

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

Dr David J Greaves

University of Cambridge  
Computer Laboratory



Presented at  
Memocode'15, September 2015, Austin Texas


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


## Talk Topics.

- Mainstream hardware description language features,
- Tagged data and Time/Space folding, re-pipelining,
- HCL Concept: Lava, Chisel + HardCaml,
- Metaprogramming: motivation for clean and more powerful elaboration,
- Four examples of powerful constructs on Chisel,
- Brief discussion of interaction between design styles.

# RTL – Living in the dark age ?

## Verilog and VHDL:

- 
- Successful because they combine structure, behaviour and testbench
  - Remain 'kingpin' between front-end and back-end flow
  - System Verilog a worthwhile step forward.

- 
- No denotation of which data is live,
  - No modern compiler warnings (uninitialised variable ...),
  - No symbol table/bounds checking for layout in RAM,
  - All concurrency is in the programmer's head,
  - No mutex or FIFO primitives,
  - No synthesisable TLM.

# Staged Evaluation aka Metaprogramming

- Part of the program runs at 'compile time' – the *elaborate phase*.
- The elaborated program consists of a hardware circuit that runs at run time: the *execution phase*.

*For example, Verilog and VHDL have generate statements and generate variables which disappear during the first stage.*

# RTL – Goodie 1 – Assignment Packing

*Elaborates the denotational semantics at compile time.*

An (order-sensitive) imperative program

```
if (e1) foo = foo+1;  
bar = foo + 2;  
foo = 3;
```

converts to “pure RTL” (unordered list per clock domain)

```
bar <= (e1 ? foo+1:foo) + 2;  
foo <= 3;
```

- *But can explode complexity when complex array subscript comparison.*

# RTL – Goodie 2 – State machine PC inference.

```
always @(posedge clk) begin
    foo <= foo + 1;
    if (e1) @(posedge clk)
        foo <= foo + 2;
end
```

converts to “pure RTL”

```
pc <= e1 & !pc;
foo <= foo + (pc ? 2: 1);
```

*However state machine synthesis banned in many house styles and later synthesisable subset definitions.*

# Hardware Construction Language

A (rich) language for writing programs that prints out a circuit diagram.

But does it:

- Ensure syntactically well-formed output ?
- Semantically well-formed as well ? E.g. no two outputs wired together?
- Include simple optimisations?
  - Discarding disconnected logic,
  - Constant propagation and identity folding:

Eg:  $\text{exp} \ \&\& \ \text{true} \rightarrow \text{exp}$

# Lava HCL

## 'Lava Hardware Design in Haskell'

Make use of all standard combinators such as Fold, Map and Zip.

Different instantiations of the leaf nodes for

- Simulation
- Synthesis
- Verification

Bjesse, Koen, Sheeran, Singh 1998

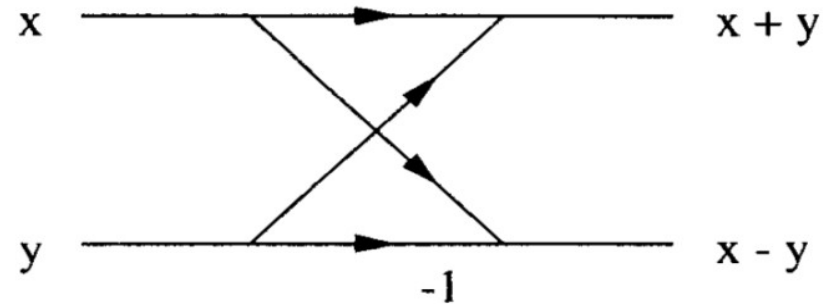


Figure 9: A butterfly

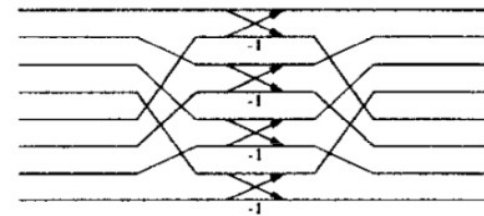


Figure 10: A butterfly stage of size 8 expressed with riffing

```
bfly :: CmplxArithmetic m
      => [CmplxSig] -> m [CmplxSig]
bfly [i1, i2] =
  do o1 <- csubtract (i1, i2)
     o2 <- cplus (i1, i2)
     return [o1, o2]

bflys :: CmplxArithmetic m
       => Int -> [CmplxSig] -> m [CmplxSig]
bflys n =
  riffle >-> raised n two bfly >-> unriffle
```



# Data-Dependent Control Flow ?

The 'if' statement is part of any programming language, but how much conditional execution does our language support at run time ?

- Lava's elaborate phase is very rich, it certainly contains 'if' statements.
- But all run-time conditional flow was through explicitly printed multiplexors.

*Generally we desire greater expressivity than that...*

# Time/Space Flexibility

- We would like to use one entry of the design for either:
  - Fast execution using a lot of hardware
  - Slower execution using less hardware
- We should favour languages that are amenable to rapidly changing between these styles,
- while still being '*resource aware*' – engineers understand roughly how many gates they are using as they write each line.

