Layering RTL, SAFL, Handel-C and Bluespec Constructs on Chisel HCL.

David J Greaves
Computer Laboratory, University of Cambridge, UK.

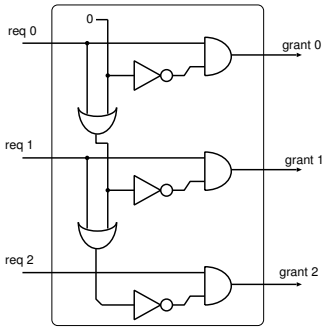

Fig. 1: Example circuit, a static-priority arbiter.

***Abstract*—Chisel is a hardware construction language that supports a simplistic level of transactional programming via its Decoupled I/O primitives. In this paper we describe extensions that layer popular design paradigms on the Chisel substrate. We include RTL, SAFL-style functional hardware description, Handel-C message passing and Bluespec rules. We then briefly discuss interworking between these design styles.**

I. Introduction

A Hardware Construction Language is here defined as a programming language for programs that 'print out' the circuit diagram, or netlist, for a hardware design. They can vary greatly in their expressive power. Of interest are those that ensure the generated netlist is not only syntactically correct but also semantically correct. The latter implies that design rules such as no two gate outputs are directly connected are obeyed, or that the widths of two busses being multiplexed is the same. Two notable recent players are Chisel [1] and HardCaml [2]. An earlier language was Lava [3], [4]. All three make great use of the standard combinators and paradigms found in functional programming languages, such as ML and Haskell. Bluespec, another recent language, also leverages this paradigm, but differs in a couple of ways. The first difference is that it re-implements the functional language itself with its own syntax and parser, whereas the other three are DSLs (domain-specific languages) that are embedded in a powerful host language. Chisel is embedded in Scala, Lava was embedded in Haskell and HardCaml is embedded in OCaml. We shall cover the second Bluespec difference later on. All of these processing systems generate syntactically-correct RTL that abides with port binding rules.

The two everyday RTL languages, Verilog and VHDL, embody 'generate statements' and 'generate variables' which provide the hardware construction aspect just defined. These languages have staged execution (or embody metaprogramming) where part of the program is run at compile time and the remainder at run time. The compile-time execution is known in the field as the 'elaboration'. All generate statements and genvars only exist during elaboration and only influence the amount of hardware generated. The generated hardware, when switched on and clocked, provides the run-time execution. Sometimes it is a little tricky to work out whether something is executed entirely at compile time or else embodies some run-time behaviour (e.g. a function invoked with constant parameters that makes use of a run-time free variable in its body), but generally it is obvious.

The execution of the combinators and functional programming we find in the hardware construction languages already mentioned is constrained to the elaboration stage. They all generate a finite amount of hardware with static allocation of user variables to registers - there is no run-time dynamic storage allocation. Nonetheless, both stages of execution potentially embody '`if`' statements and other control flow statements such as case or switch. When one of these statements alters which others get executed we have 'data-directed control flow'.

As a concrete example, in Chisel, the following Scala fragment creates a static-priority arbiter with n inputs and n outputs. By extending the Module class it becomes a separate hardware component (Verilog module) in the RTL output from Chisel. All Chisel modules have an I/O Bundle which commonly has a fixed set of terminals, but in this example all of the I/O pairs are generated by the higher-order functions map and fold, both of which are fully executed at compile time (elaboration time). The '&', '|' and '!' symbols are overloaded to generate real AND, OR and NOT gates and the 'new Bool' generates the wire from one level to the next.

```
class genericPriEncoder(n_inputs : Int) extends Module
 {
   val io  = new Bundle {  }
   val terms = (0 until n_inputs).map (n => ("req" +
        n, "grant" + n))

   terms.foldLeft (Bool(false)){ case (sofar, (in,
        out)) =>
      val (req, grant) = (Bool(INPUT), Bool(OUTPUT))
      io.elements += ((in, req))
      io.elements += ((out, grant))
      grant := req & !sofar
      val next = new Bool
      next := sofar | req
      next
   }
 }
```

A semantic difference between hardware construction languages (like HardCaml, Lava and Chisel) and standard RTL logic synthesis is that hardware construction languages (mostly) lack behavioural flow. This becomes a clearly distinctive differentiator when we consider infinite loops: hardware generally lasts for ever, repeating the same behaviour over and over. In standard RTLs this is expressed directly as a behavioural infinite loop in the source code. On the other hand, HCLs execute once with all loops exiting at compile time. A related distinction is the presence of a *packing transform* that combines multiple updates expressed sequentially into a parallel update of all of the left-hand sides at once (on a clock edge). This conversion at compile to a set of parallel register assignments is also known as behavioural elaboration or symbolic evaluation. It is done by collating the assignments to variables and array locations. This process is exactly the procedure used to formally define imperative languages using denotational semantics. Richer behavioural expression uses program-counter inference, where each compile-time eternal loop pauses at a number of different places in its path, each requiring a code point in a run-time (hardware) program counter.[1]

None of Lava, HardCaml or Chisel provides data-directed control flow, although Chisel, at least, supports relatively usable synthesis of multiplexor trees from nestable constructs using its 'when' statement. Instead, in these hardware construction languages, the user must manually instantiate a finite-state machine to get this effect, whereas in our RTL layer of §II-A and with with HLS (high-level synthesis) tools, such as LegUp [5] and Kiwi [6], this is their natural behaviour.

II. Building popular expression styles on top of Chisel

We now give some examples of richer coding styles that can usefully be constructed on top of the netlist generation facilities of Chisel.

