Further Sub-cycle and Multi-cycle Schedulling support for Bluespec Verilog

David J. Greaves

ABSTRACT

Bluespec [12] is a hardware description language where all behaviour is expressed in rules that execute atomically. The standard compilation semantics for Bluespec enforce a particular mapping between rule firing and hardware clock cycles, such as a register only being updated by exactly one firing of at most one rule in any clock cycle. Also, the standard compiler does not introduce any additional state, such as credit-based or round-robin arbiters to guarantee fairness between rules over time. On the other hand, many useful hardware resources, such as complex ALUs and synchronous RAMs, are pipelined. Unlike typical high-level synthesis tools, in standard Bluespec such resources cannot be invoked using infix operators in expressions such as A[e] or e1*e2 since binding to specific instances and multi-clock cycle schedules are required. In this paper we extend the reference semantics of Bluespec to decouple it from clock cycles, allowing multiple updates to a register within one clock cycle and automatic instantiation of arbiters for multi-clock cycle behaviour. We describe the new semantic packing rules as extensions of our standard compilation rules and we report early results from an open-source, fully-functional implementation.

1 INTRODUCTION

Bluespec [12] is a programming language for generating hardware circuits. The Bluespec language was created at MIT and is now promoted by Bluespec Inc.. The compiler from that company is only available under license.

Although there is no accepted taxonomy of high versus low-level languages for hardware design, we can roughly relate a gate-level netlist to machine code, RTL to assembly language, hardware construction languages such as Chisel[1] and Lava[2] as low-level languages and anything that makes automatic assignment of work to clock cycles as high-level languages. Accordingly, Bluespec can be classed as a high-level language. However, it arguably sits at a lower level than traditional HLS (high-level synthesis) since Bluespec does not make heuristic-guided searches for optimal binding of operations to FUs (functional units such as ALUs and RAMs) or multi-cycle static schedules. Programs in a 'Hardware Construction Language', such as Chisel, essentially 'print out' an RTL or structural design. This process is called structural elaboration. HardCaml from Jane Street and CLaSH[15] are further examples. The generate statements of Verilog and VHDL form the hardware construction languages of those RTLs. Bluespec embodies a sophisticated hardware construction language based on functional programming combinators. The structural elaboration may contain loops and other control flow constructs, but the elaboration is performed entirely at compile time. Hence none of the conditional statements processed in the hardware construction language depends on any run-time data. There is no data-dependent control flow in the elaboration language.

```
module mkTb1 (Empty);

   Reg#(int) x <- mkReg (23);

   rule countup (x < 35);
      int y = x + 1;       // This is short for  int y = x._read() + 1;
      x <= x + 1;          // This is short for  x._write(x._read() + 1);
      $display ("x = %0d, y = %0d", x, y);
   endrule

   rule done (x >= 30);
      $finish (0);
   endrule

endmodule: mkTb1
```

Figure 1: A short, flat Bluespec program with two rules sharing one register.

Bluespec is based around the concept of modules and rules. A module contains zero or more rules. A module also instantiates zero or more lower modules. Modules instantiated at the lowest levels are primitives, such as FIFOs, registers and RAMs. Bluespec starts structural elaboration at a top-level module. The module hierarchy is nominally flattened during the structural elaboration process. Once elaboration is complete, we have essentially a flat collection of interconnected Bluespec rules and primitives.

The standard compilation semantics for Bluespec enforce a particular mapping between rule firing and hardware clock cycles, such as a register only being updated by exactly one firing of at most one rule in any clock cycle.

Fig. 1 presents a small example with two rules: one called countup increments, the other, called done, exits the simulation.

A module may also export methods that can be invoked by other modules. These are normally re-entrant, being elaborated freshly for each rule that invokes them. But where the design hierarchy is partitioned into separate compilation units, which can be done with compiler directives or annotations embedded in the source code, a set of terminals is created in the module signature for each callable method. There is then a variation in semantic in that the method can be called at most once per clock cycle.

In this paper, we extend Bluespec by: 1) allowing multiple updates to registers within one clock cycle, 2) allowing rules to be fired more than once per clock cycle, 3) supporting automatic fair scheduling of rules and 4) register forwarding for multi-clock cycle operators.

2 FORMALISM

There are several available descriptions of Bluespec reference semantics. The Kami Bluespec project has published reference semantics of a Bluespec subset in Coq [3]. We used the Bluespec Reference Guide version 3.8 dated April 2004 and and revision, 16 dated June 2010. These were both found online and there is no discernible change in the core semantics between these editions.

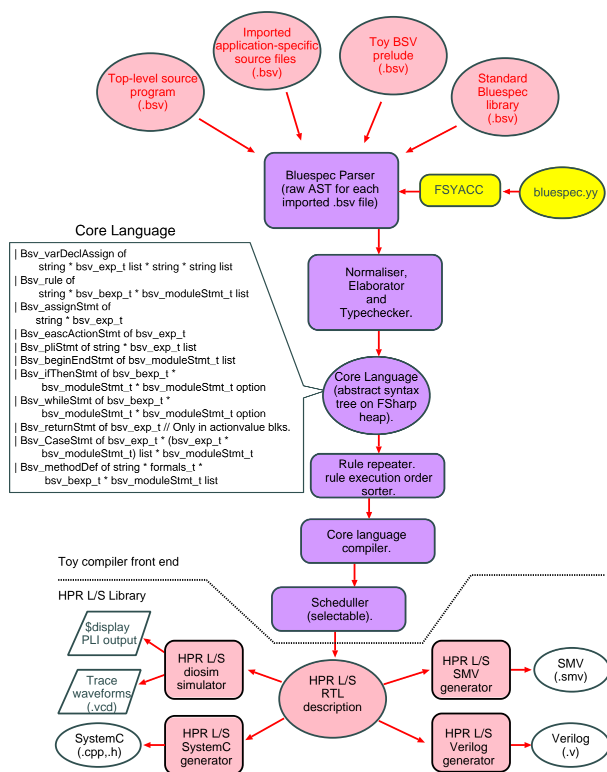

Figure 2: Overall structure of the Open-Source Compiler.

