructions are defined as C macros, with Hume Abstract Machine (HAM) labels translated directly into C labels that can be branched to using a `goto`. For the **Match** instruction, we define the macro shown below. By xoring the input against the literal pattern, we will obtain a value which is 1 for each bit where the pattern matches the input and 0 otherwise. We then complete the match by setting each bit that is matched by a variable to 1. In this way, wherever the pattern matches, we will obtain a 1, and wherever it does not match, we will obtain a 0. We then check this against a mask that is all 1s for the number of input bits, branching to the `fail` label if unsuccessful.

```
#define Match(fail,input,nlits,lits,nvars,vars) \
      { unsigned match = ~(~thisbox->inp[input] ^ ~lits) | vars; \
        const unsigned mask = (1<<nlits)-1; \
        if((match & mask) != mask) goto fail; \
      }
```

Finally, the **Load** instruction simply loads the literal value into the accumulator and the **Select** instruction is used to select the appropriate bits from the required input position.

```
#define Load(val) { accum = val; }

#define Select(input,posn,size,shift)\
      { accum |= ((thisbox->inp[input] & ((1<<(posn+size))-1)) >> posn) << shift; }
```

### 4.3 Performance Results

Table 1 shows performance results for a number of HW-Hume programs running under three different implementations: $t_{hami}$ gives execution times for the prototype Hume Abstract Machine [20], a bytecode interpreter written in portable C; $t_{humec}$ gives corresponding times under the general Hume to C template-compiler we are constructing as part of the EmBounded project; and $t_{HW}$ gives times under the HW-Hume implementation we have described here. Space usage is given for the HAM interpreter, $s_{hami}$ and for the implementation described here, $s_{HW}$. Figures in brackets are those predicted by the cost model. All timings were obtained on a 1.67GHz Apple Powerbook G4 running MacOSX 10.4.8 and represent the average of 10 executions. Timings were recorded from box start to box end, and all C compilation was performed using gcc 4.0.0 using `-O2` optimisation. Our results show that, for these examples, the template compiler is slightly more than ten times faster than the bytecode compiler, and that the HW-Hume implementation is between 2.8 and 10.9 times faster than the latter implementation. While dynamic memory usage is low for the HAM interpreter at between 130B and 740B, it represents only a few words of memory for the HW-Hume implementation, being between 11 and 60 bits. Binary program size is also acceptably small. On an Intel Pentium IV running Linux, the total binary size for the HW-Hume multiplexor program, including all static and dynamic data and program code is 3526 bytes.

**Table 1.** Performance Comparisons

| **Program** | $t_{hami}$ | $t_{humec}$ | $t_{HW}$ | $s_{hami}$ | $s_{HW}$ |
|---|---|---|---|---|---|
| adder | 442$\mu$s | – | 7.0$\mu$s | 130B (130B) | 17b |
| multiplexer | 149$\mu$s | 12.9$\mu$s | 4.62$\mu$s | 732B (740B) | 60b |
| multiplexer2 | 275$\mu$s | 24.4$\mu$s | 5.25$\mu$s | 660B (664B) | 56b |
| lights | 286$\mu$s | 21.5$\mu$s | 1.96$\mu$s | 136B (240B) | 11b |

## 5 Hardware Implementation from HW-Hume

A hardware implementation can be obtained from HW-Hume in one of two main ways. Firstly, *netlists* can be generated directly from the description of Hume boxes and wires. Netlists, such as the widely-used EDIF [1], describe a collection of hardware devices, in terms of instances of master definitions, plus the interconnections between those devices, in terms of the *ports* associated with each device. It is then necessary to refine these netlists to include timing, placement and detailed functional behaviour, so that a hardware implementation can be obtained. Although substantial manual intervention may be required in later stages, there is considerable flexibility over the form of the