A. RTL-style: Using eternal process loops containing data-dependent control flow.

The main feature of behavioural register transfer language (RTL) is a thread of execution inside an infinite (or eternal) process loop. The behaviour of the hardware mirrors the behaviour of the thread. Synthesis standards and house styles in individual companies may restrict the coding styles used, but in general, any variable can be updated in any number of places in the loop and, flow of control can take conditional branches and the loop can be explicitly paused at any point waiting for the next clock edge. The most recent update to each variable at that point is presented to the D-inputs of the registers and they take on that value at the clock edge. In Verilog, the pause is typically denoted with '@(posedge clk)'. Chisel provides an analogous 'step(n)' method in its testbench stimulus language but does not provide such a construct in its synthesisable subset: it is an HCL and not an HDL. This is because synthesisable code extends Chisel.Module but the step method is unbound inside Module. To avoid confusion between testbench and application code, we used upper case to define 'STEP()' and 'STEP(n)' to pause for 1 and n clock cycles respectively in application code.

Turning now to the process of finding the most recent assignment to a variable, the packing transform, we find the Chisel internal mechanisms already fully support guarded updates in order to provide its 'when/else/otherwise' and 'switch' statements. However, when a succession of these are normally placed in a Module body (or nested inside each other) they behave in *parallel*, with only the WaW (write after write) hazards being resolved according to the elaboration thread's program order with the last write having the highest priority. Of course, this contrasts with everyday imperative programming where they act in *serial* with the most recent write being the value used when reading the variable or array location. The same difference, with respect to the value of a variable occurring on the right-hand side of an assignment in an expression, is manifested by Verilog's two different assignment operators, blocking and non-blocking.

Owing to Scala's wonderful extensibility, it is rather easy to introduce some 'always' syntax that introduces a run-time process loop. The keyword 'always' is used in Verilog to define a thread that infinitely loops: it is short for 'initial while (1)'. The necessary Scala to essentially define a new keyword is

```
object always
{
  def apply(codeBlock: => Unit) =
  {
    rtl_compile(codeBlock)// csyn-style RTL compiler call
  }
}
```

where the 'object' keyword defines the static methods of a new class, called 'always', that can be directly applied to a parsed but uncompiled block of source code. Also, to enable Scala's own **if** statement to still be freely used during the elaborate phase, we defined our own IF, ELIF and ELSE constructs for use inside always blocks for run-time, data-dependent control flow. Again, each takes a block of code and the first two additionally take a preceeding Chisel.Bool curried argument that is the normal guard expression for a conditional block.

This now enables us to write, for example, the following RTL-style Chisel module. It generates an alternate pattern of 1 and 2 on its output that is interspersed with the number 3 for two cycles after the 2 when the input is asserted.

```
class RtlExample extends Module {
  val io  = new Bundle {
    val din = Bool(INPUT)
    val mon = UInt(OUTPUT, 3)
  }
  always {
     io.mon := UInt(1)
     STEP()
     io.mon := UInt(2)
     STEP()
     IF (io.din) { io.mon := UInt(3); STEP(2); }
  }
}
```

[1]Older Verilog compilers would support state machine inference from the Verilog's control flow, but this was discontinued (or at least made to flag warnings) when the synthesisable subset was defined.

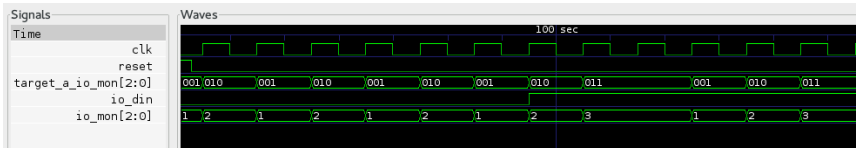

Fig. 2: Timing diagrams from RTL-style demonstration.

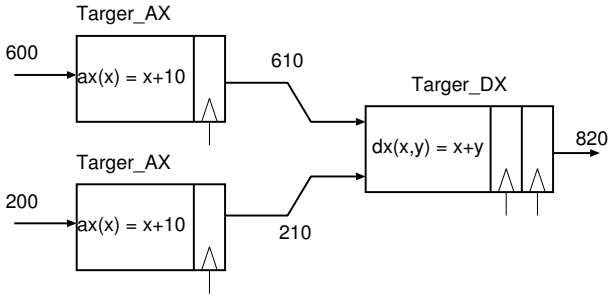

Fig. 3: Example, trival TLM calling structure.

The apply method for always contains an RTL compiler along the lines of CSYN [7]. This is similar to many other software-to-hardware compilers. It uses a one-hot encoding of the program counter for each region of code that starts after a STEP call. At the hardware level, each region of code has an activate input that makes it run. Using the same mechanisms that Chisel already uses for its 'when' statement, the side effects of assignments are only enabled when the appropriate region is active.

The '`IF`' statements fork and join the activate path around the conditional code. With a suitable stimulus on the `din` input, a simulation of the generated RTL gives the expected waveform on the `mon` output, as shown in Figure 2. The program counter uses four additional flip-flops, not manifest in the source code. FPGA tools or Design Compiler might typically re-encode the one-hot state machine to a more optimum form for the target technology, so there is probably little point putting more optimisation effort into our extensions.

B. SAFL-style: Statically-Allocated Functional Language

In this subsection we report on a DSL extension to Chisel that generates SAFL-like transactional circuits.

SAFL [8] is a hardware synthesis language that uses a subset of ML syntax; in particular, any recursion used must be tail recursion. This ensures finite circuits are generated. (Ghica's *Geometry of Synthesis IV* [9] is also functional but added a stack to the generated hardware to support recursion.) SAFL is best described as a synthesis language, rather than a construction language, since the function applications and the IF statements are projected through to the execution phase. It uses its own compiler and, in contrast to HardCaml, is not a DSL embedded in ML. The language was described as 'resource aware' meaning that the engineers know exactly how much hardware they will generate with each statement they write. Being functional, it is highly-ameanable to time/space folding since, without imperative updates, there are no RaW hazards that might get re-ordered.