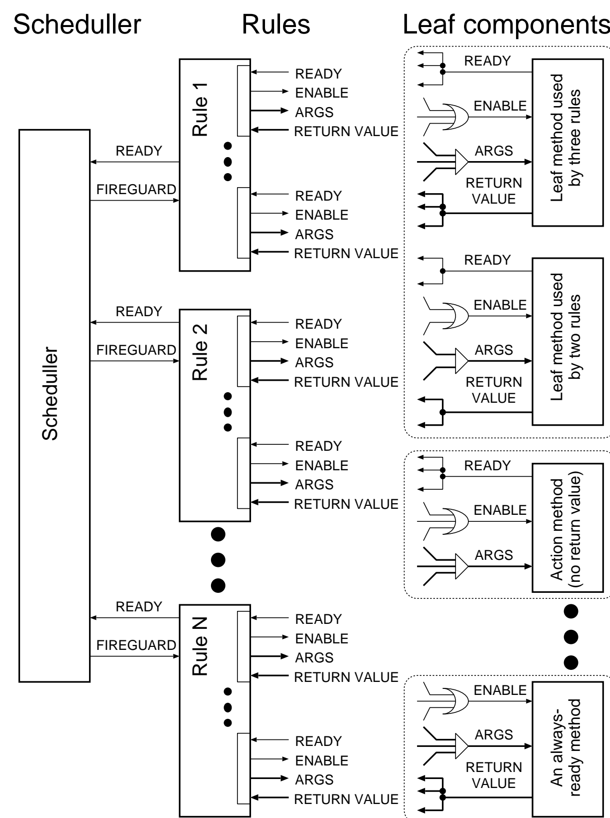

Figure 3: General view of the Bluespec wiring between scheduller, rule logic and leaf components. Each rule connects to some number of leaf components. Dotted outlines show example component boundaries.

In this section we present the compiler operation and code structure using a formalism that is identical in behaviour to an open-source implementation of Bluespec known as the Toy compiler [9]. This compiler does not have the maturity or all of the features (such as support for harmonic clocks) of the commercial offering, but can compile a fair number of publicly available test programs and was sufficient for proof of concept of our experimental extensions. We believe the commercial Bluespec compiler is coded in Haskell whereas the toy compiler (Fig. 2) is coded in FSharp using mainly the OCaml subset.

Fig. 3 presents a general sketch of the overall setup. We believe roughly the same approach is used in the commercial implementation, as described in [5, 14]. In the next section we will explain the implementation of our extensions with respect to our formalism.

After elaboration of the generating functions we have the on-heap representation of the current compilation unit. Note that module boundaries are not shown since they have disappeared during elaboration. Similarly, which interface or component a method is part of has also largely disappeared (see §2.1 where the component is relevant for re-entrance management). There are five main forms on the heap:

Rule definitions	$R_i(g, [A_{i,0}, A_{i1} \ldots])$	y
Exported methods	$XM_i(CM_{i,0}, CM_{i,1} \ldots])$	-
External method references	$EM_i([r_i, a_{i,0}, a_{i,1} \ldots])$	-
Leaf component methods	$CM_{i,0}, CM_{i,1} \ldots$	y
Local method definitions	$LM_i()$ ...	-
Pragmas/Directives	...	-

Currency Field	Symbol	Dir	Initial Value
Method unique name	CM_i	-	iname.mname
Enable net	en_i	in	0
Ready expression	rdy_i	out	"CM_i_rdy"
Argument list expressions	$arg_{i,j}$	in	[x,x,...]
Result bus name option	rv_i	out	"RV_i_en"

Table 1: Fields present for each method in the currency.

A 'y' in the third column denotes that the form is illustrated in Fig. 3. The exported methods are not shown since they can be converted to rules. The external method references are not shown since they are no different from the leaf/imported methods. The main compilation step is essentially to convert the abstract syntax trees inside each rule into wiring that connects up the leaf methods and then to generate a custom scheduller.

After elaboration, every method has a unique textual name. It is represented by a tuple that has four further fields that are RTL expressions. We call this tuple the *currency* of the method, denoted Σ. In this table, the direction of the connection on the method implementation is shown, but this is reversed on a method reference to enable one-to-one wiring.

Some methods take no arguments or return no results, so the resulting fields are empty. Some methods are always_ready or always_enabled which respectively means that the ready expression need not be represented since it always holds and that both ready and enable need not be represented since they both always hold. Fig.4 shows an interface definition and its hardware form.

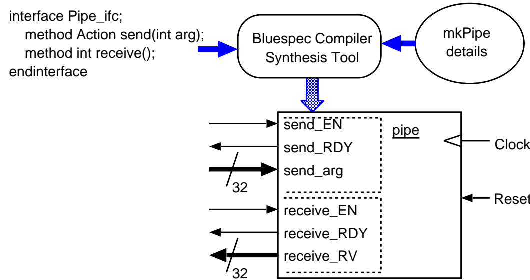

Figure 4: Typical manifestation of a Bluespec component at the net level.

For an imported method, the name is given by a concatenation of its simple string name with the module instance names or formal name in the imported interface list. For a leaf method, the prefix is the component instance name. For an exported method, the prefix is a dummy. Where methods are part of named sub-interfaces, the interface instance names are included. Overall we have a hierarchic path name to a method that is henceforth opaque.

Our first compilation step is to create the initial currency for each callable method. This currency is generated by a tree walk of all instantiated components and external method references. The RTL expressions are initialised as shown in Tab. 1. The enable is logical false, the arguments are don't-care and the ready and result fields are the net names that are connected to the components. For leaf components we also need to generate the actual component implementations, but details of that are out of scope for this paper. The components are typically registers, FIFOs, RAMs or callouts to third-party RTL (or other output language such as SystemC).

The external method references are the flattened contents of the top-level module's interface. Each exported method *EM*() is an entry point invokable at the net level by rules in parent compilations or testbench or pad ring etc.. For all other methods, we are responsible for driving the enable input, but for an exported method, the enable signal will be an input to the current compilation. The opposite applies for the ready signal. Exported methods can be treated as rules in most respects. They are converted to rule form that has a phantom external method call, treating its arguments as results, result as arguments and swapping over ready and enable. This requires handling a method with multiple results, but this is trivial, since in RTL terms, this is just a wider result bus. A problem sometimes arises from the allowable causality between ready and enable that is orthogonal to the work presented here.[1]

[1]A combinational loop between the schedulers present at different levels of an incremental compilation will arise in certain circumstances. The commercial compiler does not support incremental compilation whereas the Toy compiler does not check the causality problem, so back end tools can sometimes flag a warning.

There is no difference in treatment required between external method references and child component methods. This is why they are not shown on Fig. 3. We need not mention external method references further.

The final structure present after elaboration is local method definitions. These have the same type and form as exported methods but they are not listed in the exported interface. Instead, they are only called by the local rules and, as mentioned earlier, they are elaborated on-the-fly during rule compilation.