final hardware implementation. In HW-Hume terms, a *template* is a master definition, a *box* is an instance, box inputs/outputs are ports, and *wires* define interconnections. Alternatively, the C we have produced from our software implementation above could be passed as input to Handel-C [9] or a similar FPGA notation. This will then generate netlists and other required information so that an FPGA implementation can be produced. An example EDIF netlist for the half-adder above might be:

```
(edif halfadder

 -- version info
(edifVersion 2 0 0) (edifLevel 0) (keywordMap (keywordLevel 0))

-- this library
(library humeprogram
  (edifLevel 0) (technology (numberDefinition )           -- preamble
     (simulationInfo (logicValue H) (logicValue L)))

  (cell (rename HALFADDER "halfadder")(cellType GENERIC)  -- half-adder
    (view COMPASS_mde_view (viewType NETLIST)             -- netlist
      (interface
         (port a (direction INPUT))                       -- in/out ports
         (port b (direction INPUT))
         (port s (direction OUTPUT))
         (port c (direction OUTPUT)))))


   -- export the design
  (design HALFADDER (cellRef HALFADDER (libraryRef humeprogram)))))
```

It is also necessary to construct any required instances of HALFADDER and link these into a coherent network.

### 5.1 Hardware/Software Integration Issues

Hardware components can be integrated into HW-Hume software programs either by completely replacing some box, where they are equivalent to the HW-Hume source, or as unspecified "pseudo-boxes". In either case, it is necessary to provide linkages between software and hardware so that such boxes will react to (possibly software) inputs and produce outputs that can be directed to software boxes. For example

```
operation "count1" to "74HC393/1"  :: vector 2 of Bit -> vector 4 of Bit;
operation "count2" to "74HC393/2"  :: vector 2 of Bit -> vector 4 of Bit;
```

might specify two pseudo-boxes `count1` and `count2`, one attached to each half of a 74HC393 four-bit binary counter. The two one-bit inputs are a clock signal and a master reset input in each case. These boxes can be connected to software in the usual way. Note that in this case, an explicit clock signal must be threaded as an additional input to each HW-Hume box where it is required.

## 6 Related Work

Declarative hardware description languages are an attractive approach, allowing clean separation of functionality from behavioural detail, supporting automatic circuit generation, and promoting much higher level of abstraction than found in the industry-standard VHDL notation, for example. One early declarative approach, Ruby [27], was based on relational calculus. While there is an obvious link between logic gates and logical relations, in practice, most hardware circuits map some inputs to some outputs. It follows that functional approaches to hardware description are not only possible, but also completely appropriate, and several examples have been described previously.

There have been several approaches to developing functionally-based notations for hardware. Lava [6,11], produced in association with Xilinx Corporation, uses an embedded domain-specific language approach, extending Haskell with operations that allow the high-level description of FPGA circuits. Where Lava uses non-strictness to specify links between hardware components, in Hume, boxes/wires serve the same purpose. Other similar approaches include Intel's ReFL$^{ect}$ language [19], which is used commercially to verify properties of their processor designs; the Hawk hardware verification language [26]; the Hydra system for logic circuit specification; the *functional derivation* approach, for deriving FPGA circuits from Haskell specifications [25]; the lenient, purely functional language Confluence for designing synchronous circuits [2], the imperative HDCaml hardware design/verification language [3]; the SAFL hardware description language [31]; and the same authors' Flash notation for hardware/software codesign [32]. Compared with HW-Hume, the most obvious differences in these notations are their use of a single-level language rather than a separation between coordination and expression, their inclusion of high-level features such as higher-order functions and direct recursion (though these may be mapped from higher levels of Hume into HW-Hume programs), and the general absence of asynchronous constructs. The decision to include asynchronous constructs in Hume is a careful one. The advantage of a synchronous language design such as Lustre [10] is in terms of a simpler semantic model, that consequently simplifies the construction of cost models. However, while asynchronous systems can generally be restricted to synchronous cases, and this can be detected using model-checking as we have done in this paper, it is considerably more difficult to describe asynchronous systems starting from a purely synchronous basis. Recent work has therefore seen hybrid notations, such as Lucid-Synchrone [14], which combines finite-state-automata and a synchronous communication model, or notations that explicitly expose clocks as additional inputs to otherwise synchronous systems [13].

While model checking has been successfully applied to several imperative languages, for example in the shape of NASA's Java Pathfinder [24] or Microsoft's Terminator [15] tools, there are fewer systems combining functional languages with model checking. Apart from our own work on HW-Hume and Spin [18], the most relevant work of which we are aware is that on ReFL$^{ect}$ [19], on verifying SAFL programs [16], and on verifying resource properties in Erlang [33]. A key difference from our work is that we deal with real-time properties as well as liveness and safety. Since we have constructed a formal model of the Hume coordination layer, which is identical to all Hume levels, we are also able, in principle, to work at arbitrary levels of Hume and to prove properties on transformed code.