In the SAFL-style, there are two baseline rules that control the amount of hardware generated:

1) Leaf operators, such as '+' and '*', occurring textually in the souce code are freshly instantiated in the hardware for each occurrence.
2) The same goes for function definitions, which means function applications of a named function are serialised over the shared body with argument and return value multiplexors.

Time-to-space folding can be implemented in SAFL-style languages with a uniquify operator, called UF. In the following example, this replicator provides a fresh, identical copy of the complex multiplier, thereby enabling both arguments of function $g()$ to be run at once.

```
fun cmult x  y =  (* SAFL-ML complex multiplier *)
    let ans_re = x.re*y.re - x.im*y.im
    let ans_im = x.im*y.re + x.re*y.im
    in (ans_re, ans_im) // 4 multipliers, 2 adders.

let use_time  = g(cmult a b, cmult c d)
(* uses 4 multipliers, 2 adders + resources for g *)

let use_space = g(cmult a b, (UF cmult) c d)
(* uses 8 multipliers, 4 adders + resources for g *)
```

Chisel has an underlying datatype for generated hardware called Bits. Chisel also provides the 'Valid' and 'Decoupled' interface wrappers in its library that add request and ready signals to an interface. But these wrappers do not automate the wiring inside a leaf copmponent; they only help with inter-module wiring. For a SAFL-style portion of a circuit description, every expression needs to be associated with handshake wires. Accordingly, we work at the Chisel.Bits level and implemented a guardedBits class that adds handshake wires to every subexpression in an expression. One might think this will force us to redo all of the work in the Chisel library that already overloads every arithmetic and logic operators, but owing to Scala's facility of 'implicit functions' to provide automatic coercion from one form to another when appropriate, this has not been a problem so far. For instance, a constant expression, such as UInt(42), can be freely used as a guarded expression provided the appropriate implicit coercion has been imported with '`using SAFL._`' at the head of the file. When used, on a simple constant, the implicit coercion will add an 'always ready' acknowledge output signal, that is wired to logic 1, and a request input signal that is simply ignored.

SAFL-style operators, including simple primitives like addition, are treated as function calls. This is already how Chisel

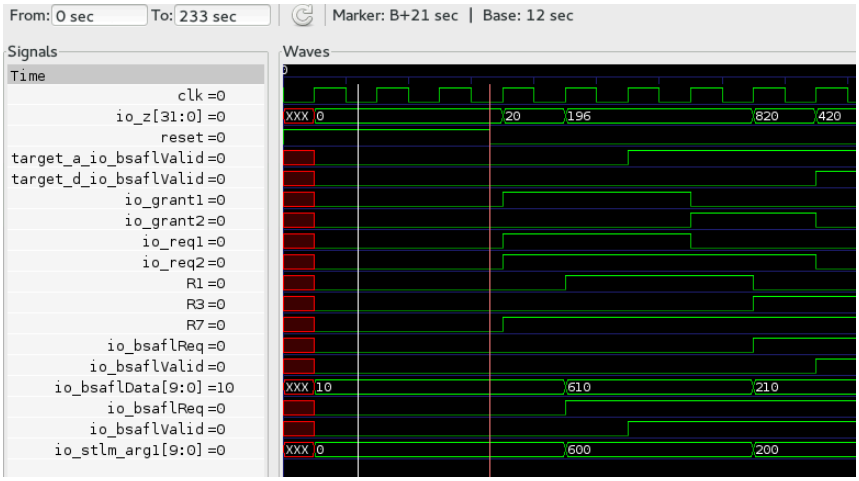

Fig. 4: Timing diagrams from SAFL-style example.

works. SAFL uses strict call-by-value, so all function calls can only execute their bodies when all of their input arguments are ready.

By defining a transactional bundle that extends the net-level bundle of Chisel, and performing operations on that, we can enrich the language so that the compiler performs automatic synthesis of handshaking nets. Callable components are generally either fully-pipelined, meaning they can start a new operation on every clock cycle, but have some fixed latency, or else go busy while internally processing for some potentially variable number of cycles. We support both styles of component in the same framework. To wrap up a simple Chisel function with a fixed delay of one clock cycle, making its I/O is fully transactional, we can use the style of the following example:

```
// An example TLM callable Module: One method,
// fully-pipelined with delay of one.
class Targer_AX extends Module
{
  val io = new TLM_bundle_lc
  { // Register TLM callable function with one pipeline
       delay.
     tlmBind_a1(ax_fun _, 1)
  }
  def ax_fun(x:UInt) = Reg(UInt(32), x + UInt(10))
}
```

The function `ax_fun` has been registered as a callable method in the module Targer_AX. It can now be called as 'x1' in the following fragment. It can also be called as 'ax_fun' owing to Scala's introspection. The '`_a1`' suffix on the bind denotes that it is a function of one argument (monadic). The second argument to bind of unity denotes the fixed pipeline latency of one clock cycle. The '`Reg`' call in the function body is Chisel's standard way of delaying a result one clock cycle. Note that operations as simple as 32-bit addition of a constant would typically not warrant a pipeline delay in contemporary technology; we are just using addition as a simple example.

Having multiple methods on an instantiated object has no semantic meaning in a purely functional setting, but most real designs use RAMs or otherwise contain state and then the association of methods to objects makes sense. Therefore we allow more than one function to be exported from a given Module (multiple bind calls), thus supporting overloading and so on, which means that 'x1' would become ambiguous when more than one monadic function is registered as TLM-callable.