Each rule, R_i, has a name, an explicit guard, g and the action from its body. The explicit guard is a user-provided Boolean expression (such as $\text{x}<35$ in the 'countup' rule). The rule body is an unordered list of actions to be performed atomically. Actions are mainly method invocations in the currency, but there can also be a few invocations of built-in primitives, such as diagnostics, so-called *wires*, such as PulseWire, and PLI calls, none of which we shall explain herein. Register assignments syntax, such as $\text{myreg} <= \text{exp};$ has already been de-sugared to myreg._write(exp) and register reads have likewise been converted to myreg._read().

The set of rules is first put in a linear order called the execution order. This is sorted according to the textual order encountered during elaboration that has been conservatively sorted to respect any execution_order partial ordering directives from the user[2]. Where the superscalar extension is used, §3.3, rules may be replicated in the list. Rules are named by the user when defined and their full name becomes the instance name of their module suffixed with that user name (and then suffixed with a natural number under superscalar issue). Each rule name is extended with two suffixes to give the names of the two nets that connect it to the scheduller. These nets are called ready and fireguard.

Rule compilation is then performed on each rule in turn in the chosen order. Each rule compilation results in an expression to be continuously assigned to the rule's ready net and also in rewrites of the method currency. The currency rewrites add further disjuncts to the enable field and additional inputs to multiplexors for the argument busses.

Fig. 5 gives the rule compilation procedure in denotational style. The FSharp implementation is identical. Every expression has a so-called intrinsic-guard which is the conjunction (ANDing) of the intrinsic guards of its sub-expressions. The enable for a command is denoted with α and the intrinsic guard is denoted with ρ. The initial enable expression for a rule is its fireguard which is its 'go' signal from the scheduller.

The evaluator function for a command varies from the evaluator function for expressions in that it does not return a value. Both evaluator functions can augment the currency, Σ and add new conjuncts into ρ.

Actions may be conditional within a rule body, guarded with control flow statements such as 'IF' and non-strict operators such as '? :'. Control flow is handled by guarding the enable expression with the control flow predicate. At entry to a conditional control

[2]Sorted also to obey certain causality requirements from additional annotations, such as on the so-called *wires* which must be written before they are read to send combinational data between rules.

$$[\![\ \text{const_expression}\ ]\!]_{\alpha,\Sigma,\sigma,\rho} = (\text{const_expression}, \Sigma, \rho)\ \text{(constant expression)}$$

$$[\![\ \text{v}\ ]\!]_{\alpha,\Sigma,\sigma,\rho} = (\sigma(v), \Sigma, \rho)\ \text{(variable r − mode)}$$

$$[\![\ \text{e1 op e2}\ ]\!]_{\alpha,\Sigma,\sigma,\rho} = \text{let}\ (v_1, \Sigma', \rho') = [\![\ \text{e1}\ ]\!]_{\alpha,\Sigma,\sigma,\rho}\ \text{and}\ (v_2, \Sigma'', \rho'') = [\![\ e2\ ]\!]_{\alpha,\Sigma',\sigma,\rho'}\ \text{in}\ (v_1\ \text{op}\ v_2, \Sigma'', \rho'')\ \text{(primop/function)}$$

$$[\![\ \text{[e1; e2...]}\ ]\!]_{\alpha,\Sigma,\sigma,\rho} = \text{let}\ (v1, \Sigma', \rho') = [\![\ \text{e1}\ ]\!]_{\alpha,\Sigma,\sigma,\rho}\ \text{let}\ (v2, \alpha, \Sigma', \rho') = [\![\ \text{e2}\ ]\!]_{\alpha,\Sigma',\sigma,\rho'}\ \ldots\ \text{in}\ ([v1, v2, \ldots], \Sigma^*, \rho^*)\ \text{(expression list)}$$

$$\begin{aligned}[\![\ \text{method_name(e_list)}\ ]\!]_{\alpha,\Sigma,\sigma,\rho} \quad = \quad & \text{let}\ (\text{e_list}', \Sigma', \sigma, \rho') = [\![\ \text{e_list}\ ]\!]_{\alpha,\Sigma,\sigma,\rho}\ \text{(leaf method app)} \\ & \text{let (en, rdy, args, rv)} = \Sigma(\text{method_name}) \\ & \text{let args}' = (\alpha)\ ?\ \text{e_list}' :\ \text{args} \\ & \text{in}\ (\text{rv}, \Sigma'[(\alpha \vee \text{en, rdy, rv, args}')/\text{mname}], \rho' \wedge \text{rdy})\end{aligned}$$

$$\begin{aligned}[\![\ \text{method_name(arg)}\ ]\!]_{\alpha,\Sigma,\sigma,\rho} \quad = \quad & \text{let}\ (\text{actual}, \Sigma', \rho') = [\![\ \text{arg}\ ]\!]_{\alpha,\Sigma,\sigma,\rho}\ \text{(local method app)} \\ & \text{let (bound_var, body)} = \Sigma(\text{method_name}) \\ & \text{let}\ \sigma' = \sigma[\text{actual/bound_var}] \\ & \text{let}\ (\text{rv}, \Sigma'', \rho'') = [\![\ \text{body}\ ]\!]_{\alpha,\Sigma',\sigma',\rho} \\ & \text{in}\ (\text{rv}, \Sigma'', \rho'')\end{aligned}$$

$$[\![\ \text{c1; c2}\ ]\!]_{\alpha,\Sigma,\sigma,\rho} = \text{let}\ (\Sigma', \rho') = [\![\ \text{c1}\ ]\!]_{\alpha,\Sigma,\sigma,\rho}\ \text{in}\ [\![\ \text{c2}\ ]\!]_{\alpha,\Sigma',\sigma,\rho'}\ \text{(parallel composition)}$$

$$[\![\ \text{if}\ (g)\ \text{c1}\ ]\!]_{\alpha,\Sigma,\sigma,\rho} = \text{let}\ (g1, \Sigma', \rho') = [\![\ g\ ]\!]_{\alpha,\Sigma,\sigma,\rho}\ \text{in}\ [\![\ \text{c1}\ ]\!]_{\alpha \vee g1,\ \Sigma',\sigma,\rho'}\ \text{(control flow)}$$

Figure 5: Basic Compilation Rules for Expressions, Actions and Rules.

flow region, the enable (here known as the activation expression), α is ANDed with the control flow guard expression. [3]

For leaf method invocations, the activation expression needs to be connected to the enable input of the called method and the ready net of the called method is its intrinsic guard. Constant expressions are always ready and all expressions that are always_enabled require no fireguard. The rewrite to the currency when a method is invoked is to first evaluate the arguments in a call-by-value style, to OR-in the current fireguard, α, to its enable field, to mux in the actual args to the arg bus expressions, guarded by α and to return the value of its result bus when not void. The intrinsic guard of the method call is the conjunction of the intrinsic guards of all the arguments and the method's ready net. Hence, as method calls are compiled, their bodies are represented as augmented symbolic expressions stored in the currency. For each leaf method invocation, a write is made to a conflict log (not shown in the semantics) read when schedulling. The entry is the name of the rule, the name of the method, the memoising heap tag for each argument expression (§4) and the activation expression, α.