## 7 Conclusions and Future Work

This paper represents a first exploration of HW-Hume. HW-Hume targets low-level system descriptions, using a declarative notation combining purely functional expressions with a high-level process notation. We have shown how essential safety, liveness and real-time properties of HW-Hume programs can be specified in TLA$^+$ and how they can automatically verified with the TLC model checker. In doing this, we have provided the first example of using TLA$^+$ to model check properties in a programming language. The combination of time analysis on boxes with temporal logic is also novel, and reveals the advantage of using a layered language when performing static analysis, allowing clear separation between different aspects of the time analysis.

Since TLA$^+$ is a much higher level notation than supported by most model checkers, this allows a more direct embedding of HW-Hume semantics, and also helps mitigate the "state-space explosion problem", a major bugbear of model checking, where the checker fails because too many states have been generated. Even more interestingly, we have been able to extend the work reported here to model-check that the safety-part of a property is preserved when transforming from a higher-level into a lower-level Hume program.

We have also shown how an efficient software implementation can be produced for HW-Hume, using a template-based compiler compiling through C. This implementation is highly space efficient. For example, for the `sel` box above, we can determine a total dynamic memory usage of 42 bits (including all wiring requirements), and the complete C program in which it is embedded has a total dynamic memory requirement of 620 bytes, including all system data structures and runtime queues. HW-Hume may therefore be the world's most space-efficient functional language.

### 7.1 Further Work

In addition to producing a concrete hardware implementation for HW-Hume, as discussed above, a number of important issues remain to be addressed. Firstly, hardware/software co-design is becoming increasingly important as an approach to building embedded computer systems. As we have shown above, it is possible to produce both hardware and software implementations from a single HW-Hume definition. We believe this gives a powerful tool for developing combined hardware/software implementations from a single source specification, and intend to investigate this further. Secondly, we have already developed a powerful transformational framework allowing higher levels of Hume to be mapped into HW-Hume programs. In this way, programmers have access to higher-order combinators, repetition and other abstractions. We need to investigate whether this approach gives an effective way to provide high-level abstractions over hardware circuits. Thirdly although TLA$^+$ has proved effective for HW-Hume, when dealing with more expressive levels of Hume, it is likely to prove insufficiently powerful, since we will need to deal with more sophisticated forms of data structures, for example. We are therefore working on formalising TLA in a theorem prover. Fourthly, TLC supports a form of state-space reduction technique called symmetry which may yield further performance benefits. We intend to address how we may exploit this in HW-Hume. Fiftly, we have developed a specification language for HW-Hume, based

on [18], which captures all properties we have shown. We plan to create a translator from HW-Hume and this specification language into TLA$^+$ which automatically verifies the properties. We believe this should be a trivial thing to do. Finally, although we have defined box templates and wiring macros to reduce repetition in describing collections of boxes, we have not developed a complete hierarchy of box-combining forms. This would effectively involve constructing a higher-order calculus of boxes, and would allow more modular and scalable verification of properties.