We can now compose transactional modules such that they can makes calls on each other with syntax such as:

```
val target_a = Module(new Targer_AX())
val target_d = Module(new Targer_DX())
val ... = target_d.io.x2(target_a.io.x1(e1),
     target_a.io.x1(e2))
```

The above fragment illustrates a key aspect of SAFL semantics, that the `target_a` is called twice, potentially in parallel, before target_d's TLM-method of arity two is invoked. Note, SAFL is strictly call by value. But with only one instance of Targer_AX, the executions of target_a are automatically serialised in the time domain. A holding register is tacitly generated to store the output of the first execution, but visible in the generated RTL.

As is now apparent, in our SAFL-on-Chisel implementation we do not need a special uniquify combinator. Instead, the number of instances of a callable target is controlled, in the obvious way, by the number of times Scala's `new` operator is applied to the target class. When different instances are specified in a parallel context, they are invoked in parallel, but when an instance reference is repeated the operation is automatically serialised, despite it being in a potentially parallel context.

Because guardedBits is used for each argument, under the hood, each call site has its own `req` and `valid` handshake pair to a wrapper around an instance of the target class. In the wrapper, a priority encoder selects one contending call site requester for service and acknowledges that call site when the callee completes.

Registering completely constant values is a waste that should be cleaned up in back-end tools by constant folding optimisation, but this may not be the case for FPGA where the unavoidable global/power-on reset has an observable consequence. Therefore we implemented AlwaysReady flagging, where the guardedBits class has an indication of expressions that, although not constant, are permanently live and can always be read without a handshake. User variable register outputs are the main example. By propagating this information inside the Chisel/SAFL system we avoid relying on the backend, as well as usefully simplifying the circuit which helps with debugging owing to less complex handshake wiring. Specifically, it helps with our policy for argument synchronisation, presented in the next paragraph.

At a call site, there are various possible implementations for how long the argument resources are tied up — are they released at the start of the body computation or at the end of it? For variable delay callees, the choice of this matter will affect the user's coding style - he may need to register the input arguments if the framework is not going to hold them steady and he needs to refer to them beyond his first clock cycle.

```
class Targer_DX extends Module {
  val io = new TLM_bundle_lc
  { // Register TLM callable diadic function with two
      pipeline delays.
    tlmBind_a2(dx_fun _, 2)
  }
  def dx_fun(x:UInt, y:UInt)= Reg(Reg(UInt(32), x + y))
}
  -- snip --
  val unit_a = Module(new Targer_AX())
  val unit_b = Module(new Targer_BX())
  val unit_d = Module(new Targer_DX())

// Diadic - single use test
//   val a0 = unit_d.io.run2(unit_a.io.run1(arg1K),
    unit_b.io.run1(arg2K)

// Diadic - reuse of same component AX
   val a1 = unit_d.io.run2(unit_a.io.run1(arg1K),
       unit_a.io.run1(arg2K))

// Invoke and down convert to unguarded for rest of
    design
   val a2 = SAFLImplicitx.
        ex_drop_chisel_data_from_guarded(answer)
   io.z := a2
   io.v := a2.isValid()
```

Envisioning FPGA targets, that are register rich, we implemented the following default strategy. The activate request is forwarded to all argument expressions in parallel. If the valids come back in differing clock cycles then the results from the earlier ones are registered locally and the requests de-asserted. This ensures that any resources that may be tying up completion of the others in the parallel composition are freed, avoiding deadlock. AlwaysReady expressions are ignored in this consideration since they always come back immediately and would be registered unnecessarily. When all inputs are ready, the callee is triggered and the callee **cannot** rely on the input arguments being held beyond the first clock cycle since the argument resources are then freed (their requests de-asserted). This makes those resources available for use elsewhere, such as in the body of our function.

In the above example, AX is a method with a fixed delay of one pipeline stage and DX has two stages of delay. As stated, one would normally encapsulate a much richer computation than the simple additions used for example purposes. Figure 4 shows the timing waveforms as the two calls to AX get serialised, including the inputs and outputs to its priority arbiter. It shows the answer 820 finally being generated, after the single instance of AX is used in turn for each sub-expression. The top-level request is invoked by the `ex_drop_chisel_data_from_guarded` method which here is explicitly called but could be inserted by Scala's implicit mechanism.

Finally we show the coding style for TLM callbacks that have variable delay. We make explicit connection to the underlying handshake nets, but this is now the exceptional case (compared with Chisel's Valid/Decoupled). These connections are made by reading from and assigning to the results of getValid() and get getReq(). Here we supply -1 for the fixed latency operand to the tlmBind methods, thereby informing the TLM subsystem not to generate its own timing shift register between the request and valid handshake wires.

```
class GCDunit extends Module {
  val io  = new TLM_bundle {
     tlmBind_a2(gcd_fun _, -1)
  }
  def gcd_fun(arg_a :UInt, arg_b :UInt):UInt = {
     val x = Reg(UInt())
     val y = Reg(UInt())
     val p = Reg(init=Bool(false))
     when (io.getReq() && !p) {
       x := arg_a
       y := arg_b
       p := Bool(true)
     }
     when (p) {
       when (x > y)  { x := y; y := x }
       .otherwise    { y := y - x }
     }
     when (!io.getReq()) { p := Bool(false) }
     io.getValid() := (y === Bits(0) && p)
     x
  }
}
```

C. Supporting Seq/Par and Handel-C-like Channel Communication

In this section we demonstrate Handel-C-like Seq/Par and Channel Communication on top of Chisel.

Handel-C [10] is a parallel programming language for hardware design based on Occam [11] and CSP [12]. Process loops are again used to define eternal hardware. It uses explicit message passing between each process using a flat, global namespace of channels. Shared variables are banned, giving a pureness and amenability to time/space folding, as also found in Erlang [13].