For local method invocations, the method body is looked up at compile time, the actual arguments are evaluated as for a leaf method call and the resulting expressions are bound in the environment, σ, as formal/actual pairs and finally the body is compiled. The local method definitions are held in Σ as a variant form and our presentation for only one argument is generalised in the real implementation. Note that everyday runtime variables do not exist: they have been elaborated into read and write TLM calls on the corresponding static instance.

Arithmetic operators and built-in functions are preserved symbolically or executed straightaway when applied to constant expressions.

The contents of all of the actions in a rule body are flattened and treated as a single un-ordered list. This is because for all common components, the required atomicity of rule firing is guaranteed by their hardware representation. For instance, RTL registers can be read and written in the same clock cycle, with the current value being returned and the next value queued as a pending update to be committed on the next clock edge in the RTL simulator or master section of a master-slave flip-flop in real hardware.

What would be the sequencing operator, in a normal imperative language, denoted with semicolon, is in fact parallel composition owing to the underlying RTL-like blocking assignment semantics.

After all rules are compiled, the argument and enable expressions can be written out as RTL continuous assignments to the argument busses and enable connections of the leaf components. As a result of the rule compilation, there is a multiplexor tree in the RTL for each leaf method argument that is driven from more than once place. The logic also contains all of the in-line arithmetic and logic operators found in the language (apart from the pipelined ones we mention later). The Toy compiler is built on the HPR L/S

[3] There are strict and non-strict versions of control flow supported by both compilers: The presentation here is of the strict form, but non-strict does not require the intrinsic guard of un-executed code to hold.

library that intrinsically implements constant propagation, subexpression sharing and logic minimisation using Espresso, so the generated RTL is not overly verbose.

The Toy compiler also implements other parts of the language that are not relevant here, such as mutable vectors that can elaborate to register files and the Bluespec FSM sub-language.

As a final step in compiling a rule, the explicit guard expression is compiled with the expression compiler. This returns both a condition and an intrinsic guard. These two are AND-ed together and then AND-ed with the ρ intrinsic guard of the rule body and returned as the ready condition for the rule as a whole.

2.1 Re-entrant Leaf Components

The only time that it matters which component a method is of is when there are re-entrancy restrictions. For instance, the implementation of a component may share resources between exported methods such that both cannot be invoked in the same clock cycle. Such conflicts are recorded as markup in the interface definition files for these components. They are copied into the conflict log as the component is instantiated where the prototype entries are extended with the full component instance name.

2.2 The Bluespec Scheduler

Each leaf component represents a structural hazard where the input multiplexor is non-trivial (ie. has one or more argument inputs). Rules that need to drive different expressions into the argument of a leaf component are said to conflict. Conflicts are computed by collating the conflict log entries. In the absence of firing rate targets (§3.6), it is the scheduler's job simply to prevent the firing of conflicting rules and to try to avoid rule starvation.

As mentioned, all method applications have a unique tag after flattening and each argument expression has a natural number heap index. Hence all potential operations on a leaf component have a unique composite name. They also have logged against them activation expressions within the rule (the α expression at the point where logged) as a side effect of rule compilation. A conflict candidate is now defined as a single application tag with differing heap indecies guarded by a rule name and activation expression. The intrinsic guard and explicit guard also need to be conjuncted with the activation expression to give the final might-be-invoked expression for a conflict candidate. A rule conflicts with another one if any of their conflict candidates differs in argument expressions and the intersection of the two might-be-invoked expressions is not false, as determined by the proof techniques embodied in the memoising heap.

As said earlier, where application tags apply to methods on components where those methods are flagged as non-reentrant, the application tags are considered to identify (hold true under equality) for this purpose.

Rules that do not conflict with anything can fire as often as they are ready and the output from the scheduler for such rules is a combinational assignment of the fireguard from the ready guard. We call this the rule tie off.

The schedulling problem can be phrased as a graph where the rules are nodes and arcs exist between rules that conflict. Each rule has a ready guard and conflicts are arcs between rules. Arcs with disjoint (cannot both hold) ready guards can be disregarded and this could be the basis of a schedulling algorithm. The approach used in the Esposito scheduler [5] is instead to convert rules such that they no longer conflict with anything and then tie them off. The principle transform is to merge a pair of conflicting rules into one. Where they had disjoint ready conditions, as can be seen from the compilation semantics, treating them as one or two rules makes no difference.

Where the ready conditions are not disjoint, one of them may imply the other. The approach then is to give static priority to the one that may be ready less than the other, unless overriden by decreasing_urgency mark up. If no such implication exists, a static priority given by textual position in the source files, overriden by decreasing_urgency directives is applied. The higher priority rule is schedulled first and its conflicts are removed from the remainder by adding its negated fireguard to the ready-to-fire condition of the remainder.

Where one rule conflicts with another, such that they cannot both be schedulled and static priority will lead to starvation of the lower priority one, a severe compile time error is issued (§3.1).

Schedulling decisions can also be guided to avoid large combinational delay in the control logic, but the Toy compiler does not do this.

3 EXTENSIONS

In this section we report on some novel feature extensions we have added to the open-source Toy compiler.

3.1 Automatic Insertion of Stateful Schedulers

Standard Bluespec semantics are that a static schedule is created that is executed afresh every clock cycle with no schedulling state carried from one clock cycle to the next. §2.2 explained that some rules may be starved of service under the static schedule. Manual instantiation of arbiters and other anti-hog mechanisms is one solution. Our extension is an automated system that inserts stateful schedulers guided by rule firing rate targets.