## References

1. Electronic Design Interchange Format Version 2.0.0,Technical ANSI/EIA-548-1988 (1988)
2. Confluence: `http://www.confluent.org/wiki/doku.php?id=confluence` (2006)
3. Hdcaml: `http://www.confluent.org/wiki/doku.php` (2006)
4. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21, 181–185 (1985)
5. Apt, K.R., Olderog, E.-R.: Verification of Sequential and Concurrent Programs, 2nd edn. Springer, Heidelberg (1997)
6. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in Haskell. ACM SIGPLAN Notices 34(1), 174–184 (January 1999)
7. Bonenfant, A., Ferdinand, C., Hammond, K., Heckmann, R.: Worst-Case Execution Times for a Purely Functional Language. In: this proceedings, Springer, Heidelberg (2007)
8. Boussinot, F., de Simone, R.: The Esterel Language. Proceedings of the IEEE 79(9), 1293–1304 (September 1991)
9. Butterfield, A., Woodcock, J.: prialt in Handel-C: an operational semantics. Int. J. Softw. Tools Technol. Transf. 7(3), 248–267 (2005)
10. Caspi, P., Pilaud, D., Halbwachs, N., Place, J.: Lustre: a Declarative Language for Programming Synchronous Systems. In: Proc. POPL '87 – 1987 Symposium on Principles of Programming Languages, München, Germany, pp. 178–188 (January 1987)
11. Claessen, K., Pace, G.: An Embedded Language Framework for Hardware Compilation. In: Proc. Conf. on Designing Correct Circuits (DCC 2002) (2002)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
13. Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: N-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In: Proc. POPL '06: ACM Symposium on Principles of Programming Languages, pp. 180–193. ACM Press, New York (2006)
14. Colaço, J.-L., Pagano, B., Pouzet, M.: A Conservative Extension of Synchronous Data-flow with State Machines. In: Proc. ACM International Conference on Embedded Software (EMSOFT'05), Jersey City, New Jersey, USA (September 2005)
15. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond Safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
16. Foster, J.N.: Model Checking for a Functional Hardware Description Language, BSc Dissertation, Cambridge University. PhD thesis (2002)
17. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: A Declarative Language For Synchronous Programming of Real-Time Systems. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 274, pp. 257–277. Springer, Heidelberg (1987)
18. Grov, G., Ireland, A., Michaelson, G.J., Hammond, K.: Verifying Temporal Properties in HW-Hume. Technical report, Heriot-Watt University, School of Mathematical and Computer Sciences (February 2006)

19. Grundy, J., Melham, T., O'Leary, J.: A Reflective Functional Language for Hardware Design and Theorem Proving. J. Funct. Program 16(2), 157–196 (2006)
20. Hammond, K.: Exploiting Purely Functional Programming to Obtain Bounded Resource Behaviour: the Hume Approach. In: Central European Summer School on Functional Programming, July 2005, Springer, Heidelberg (2006)
21. Hammond, K., Michaelson, G.: Bounded Space Programming using Finite State Machines and Recursive Functions: the Hume Approach. Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM), in preparation(2006)
22. Hammond, K., Michaelson, G.J.: Hume: a Domain-Specific Language for Real-Time Embedded Systems. In: Pfenning, F., Smaragdakis, Y. (eds.) GPCE 2003. LNCS, vol. 2830, pp. 37–56. Springer, Heidelberg (2003)
23. Hammond, K., Michaelson, G.J.: Predictable Space Behaviour in FSM-Hume. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, Springer, Heidelberg (2003)
24. Havelund, K., Pressburger, T.: Model Checking JAVA Programs using JAVA PathFinder. Int. Journal on Software Tools for Technology Transfer 2(4), 366–381 (2000)
25. Hawkins, J., Abdallah, A.E.: Behavioural Synthesis of a Parallel Hardware JPEG Decoder from a Functional Specification. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 615–619. Springer, Heidelberg (August 2002)
26. Launchbury, J., Matthews, J., Cook, B.: Microprocessor Specification in Hawk. In: Proc. International Conference on Computer Languages, pp. 90–101 (1998)
27. Jones, G., Sheeran, M.: Circuit design in Ruby. In: J. Staunstrup, editor, Formal Methods for VLSI Design, pp. 13–70. North-Holland (1990)
28. Lamport, L.: The Temporal Logic of Actions. ACM TOPLAS 16(3), 872–923 (1994)
29. Lamport, L.: Specifying Systems — The TLA+ Language and Tools for Hardware and Software Engineers, Reading, Massachusetts. Addison-Wesley, London (2002)
30. Lamport, L.: Real-Time Model Checking Is Really Simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
31. Mycroft, A., Sharp, R.: A Statically Allocated Parallel Functional Language. Automata, Languages and Programming, pp. 37–48 (2000)
32. Mycroft, A., Sharp, R.: Hardware/Software Co-Design Using Functional Languages. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 236–251. Springer, Heidelberg (2001)
33. Earle, C.B., Arts, T., Derrick, J.: Verifying Erlang Code: a Resource Locker Case-Study. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 184–203. Springer, Heidelberg (2002)
34. Vasconcelos, P.B.: Cost Inference and Analysis for Recursive Functional Programs. PhD thesis, University of St Andrews, 2006. in preparation
35. Vasconcelos, P.B., Hammond, K.: Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 86–101. Springer, Heidelberg (2004)