A notable feature of Handel-C is the PAR block constructor that places statements in parallel. There is also the SEQ

constructor which provides conventional sequential execution of statements, like a begin/end construct in RTL processes and other imperative programming languages. Power arises from the ability to arbitrarily nest these blocks inside each other. Handel-C had originally a very strict and easy-to-implement approach to projecting these statements into hardware. Each leaf assignment statement, to a local variable or output channel, consumed exactly one clock cycle. PAR blocks nested inside other PAR blocks have no effect and can be flattened. The same applies for SEQ blocks nested inside SEQ blocks. Therefore, the outer process loop, without loss of generality can be considered always to start in SEQ mode and the RTL process loop compilation strategy, already described, can be borrowed. This supports SEQ and IF. An extension is required for PAR. As in previous Occam-to-hardware transforms [14], when a PAR block is encountered, the surrounding SEQ block activity net is wired to the activate inputs of all sub-blocks within the PAR statement. A hardware barrier is placed at the exit from the PAR block that waits for the last arriving thread before proceeding with the next SEQ statement. The hardware barrier requires one flip-flop, but otherwise follows the same control flow paradigm as the all-inputs-ready block for the SAFL TLM arguments.

Our implementation, being based on the RTL elaborator, implements the RTL assignment packing transform, where more than one update is made per clock cycle. This is an optimisation with respect to standard Handel-C which could be disabled when desired.

Handel-C has blocking send and receive primitives operators which each take a named channel as an argument. The output operation is diadic, taking also the value to be sent. The input operator is monadic and is a leaf expression. The conventional syntax is to use the exclamation mark for output and the question mark for input. For embedding in Scala we used straightforward overloading of the channel class, with 'chan.send(val)' for output and 'chan()' for input. The channels behave as FIFOs and, in our implementation, their depth is set in the constructor.

We defined the class HChan to serve as Handel-C's channels. They are a wrapper around the Decoupled Queue FIFO found in the standard ChiselUtil library. Unlike the Chisel queues, they do not have net-level connections in their I/O bundle, since the queue and dequeue are method calls.

Here is an example of Handel-C coding style using just one channel that is written in one process and read in a second process. The body of the sending process is a SEQ block as that is the default for processes. It first sends the number on the channel and then sends vv. It increments vv in parallel. In strict Occam, by doing the increment in parallel we save a clock cycle, whereas the packing transform, when in its default enabled condition, will do this anyway and the PAR construct makes no difference in this tiny example. The variable vv could be operated on by both processes using this coding style, leading to potential race conditions. Better style would be to restrict the scope of all variables by defining them inside the always blocks.

```
class HandelExample extends Module {
  val io  = new Bundle {  val mon = UInt(OUTPUT, 3) }
  val porta = new HChan(UInt(32), 1)
  val vv = Reg(UInt(32), init=UInt(0))
  always {
     porta.send(UInt(4))
     PAR { porta.send(vv)
           vv := vv + UInt(1)
         }
     STEP() // Please eliminate me!
  }
  always {
     io.mon := porta() & UInt(7);
     STEP() // Please eliminate me!
  }
}
```

No 'STEP()' calls should be required in the always blocks owing to the I/O operations being blocking. However, our current, first draft implementation, suffers from a fantom combinational loop without them, owing to the possibility, for instance, that the blocking channel read in the second process is always ready. Handel-C would not suffer that owing to the assignment to io.mon taking one clock cycle by itself. We could change the implementation of the I/O primitives to always require one clock cycle. Better, in the long run, would be to use a post processor that makes a global retiming of the design, inserting the clock control at points which are balanced in terms of critical path.

D. Supporting Bluespec-like Guarded Atomic Rules

The same infrastructure, with a little tweaking can also provide the guarded atomic updates of Bluespec logic synthesis [15]. Bluespec also provides a global scheduller that avoids resource starvation when updates contend: we see no problem implementing that within our framework in the future.

Although SAFL and Bluespec both automatically generate bi-directional handshake wires for every callable method, the time-domain semantics are completely different. Bluespec proactive behaviour all originates from Bluespec *rules* and all code is enclosed in the body of a rule or the body of a method that is called from a rule, or indirectly via further methods. Additionaly, Bluespec associates every expression with an *implicit guard* that is the conjunction of the implicit guards of all the sub-expressions within it. Leaf terms in expressions are either 'AlwaysReady' with implict guard always holding, or else can have an *explict guard* which is an arbitrary Boolean expression (that itself may have implict guards). Further guards arise from language-intrinsic conditions which include a register can be written at most once in a clock cycle and a rule can fire at most once in a clock cycle. A method call is only executable when the implicit guard of the method itself holds and also the guards of all the argument expressions passed in. Hence, in the following fragment, in regular Bluespec, the guards for e1 and e2 must simultaneously hold, together with the guards for all the resources used in the bodies of the three methods mentioned, then the rule which embodies this code can fire.

A Bluespec rule is an unordered list of commands to be fired in parallel (atomically) and an optional explicit rule guard that is added to the conjunction of all the implicit guards

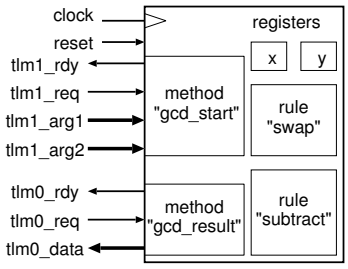

Fig. 6: Block diagram of GCD unit with Bluespec-style handshake nets for each method.

of the commands. The principle idea of Bluespec is that a rule fires atomically and this has a natural parallel in the generated synchronous hardware where all flip-flop master sections within a clock domain are simultaneously applied to their slaves. The updated values all become visible atomically. Additionally, the parallel composition of components in the rule body is the same as the default paradigm for a Chisel Module body, hence we do not need the SEQ/always process abstraction.

```
val ... = target_b.io.x2(target_a1.io.x1(e1),
    target_a2.io.x1(e2))
```

Bluespec-like rules can be embedded in Scala following the concrete syntax illustrated in the right-hand-side of Fig. 5. This generated I/O nets are illustrated in Fig. 6. If a method is registered in a BSV_bundle with a `bsvBind` callback, the Bluespec semantics are used. Standard Bluespec embodies a scheduller that packs operations into a clock cycle, but which does not make multi-cycle schedules. Hence all method calls must be non-blocking so that execution never extends beyond one clock cycle.