We present an example that uses two compilation units that sit each side of an interface and are separately compiled. A single compilation unit would allow both rules in the parent/master unit to fire at once, since the method in the interface will be elaborated separately for each call. But as separate units, we have a structural hazard: the method is manifested at the net level in the interface and can only be invoked once per clock cycle.[4]

```
//Interface definition:
interface BarFace;
  method Action orderDrink(int which, int no);
endinterface
//Lower unit definition:
module mkBarTender(BarFace);
    Reg#(Bit#(10)) beerdrink <- mkReg(20);
    Reg#(Bit#(10)) winedrink <- mkReg(10);

    method Action orderDrink(int which, int no);
```

[4]More typically, the lower unit would be a leaf component not implemented in Bluespec, since we were told the commercial compiler does not really support incremental compilation, but it is nicer to present both halves of the example in Bluespec.

```
      if (which == 1) beerdrink <= beerdrink + no;
      if (which == 2) winedrink <= winedrink + no;
   endmethod

   rule shower if (True);
     $display("Beer is %1d and wine is \
           %1d", beerdrink, winedrink);
   endrule
endmodule
```

```
//Upper unit definition:
module mkTest1iBench();
   BarFace fbar <- mkBarTender();

// Beer should increase by two every clock cycle.
   rule drinkBeer;
     fbar.orderDrink(1, 2);
   endrule
// drinkWine would normally be starved.
   rule drinkWine if (True);
     fbar.orderDrink(2, 10);
   endrule
endmodule
```

Compiling with the standard semantics we see a starvation warning from the compiler:

```
** Starvation detected: rule mkTest1iBench.drinkWine
** rules being greedy are mkTest1iBench.drinkBeer
```

And simulation demonstrates that wine is never consumed owing to beer hogging:

```
Beer is 22 and wine is 12
Beer is 24 and wine is 12
Beer is 26 and wine is 12
...
```

This demo is now compiled with our extension that instantiates run-time arbiters to provide fairness between the rules. The simulation listing now generated is

```
Beer is 20 and wine is 10
Beer is 22 and wine is 10
Beer is 22 and wine is 20
Beer is 24 and wine is 20
Beer is 24 and wine is 30
...
```

We see the rules have taken it in turns to fire.

In our current approach, arbiters are inserted to gate the firing of all rules in a conflict group that would otherwise suffer starvation. Each rule has a number of so-called *shares* which is typically unity but which can be overriden with user directives or by a future firing rate target mechanism (§3.6). An arbiter allocates one share per clock cycle.[5] Two types of stateful scheduler are generated: one is round-robin where all shares have equal priority and the other is a static priority that turns into round-robin under heavy load but resets during an idle cycle where no share is serviced. Examples are being placed online of generated code and manual control of share allocation [7].

[5]As future work we need re-abritrate on a super-scalar basis way when super-scalar rule firing is allowed (§3.3).

3.2 Multiple Action Method Invocations Per Clock Cycle

As an extension to standard Bluespec semantics, we relax restrictions on action method calls that previously could only be called once per clock cycle, such as the _write(exp) method found on the register primitives. The semantic behaviour of such methods, and those that interact with them, now needs to be hard-coded inside the main part of the compiler and modelled at compile time, rather than the component just being treated as a black-box. The commercial compiler has a similar mechanism for registers: the Ephemeral History Register [13]. In order for an action to be seemingly performed more than once per clock cycle, the intermediate side effects need to be held symbolically at compile time with only the final value(s) being committed to the real hardware. This technique is the same as used for synthesis of blocking register assignments in Verilog logic synthesis [6], but it can be used for any primitive provided the actual component has sufficient ports or bandwidth for the net effect of the composite operations to be flushed to/from the real hardware in one clock cycle (e.g. a FIFO or RAM with more ports in reality than made apparent to the user).

We perform the relevant operations in additional compile-time environments, denoted σ_x, instead of the net-level/run-time currency Σ. Bluespec registers observe RTL-like evaluate-commit semantics, so two halves to the new compile-time environment are needed. These are σ_p and σ_c for the pending updates and committed values respectively The latter also takes on the role of σ in the baseline formalism and holds local method argument bindings in a discriminated union.

Both the σ_p and σ_c environments start off empty before the first rule is compiled. Register writes and reads are now performed as Fig. 6. After each rule, a commit routine is run that copies the updates out of σ_p into σ_c. To do this, for each (r, α, e) entry in σ_p, if there is an existing entry in σ_c, such as (r, α_0, e_0), the new one is given precedence with a query-mux as $(r, \alpha \vee \alpha_0, (\alpha)?e : e_0)$. Finally, after all rules are compiled, σ_c is committed into Σ in a similar way.

Extending such superscalar behaviour to other devices, such as hardware RAMs and FIFOs, is a little more complex and requires additional or wider ports on physical devices to increase the bandwidth, but, in the future, this detail can be fully hidden from the Bluespec user in the same style as for the simple register.

3.3 Multiple Rule Executions Per Clock Cycle.

Standard semantics are that a rule is fired at most once per clock cycle.

Our enhancement to allow multiple action call invocations per clock cycle facilitates a simple approach to super-scalar performance when the same rule is applied more than once per clock cycle. This is implemented by selectively repeating a rule in the execution order under manual markup (or with targets from §3.6)[6]. Rule repetition needs to be used sparingly since combinational logic can grow quickly when operating on vectors of registers where the address equality cannot be determined at compile time. As rules are repeated, the main parameters of interest are the increase

[6] Where so-called *wires* interconnect several rules, interconnected components of rules needs to be replicated *en masse* with a wire reset operation interposed. Static analysis of which rules use which wires determines and enforces this side condition.

$$
\begin{aligned}
[\![\ \text{myreg.write}(e)\ ]\!]_{\alpha,\Sigma,\sigma_c,\sigma_p,\rho} \quad &= \quad \text{let } (v, \Sigma', \sigma_c', \sigma_p', \rho') = [\![\ e\ ]\!]_{\Sigma,\sigma_c,\sigma_p,\rho} \\
&\quad\quad \text{let } \delta = \text{if (myreg}, \alpha_1, v_1) \text{ is present in } \sigma_p \text{ then (myreg}, \alpha_1 \vee \alpha, (\alpha)?v : v_1) \text{ else (myreg}, \alpha, v) \\
&\quad\quad \text{in } (\Sigma', \sigma_c', \sigma_p'[\delta/\text{myreg}], \rho') \\
[\![\ \text{myreg.read}()\ ]\!]_{\alpha,\Sigma,\sigma_c,\sigma_p\ \rho} \quad &= \quad \text{let (en, true, args, rv)} = \Sigma(\text{myreg}) \\
&\quad\quad \text{if (myreg}, \alpha, v) \text{ is present in } \sigma_c \text{ then } ((\alpha)?v : \text{rv}, \Sigma, \sigma_c, \sigma_p, \rho) \\
&\quad\quad \text{else } (\text{rv}, \Sigma, \sigma_c, \sigma_p, \rho)
\end{aligned}
$$

Figure 6: Supporting multiple writes to a register within a clock cycle.