As with SAFL, a bi-directional pair of handshake wires is automatically generated for each method or sub-expression. The 'request' input is the same for both systems, but the 'valid' output that indicates a SAFL function call is complete is replaced with a Bluespec 'ready' output that indicates an expression or method can be invoked. The Bluespec 'ready' nets rise upwards in the design hierarchy and contribute to a conjunction over all the resources used in a rule and the request net is a broadcast downwards to all those resources to fire at once. In both the SAFL and Bluespec systems, the downwards nets controls multiplexors that route the arguments to the resources. We defined the Scala DSL extension '`rule`' and extended the Chisel I/O Bundle with suitable methods for registering Bluespec methods. The Chisel 'when' conditional assignments are already semantically correct for Bluespec's conditional execution owing to their parallel composition.

We thought that the introspection features of Chisel (a sugaring of the Java Reflection API) would enable us to explore the abstract syntax tree of the arguments used in a rule and its descendant methods to form the implicit guard. Indeed, Chisel already relies on introspection; for instance, for when it infers the name for registers and other structural resources. However, all of the operators in these expressions are already overloaded by Chisel's DSL to build the hardware circuit as an abstract syntax tree. Chisel provides tree folding walker methods for such trees. So the conjunction of such guards is readily computed and ANDed with the explicit guard of the rule (and the implicit guard of the explicit guard!).

Bluespec supports a number of pragmas that can annotate rules and methods to alter the behaviour of its global scheduller. For instance, it provides two alternate semantics for reverse multiplexing the intrinsic guards of sub-expressions in different branches of a conditional expression: split or combined. Also, Bluespec implements a packing transform on complete rules, running many in the same clock cycle. This follows the same principle as the one we implemented for RTL. Further work on the Bluespec layer is needed to implement these features, but simple designs, like the GCD illustrated work fine without them.

III. Inter-paradigm compatibility

It is interesting to consider what interworking restrictions exist between the various styles we have discussed. Table I summarises the language features so-far mentioned. For maximum expressibility we might consider the enabling of all of these dialects at once. All preserve the structural hardware construction language Chisel in its entirety so that it can continue to be used and mixed at fine grain with the new paradigms. The RTL style introduces the '`always`' block which is the same a Handel-C outermost 'SEQ' block, provided we are happy to enable the packing transform (denoted as 'multiple resolved assignments per clock cycle' in the table). The PAR construct is not incompatible with RTL-style design expression: in Verilog, for instance, all of a sequence of a non-blocking assignments to different left-hand sides are essentially put in parallel and Verilog has a fork/join construct which is synthesisable when the left-hand sides assigned are disjoint over forks (although there is more than one way to interpret the affects of blocking assigns in one branch on the right-hand sides of other branches).

Although Handel-C officially bans global variables, there is no incompatibility when they are present: they just cause RaW hazards as usual. The named channels are no different from statically-instantiated FIFOs and there is no problem if the interface paradigm at one end of a FIFO is different from that at the other.

SAFL's automatic insertion of holding registers to overcome structural hazards is a desirable feature in the purely functional context, but when global variables are present as well, then unpredictable timing changes after a minor edit will tend to resolve races on them differently, as was the case with the combination of Handel-C and global variables. Avoiding global variables and combining Handel-C's channels with SAFL's schedulling and TLM modelling looks like a particularly attractive design style. It would be readily amenable to time/space folding.

Bluespec's TLM calls cannot directly invoke SAFL methods unless the SAFL code always completes in one clock cycle. This is because of Bluespec's insistence that every rule should take at most one clock cycle. For multi-cycle work, Bluespec decouples the 'put' and 'get' operations as separate methods on a structural instance. On the other hand, SAFL code can invoke a Bluespec rule without restriction, provided the handshake

```
//http://csg.csail.mit.edu/6.375
//A simple example Euclid's algorithm for computing
//the Greatest Common Divisor (GCD):
module mkGCD (I_GCD);
  Reg#(int) x <- mkRegU;
  Reg#(int) y <- mkReg(0);

  rule swap ((x > y) &&  (y != 0));
       x <= y;  y <= x;
  endrule

  rule subtract ((x <= y) && (y != 0));
     y <= y - x;
  endrule

  method Action start(int a, int b) if (y==0);
     x <= a;  y <= b;
  endmethod

  method int result() if (y==0);
    return x;
  endmethod
endmodule
```

```
//The same code with minor syntax changes for
//construction on top of Chisel.
class GCDunitBSV extends Module {
  val x = Reg(init=UInt(192, 32)) // 32-bit regs.
  val y = Reg(init=UInt(222, 32)) // With initial work.

  val x = Reg(outType=UInt(32))
  val y = Reg(outType=UInt(32))
  rule ("swap") ((x > y) & (y != UInt(0)))  {
       x := y
       y := x
    }
  rule ("subtract") ((x <= y) & (y != UInt(0)))  {
      y := y - x;
    }
  def gcd_start(arg_a :UInt, arg_b :UInt) =
      WHEN(y === UInt(0)) { x := arg_a;  y := arg_b
    }
  def gcd_result():UInt = WHEN(y === UInt(0)) { x }

  val io  = new BSV_bundle {
     bsvBind_a2v(gcd_start _) // 2 args, void return
     bsvBind_a0(gcd_result _) // 0 args, data return
    }
}
```

Fig. 5: GCD using guarded atomic actions in the original Bluespec (left) and as adapted to be built on Chisel (right).