Rule replications	Cycles needed	Area slices	Freq MHz	Speedup ratio
1	48	575	203	1.0
2	28	3684	97	0.82

Table 2: SimpleProcessor area and performance variation for GCD computation as fetchAndExecute **rule is repeated. Platform is Xilinx Virtex 7.**

in performance in terms of clock cycles needed, clock frequency and execution time when clocked with no timing margin on the critical path. The SimpleProcessor.bsv test, available online in numerous places, provides a simple demo. As shown in Tab. 2, it nearly doubles its IPC when the fetchAndExecute rule is issued twice per clock cycle, but the maximum clock speed dropped by just over half.

The SimpleProcessor design offers very few challenges for superscalar operation, since it only uses vectors of registers without any RAMs. Vectors of registers are converted to register files in the high-level Bluespec elaboration and do not raise structural hazards. Where RAMs are used with limited port count, further rule repetition will not increase performance since the scheduler will starve the additional rules owing to conflicts. Or if the run-time arbiters are inserted, the rules will not be starved, but throughput will not increase either. Measurements of further design points are online.

3.4 Simple Access To Pipelined Operators

As another extension to the standard Bluespec semantics, we provide easy access to pipelined operators where the result is stored directly in a register without otherwise being used. If the pipeline delay is greater than one, a chain of such stores must be present in the source code and the construct starts to become cumbersome (see comments in our conclusion regarding multi-cycle schedules and generic pipeline transform). We here present the situation where the pipeline delay is one stage. This requires an extra bit of compiler-generated state for each source/destination pair. The extra bit is the scoreboard forwarding flag denoted as $SB_{\text{dest},t}$. This flag (a flip-flop) records that reads of the destination should be forwarded from the pipelined operator's read bus instead of being served from the register were the data is nominally stored.

Fig. 7 shows the two steps needed to implement a forwarding path. All method invocations have already been tagged with a unique identifier t and this is used as part of the name of the forwarding path by prefixing it with the currently-being-compiled module's instance name. A given destination register may be assigned from more than one source by a single assignment so we name the path using the tag of the read operation on the pipelined operator instead of the tag in the write operation.

A pre-scan of the elaborated source code makes placeholder entries in a per-compilation forwarding dictionary for all paths that need forwarding. This is indexed by the destination register name and lists the forwarding flops and pipelined-operator result bus for that path where each is indexed by read operation application tag. The pre-scan ensures that reads of forwarded values encountered before writes during the main rule compilation are still processed correctly. The pre-scan needs to look at both operands of an assignment to make its determination and it operates by simple pattern matching. The matched sites conform to particular use patterns where the pipelined result is (essentially) immediately stored in a register. Use of pipelined operators outside of the patterns supported by pre-scan are flagged as a compile-time errors. A multiplex of such reads is also supported because

$$[\![\ \text{dreg} \Leftarrow (g)\ ?\ M_{tt}[e1] : M_{tf}[e0]\ ]\!]$$

is treated as

$$[\![\ \text{if (g) dreg} \Leftarrow M_{tt}[e1];\ \text{if (!g) dreg} \Leftarrow M_{tf}[e0]\ ]\!]$$

In the real implementation, lightweight, referentially-transparent, combinational operations, such as negation and bitfield-extract, are also allowed before storing and are replicated on the forwarding path. Conditional store in a number of different registers is supported and those registers can still be assigned elsewhere with conventional and super-scalar writes. Pattern matching is acceptable since we only intend for limited forms to be supported, albeit with arbitrary surrounding control flow complexity.

A forwarded write operation, such as $[\![\ \text{dreg} \Leftarrow_t M[e]\ ]\!]$ is not compiled in the normal way. Instead, it is manifested by the presence of a pre-loaded entry in the starting committed updates environment, σ_c. This was put there by the pre-scan. The write itself (second step in Fig. 7) now becomes the process of adding a new disjunct into the D-input for the scoreboard flop where the new term is the activation expression α. Also, we need to load the arguments into the pipelined operator, which in the BRAM example is the compiled address/subscript expression.

The read of the forwarded register requires no handling beyond what was outlined in Fig. 6. Namely, the value from the forwarding in σ_c will be served when the forwarding flop holds. That value

$$
\begin{array}{rcl}
\text{Preload of each fwd path item :} & & \\
\sigma_c & = & \text{if } (\text{dreg}, \alpha_0, v_0) \text{ is present in } \sigma_c \text{ then} \\
& & \ \sigma_c[(\text{dreg}, \alpha_0 \vee SB_{\text{dreg},t},\ (SB_{\text{dreg},t})\ ?\ \text{rdbus}(M) : \alpha_o)/\text{dreg}] \\
& & \text{else } \sigma_c[(\text{dreg}, SB_{\text{dreg},t}, \text{rdbus}(M))/\text{dreg}] \\
\text{Write operation :} & & \\
[\![\ \text{dreg} \Leftarrow M_t[e]\ ]\!]_{\alpha,\Sigma,\sigma_c,\sigma_p,\rho} & = & [\![\ \text{dreg._write(M._read}_\text{t}\text{(e))}\ ]\!]_{\alpha,\Sigma,\sigma_c,\sigma_p,\rho} \\
& = & \text{let } (ae, \Sigma', \sigma_c', \sigma_p', \rho') = [\![\ e\ ]\!]_{\Sigma,\sigma_c,\sigma_p,\rho} \\
& & \text{let } (\alpha_1, \text{true}, [a_1], \text{rdbus}) = \Sigma'[M] \\
& & \text{let } \delta = (\alpha_1 \vee \alpha, \text{true}, [(\alpha)\ ?\ ae :\ a_1], \text{rdbus}) \\
& & \text{Din}(SB_{\text{dreg},t})\ \ \mathrel{+}=\ \ \alpha \\
& & \text{in } (\Sigma'[\delta/M], \sigma_c', \sigma_p', \rho')
\end{array}
$$

Figure 7: Register forwarding modification for pipelined operators: BRAM example.

is a reference to the read bus of the pipelined operator (with the lightweight forwarding function applied). If any other super-scalar write has been committed to σ_c for that register in the meantime, this will be correctly replacing the forwarded value when its activation expression holds. The actual update to the destination register is committed, as for the super-scalar writes, after all rules have been compiled.