	RTL	SAFL	Handel-C	Bluespec	HLS
Eternal process loops (always-style)	YES	YES	YES	YES	YES
Shared Variables	YES	no	no	YES	YES
Data-dependent control flow	YES	YES	YES	YES	YES
Channel Msg Passing	no	no	YES	no	YES
Guarded Atomic Rules	no	no	YES	no	YES
Seq/Par Constructs	no	no	YES	no	YES
Multiple resolved assignments per clock cycle	YES	n/a	no	YES	YES
Compile-time Scheduller	no	YES	no	YES	YES

TABLE I: Comparison of Language Features

protocol is adapted: automation of the adaption looks to be relatively easy, triggered by the type system using Scala's implicit methods. But we have not automated it yet. We might also want to import additional, foreign, explicit guards into Bluespec rules and methods to stop Bluespec firing under fairness or safety conditions which it would not if it were controlling a regular Bluespec resource.

We now quickly report on one experiment where all the styles presented were used within a single design. Our implementation allows each Chisel module to use its own style. Interworking between styles is mostly seamless, except where we manually wrote the glue required to call between SAFL and Bluespec. The example system checks that calls to GCD are commutative in their arguments by trying them both ways around. Fig. 7 shows the following components:

1) The GCD computation uses the Bluespec module already described. Owing to the multi-cycle Bluespec computation, the argument and result must be conveyed using separate methods.
2) The GCD component is invoked twice by the 'Check-Commutative' code in the SAFL-style. SAFL function calls are blocking with the 'thread' not returning until the answer is ready. Therefore, our manually-coded 'transactor' shim is required to convert between the blocking SAFL style and the non-blocking Bluespec style. The SAFL-style code itself is written in a parallelisable form, but there is only one instance of the GCD unit provided, so our library invokes it twice in the time domain with the first answer cached in an automatically generated holding register.
3) The work and result collection from the SAFL section is implemented by a behavioural-RTL style that is also blocking and where the thread pauses at multiple STEP points.
4) The results are conveyed to a Handel-C printing process over a blocking Handel-C channel.

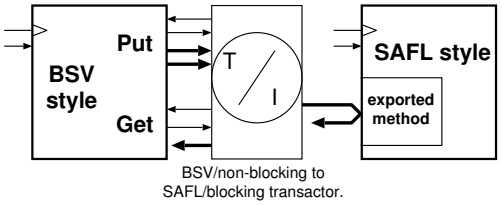

Fig. 8: Typical configuration where a Bluespec component invokes methods in a SAFL-style subsystem.

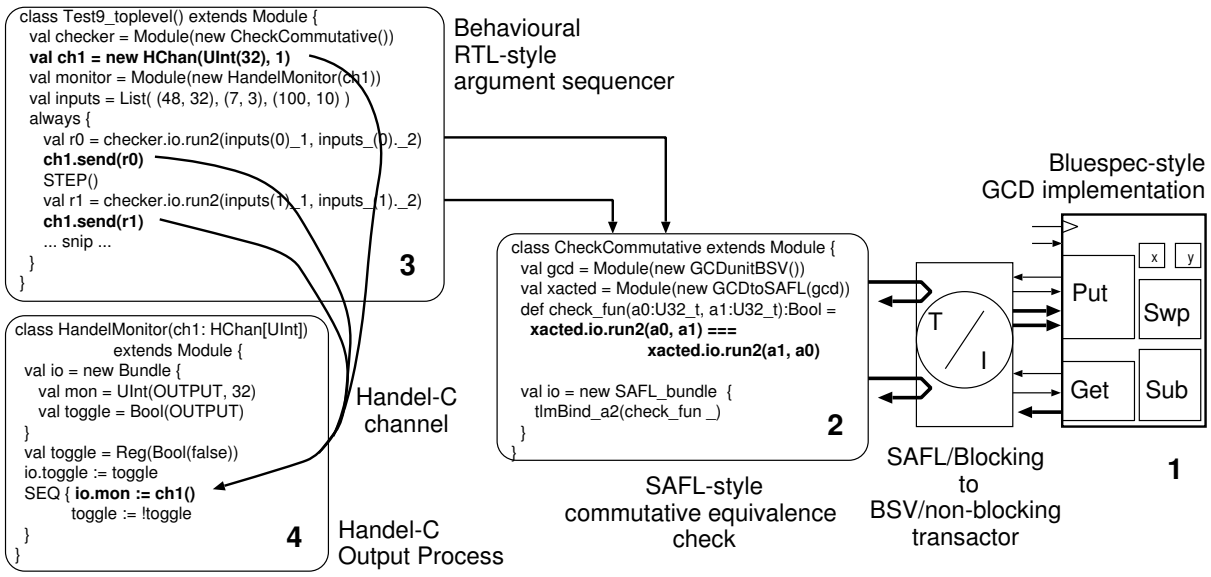

Fig. 7: Interworking demonstration: four styles combined relatively seamlessly, except for SAFL/BSV interworking that requires transactors.

An alternative transactor structure (Fig. 8) would be required to call from Bluespec to SAFL or from Bluespec to a Handel-C style blocking channel. This transactor would provide a non-blocking target pair of put/get methods and it would then initiate a blocking call on the other side. The put method would not be ready if the blocking side is busy and the get method would not be ready until at least one result is queued within the transactor. For Bluespec interworking with subsystems that are known to be non-blocking or that have no result (e.g. a posted write), a single Bluespec-style method is all that is needed.