The pipelined operator paradigm is especially useful for synchronous RAMs. In fact, this is the only resource we have tested the mechanism with, but it remains generic. In standard Bluespec, the BRAM is always accessed via the Put/Get interface since the read address must be handled in a different clock cycle from the retrieved data. Moreover, the standard Bluespec BRAM is normally instantiated with a FIFO on its output, whereas their use through our new mechanism does not need this overhead. Access via the old method remains possible but cannot be mixed on a single instance.

We added further concrete syntax to the parser to operate on BRAMs with a hybrid of the syntax for registers and vectors. The expansion of the new concrete syntax for the read operation is shown in Fig. 7. For write operators, $M[e] \Leftarrow v$ is converted to $M._\text{write}(e, v)$. BRAM writes are not multi-cycle and require no forwarding mechanisms: they are the same as register writes as they have the write data and the address being presented to the RAM instance in the same clock cycle. A schedulling conflict is issued between two RAM reads if the address expression does not appear identical and for writes if either the address or the value do not appear identical. Our test for identical expressions is described in §4. (Disagreements on written data could alternatively be treated as write-after-write events with the former simply being disregarded.) A conflict between a read and a write does not arise if the address expressions do not appear identical. Hence, simultaneous read and write of a RAM via a single port, as supported by much actual hardware, is also exploited by the scheduler. This is believed to be an improvement on the standard Bluespec approach[7].

3.5 Static Load Balancing (Proposal)

This section describes experiments that should be ready at the time of the conference.

The standard Bluespec compiler does not statically load balance leaf components (here called FUs). The binding of work to hardware resources is either manual, with TLM-style method calls on the FUs, or can be left to run-time logic using the server pool library.

But static load balancing should be easy to implement as a Bluespec extension by treating the user's instance names as virtual names where the scheduler decides the virtual to physical mapping. A mapping between virtual and physical resources introduces a set of schedulling conflicts. Our first approach will be straightforward. We run the main compile step first with virtual names and then make a virtual to physical mapping that does not introduce any rule conflicts. This approach can also spot where two BRAMs can be stored inside a single larger physical BRAM. In a future version we will allow the binding to be updated as part of the hill climbing optimisation (§3.6).

Where an FU is freely instantiatable, such as a stateless ALU, the number to instantiate can be freely chosen by the compiler. When a pipelined operator contains state, such as a BRAM, a colouring of operations to map them to the available ports must be chosen by the compiler. A BRAM can also be trivially replicated by the compiler to increase the read bandwidth, provided the writes are kept synchronised. These decisions are precisely the same as solved by HLS tools such as LegUp and Kiwi [4, 10]. Probably a mechanism for compiler-chosen bindings will benefit from manual overrides where the designer wishes to exercise tight control, but this is relatively easy to provide with future mark-ups embedded in the source code.

Where the Put/Get interface is traditionally used for operations on a stateless FU, such as a three-cycle multiplier, BlueSpec introduces unfortunate phantom state: the get() operator must be applied to the same instance that was put(v). Our pipelined operator

[7] Super-scalar writes on RAM arrays are also allowed in the real implementation since σ entries have an address option field. Only one address expression is allowed per physical port owing to the undecidable name alias problem: the compiler can generally not tell if two expressions refer to the same location.

extension reduces the need for that coding style, but where it remains, more wanton instantiation of resources should perhaps be used, with the assumption that the compiler will more than compensate.

3.6 Rule Firing Target Rates

Most of the extensions just described give the compiler greater freedom to tune the firing rates of rules. The compiler can now choose how many FUs to instantiate, whether to replicate rules and whether to throttle rules with arbiter shares. One panoramic control paradigm for these mechanisms is for the user to specify relative or absolute rule firing rate targets. The compiler cannot hope to accurately meet these targets in all but very simple designs owing to enormous uncertainty about the duty cycles and cross-correlations of the lowest-level predicates in the explicit and intrinsic guards throughout the design. Nonetheless, a fluid flow equilibrium model has been used based on default assumptions with the hope that this will at least respect the real behaviour in terms of relative firing rates. When there are found to be major discrepancies, the advanced user can add assertions about leaf predicate rates and correlations. These may also come from profile-directed feedback.

Target rates are described inside Bluespec pragmas either next to the rule (alongside other standard rule markup such as nosplit) or with the name of the rule. Examples supported by the prototype are:

```
(* target_rate = 2.0 *)
(* target_rate other_rule = 0.125 *)
(* target_rate third_rule = 2.0 * fourth_rule *)
(* target_rate third_rule = 2.0 when running *)
```

To use the rate-based approach, some number of leaf component methods and external methods should be annotated with constraints on their tolerable and expected firing rates. For instance, a register can support any write density, as can a FIFO, and for the FIFO the enqueue and dequeue rates should be identical. For an arbiter, the holding rate of a unary output is its share of the total share holding for that arbiter. Then, rules for which performance goals are desired are annotated with target rates as a real number multiple or fraction of the clock rate or of the rate of other rules. Finally, arbitrary predicates can be given target holding rates in the same style.

Often there may be more than one major operating mode for the design. The requirements and expectations are commonly different in each mode. Where the mode is manifest in the hardware state of the design, a 'when' clause, as in the fourth example, can be used.

This rest of section describes future work which should be complete at the time of the conference.

A hill climbing procedure in the HPR L/S libary then attempts to minimise the error between the target firing rates and modelled firing rates as it adjusts the bindings, shares and rule repetitions. The firing model uses the Zadeh operators [16] from Fuzzy Logic where AND is MIN and OR is MAX. These are more tolerant to unknown correlations than the probabilistic alternatives (where AND is product). Although there are no combinational loops in the guard expression network, there are sequential dependencies, so the model is evaluated until convergence at a given design point, before navigating to a neighbouring design point.

This rate model has not yet been shown to work or be useful. Some example output is online. For now the user should rely on lower-level markup of rule repetition count and the number of arbiter shares to allocate.

4 CONCLUSION

We have presented the implementation of several extensions to the behaviour of the standard Bluespec compiler that we believe will be generally useful. But these do not change the semantics of the Bluespec language. Without automatic insertion of arbiters, there exists a stuttering equivalence between the circuits generated by our compiler and the standard one. The new arbiters change the behaviour of real programs, while strictly keeping within the original guarded-atomic-action paradigm of the Bluespec approach. Our approach automatically provides greater fairness amongst rules, whereas the traditional approach is to consider rule starvation an error that should be manually corrected by the designer. (The designer will add additional conjuncts to the explicit guard of a greedy rule to make it fire less often: these clauses could refer to ready signals from a manually-instantiated stateful arbiter.) Even with that proviso, the real behaviour of both compilers makes more-or-less arbitrary decisions about how write hazards should be resolved. Ultimately it is up to the designer to write 'clean' code that uses FIFO-style interfaces and valid-tagged data so that unwanted write races are avoided.