IV. Conclusion and Further Work

We have shown how various high-level synthesis techniques can be built on top of Chisel and conveniently embedded in Scala alongside Chisel as domain-specific language extensions. Implementing libraries to support these constructs required a deep understanding of Chisel's internal mechanisms, but, in return, the richer forms of expression provided should make end user code easier to write. In both stages, we gained great power by building on a flexible hardware library and embedding as a DSL within a very powerful language like Scala. Specifically, we avoided having to re-implement a parser and we had free access to all the elaboration combinators that lead to great productivity. Re-implementing most of Haskell inside a hardware compiler was a great deal of the work required when implementing a public-domain Bluespec compiler [16].

The SAFL-style function definitions of hardware are elegant owing to being purely functional, yet allow the user or the compiler to flexibly and safely trade execution time for silicon area, but are severely limited in practical applicability owing to the RAM being a very important component in real hardware and the imperative array not being a first-class aspect in pure functional languages.

The manual allocation of work to instances may not be desirable, but the *server farm* paradigm is easily encoded in all of the presented styles. Servers with state are the only obstacle, as always. To support a future optimising scheduller, an estimated latency can also be supplied for the blocking instances.

Our prototype implementations of all of these extensions have not, so far, required any modification to the current Chisel distribution, 2.3; they are just additional libraries. The implementations are available for download on *www.cl.cam.ac.uk/users/djg11/cbgboc*. In further work we could add another main paradigm used for hardware design: synchronous languages such as Esterel and Lustre [17]. Their atomic synchronisation primitive (the 'emit' statement) requires the same interlock infrastructure that we need to add for Bluespec to stop a rule firing more than once. We see no problems with that.

Syntax-directed generation of hardware from high-level constructs tends to suffer from either having too many registers or too few, especially on the control flow arcs. Even with a simple adder, it is well known that ripple carry and full look-ahead are both poor design points. A global re-balancing phase is generally required. Back-end tools perform some degree of D-type migration, but this does not alter the overall number of pipeline stages (even if precise quantity of D-types varies owing to, e.g. being moved from the one output of an AND gate to its two inputs). This would likely be useful for all Chisel users if it provided low-level extensions that tune the level of pipelining being generated, perhaps in an iterative way. The HCL frameworks, like HardCaml and Chisel provide an ideal basis for such an orthogonal development, but a suitable API needs to be proposed.

Of course, whether one really wants to mix styles at a fine grain is a good question.

REFERENCES

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

[2] A. Ray. (2014) HardCaml: a DSL embedded in OCaml for designing and testing RTL hardware designs. [Online]. Available: http://www.ujamjar.com/open-source/ocaml/2014/06/17/hardcaml.html

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," New York, NY, USA, pp. 174–184, 1998. [Online]. Available: http://doi.acm.org/10.1145/289423.289440

[4] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas Lava," in *Proceedings of the Symposium on Implementation and Application of Functional Languages*, ser. LNCS, vol. 6041. Springer-Verlag, Sep 2009.

[5] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson, "From software to accelerators with legup high-level synthesis," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, Sept 2013, pp. 1–9.

[6] S. Singh and D. Greaves, "Kiwi: Synthesis of FPGA circuits from parallel programs," in *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, April 2008, pp. 3–12.

[7] D. Greaves, "The CSYN verilog compiler," in *International Workshop on Field Programmable Logic, FPL'95.*, ser. Lecture Notes in Computer Science, vol. 975, September 1995, pp. 198–207.

[8] A. Mycroft and R. Sharp, "Hardware synthesis using safl and application to processor design," in *Correct Hardware Design and Verification Methods: 11th IFIP WG10.5 Advanced Research Working Conference, CHARME 2001*. Springer Verlag, 2001, p. 2144.

[9] D. R. Ghica, A. Smith, and S. Singh, "Geometry of synthesis iv: Compiling affine recursion into static hardware," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11. New York, NY, USA: ACM, 2011, pp. 221–233. [Online]. Available: http://doi.acm.org/10.1145/2034773.2034805

[10] *Handel-C Language Reference Manual*, Agility, 2007.

[11] D. C. Wood and P. H. Welch, "The Kent retargetable occam compiler," in *WoTUG '96: Proceedings of the 19th world occam and transputer user group technical meeting on Parallel processing developments*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 1996, pp. 143–166. [Online]. Available: http://portal.acm.org/citation.cfm?id=270108

[12] A. W. Roscoe, C. A. R. Hoare, and R. Bird, *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. [Online]. Available: http://portal.acm.org/citation.cfm?id=550448

[13] J. Larson, "Erlang for concurrent programming," New York, NY, USA, pp. 48–56, 2009. [Online]. Available: http://dx.doi.org/10.1145/1467247.1467263

[14] I. Page and W. Luk, "Compiling Occam into field-programmable gate arrays," in *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, W. Moore and W. Luk, Eds. 15 Harcourt Way, Abingdon OX14 1NV, UK: Abingdon EE&CS Books, 1991, pp. 271–283. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.7526

[15] R. Nikhil, "Bluespec SystemVerilog: Efficient, correct RTL from high-level specifications," *Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.

[16] D. J. Greaves. (2014) CBG-BSV toy bluespec compiler. [Online]. Available: http://www.cl.cam.ac.uk/users/djg11/wwwhpr/toy-bluespec-compiler.html

[17] G. Berry, M. Kishinevsky, and S. Singh, "System level design and verification using a synchronous language," *Computer-Aided Design, International Conference on*, vol. 0, pp. 433+, 2003. [Online]. Available: http://dx.doi.org/10.1109/iccad.2003.1257813