Beyond a certain level of rule repetition, exponential logic growth will become prohibitive and perhaps generic re-pipelining transforms [8] can be applied to the resulting RTL before logic synthesis. This is easily tested within the HPR L/S library, but it would not exploit the freedom to explore alternative static schedules in the front end. Previous work extended Bluespec to support multi-cycle FUs such as infixed use of a divider that goes busy for several clock cycles or that has a long combinational delay [11]. This prevents the clock frequency being reduced by slow, yet seldom-used, sub-circuits, but it could only exploit a fraction (eg. one third) of the processing power of today's common pipelined FUs, such as FPGA DSP units since the effective initiation interval is the latency. Future work will extend our register forwarding approach with scheduller integration to fully exploit FUs with latency two and higher.

Building the compiler on top of a normal-form, memoising-heap logic system means that identity between expressions is often found, even though they were formed by complex operations applied to expressions that looked quite different in the concrete syntax. This means that certain idempotent operations, particularly writes to registers and RAMs and reads of RAMs, do not conflict when they are the same operation in reality. The same goes for any leaf method that is invoked more than once by a rule or in a clock cycle. Fewer conflicts means greater parallelism in the final design. We provided a global switch to enable this new behaviour and a 'not_idempotent' optional pragma for method definitions that should not participate, such as for FIFO queue/dequeue and other counter style operations.

The multiple-updates facility offers a much richer implementation space for the compiler. Together with virtual names for load

balancing and rule repetition, we have provided a solid set of mechanisms that allow a compiler to trade time for space to achieve performance metrics. In the future, our rate-based scheduler should offer a sophisticated approach to deploying these mechanisms. But our currently-supported manual markup of rule arbiter shares and repetition count is a good start that also preserves full control for engineers.

You may download an open source tarball of the Toy compiler with these new extensions from [7].

REFERENCES

[1] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. 1212–1221.

[2] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. , 11 pages. https://doi.org/10.1145/289423.289440

[3] Thomas Braibant and Adam Chlipala. 2013. Formal Verification of Hardware Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044 (CAV 2013)*. Springer-Verlag New York, Inc., New York, NY, USA, 213–228. https://doi.org/10.1007/978-3-642-39799-8_14

[4] A. Canis, Jongsok Choi, B. Fort, Ruolong Lian, Qijing Huang, N. Calagar, M. Gort, Jia Jun Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson. 2013. From software to accelerators with LegUp high-level synthesis. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*. 1–9. https://doi.org/10.1109/CASES.2013.6662524

[5] Nirav Dave, Arvind, and Michael Pellauer. 2007. Scheduling As Rule Composition. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE '07)*. IEEE Computer Society, Washington, DC, USA, 51–60. https://doi.org/10.1109/MEMCOD.2007.371249

[6] D. Greaves. 1995. The CSYN Verilog Compiler. In *International Workshop on Field Programmable Logic, FPL'95. (Lecture Notes in Computer Science)*, Vol. 975. 198–207.

[7] David J Greaves. 2014. *CBG-BSV Toy Bluespec Compiler*. http://www.cl.cam.ac.uk/~djg11/wwwhpr/toy-bluespec-compiler.html

[8] David J Greaves. 2019. *Generic Pipeline Transforms*. http://www.cl.cam.ac.uk/users/djg11/wwwhpr/generic-pipeline-xforms

[9] David J. Greaves. 2019. Research Note: An Open Source Bluespec Compiler. *arXiv e-prints*, Article arXiv:1905.03746 (May 2019), arXiv:1905.03746 pages. arXiv:cs.PL/1905.03746

[10] David J Greaves and Satnam Singh. 2011. Distributing C# methods and threads over Ethernet-connected FPGAs using Kiwi. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. 1–9. https://doi.org/10.1109/MEMCOD.2011.5970505

[11] Michal Karczmarek and Arvind. 2008. Synthesis from Multi-cycle Atomic Actions As a Solution to the Timing Closure Problem. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design (ICCAD '08)*. IEEE Press, Piscataway, NJ, USA, 24–31. http://dl.acm.org/citation.cfm?id=1509456.1509475

[12] Rishiyur Nikhil. 2004. Bluespec SystemVerilog: Efficient, Correct RTL from High-Level Specifications. *Formal Methods and Models for Co-Design (MEMOCODE)* (2004).

[13] D.L. Rosenband. 2004. The Ephemeral History Register: Flexible Scheduling for Rule-based Designs. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '04)*. IEEE Computer Society, Washington, DC, USA, 189–198. https://doi.org/10.1109/MEMCOD.2004.1459853

[14] Daniel L. Rosenband and Arvind. 2005. Hardware Synthesis from Guarded Atomic Actions with Performance Specifications. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design (ICCAD '05)*. IEEE Computer Society, Washington, DC, USA, 784–791. http://dl.acm.org/citation.cfm?id=1129601.1129712

[15] B. N. Uchevler, K. Svarstad, J. Kuper, and C. Baaij. 2013. System-level modelling of dynamic reconfigurable designs using functional programming abstractions. In *International Symposium on Quality Electronic Design (ISQED)*. 379–385. https://doi.org/10.1109/ISQED.2013.6523639

[16] L.A. Zadeh. 1965. Fuzzy sets. *Information and Control* 8, 3 (1965), 338 – 353. https://doi.org/10.1016/S0019-9958(65)90241-X

HPR L/S Memoising Heap

The Toy compiler is built on a logic synthesis library, called HPR L/S, that intrinsically implements constant propagation, subexpression sharing and logic minimisation using Espresso, so the generated RTL is not overly verbose.

This library uses a memoising heap that implements many distributive laws. Laws for multiplexors embody context-sensitive simplification within a multiplexor argument sub-tree. Laws also understand one-hot encoding and limited numerical ranges. The rules of the heap aim to put most expressions into a normal form, for instance, by sorting the operands to commutative and associative operators. Beyond this normalisaion, every apparently different expression is given a different natural number called its heap index. For booleans, negation is represented by negating the heap index.

There are intrinsic limitations to expression identity checking arising from computability theory. Nonetheless, the memoising heap is a helpful and cheap tool. So where we need a conservative test for array subscript equality or whether argument expressions to a leaf component are equal, we used identity of heap index.