*Associative assignments such as += are good (amenable).*

*Functional programs are very good!*

*VLSI trends increasingly want layout-time re-pipelining.*

# SAFL - Statically Allocated Functional Language

Used a variant of ML to describe hardware

- We see powerful combinators for hardware generation
- The ML 'if' is the run-time 'if' (*could not be a DSL*)
- All recursion is tail recursion, hence bounded stack space – finite state.
- But functional style did not fit comfortably with RAMs

SAFL appeared in ICALP 2000. Alan Mycroft, Richard Sharp.

# SAFL – Resource Awareness

Baseline rules control the amount of hardware generated:

1. Leaf operators occurring syntactically are freshly instantiated in the hardware *for each syntactic occurrence* in the source code.
2. The same goes for function definitions, which means function applications of a named function are *serialised* with argument and return value multiplexors.

This contrasts with High-Level Synthesis (HLS) where the designer perhaps only broadly constrains how many ALUs and RAMs to use, but the amount of random logic is unpredictable...

# Time/Space Folding in SAFL

```
fun cmult x y =  
  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.
```

A function replicator, such as **UF**, enables control of time/space folding, giving a fresh copy of a function.

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

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

*Server farms etc are also easy to provide provided everything remains stateless.*

# Chisel HCL (from UCB)

- Chisel is embedded as a DSL in Scala.
- Scala is a wonderful language
  - A superb mix of functional, imperative and OO

Scala has flexible overloading syntax that makes extensions and implicit conversions simple to deploy.

- Chisel provides all the main basic gates and memories and *powerful wiring up primitives* but not much data-dependent control flow (no imperative flow or pc inference).
- *Scala allows us to build up on top easily.*

# A Varadic Priority Arbiter in Chisel

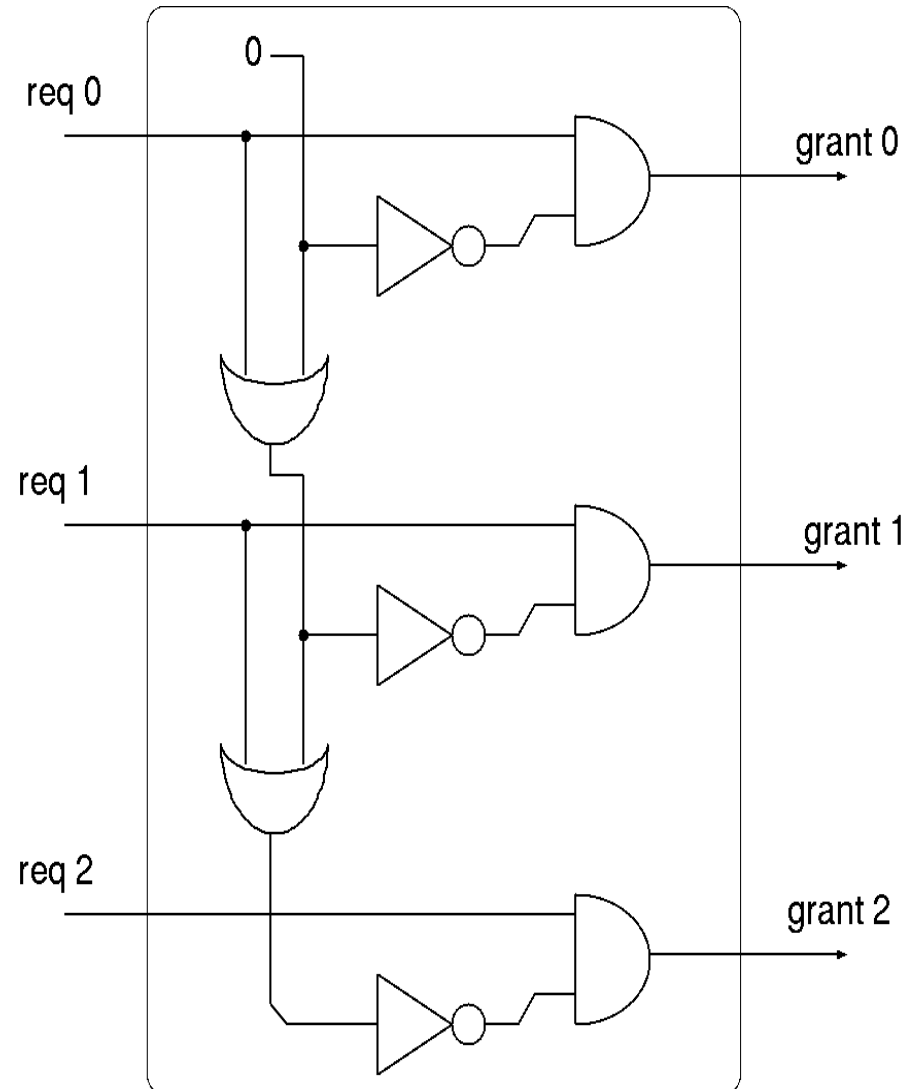
```
class genPriEncoder(n_inputs : Int) extends Module
```

```
{  
  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  
    }  
  }  
}
```

H/W components extend Module.

They do their I/O via a Bundle.

All the standard operators & | ! are overloaded for h/w generation.



# Run-time 'if' in Chisel

```
class Parity extends Module {  
  val io = new Bundle {  
    val in  = Bool(dir = INPUT)  
    val out = Bool(dir = OUTPUT) }  
  val s_even :: s_odd :: Nil = Enum(UInt(), 2)  
  val state = Reg(init = s_even)  
  when (io.in) {  
    when (state === s_even) { state := s_odd }  
    when (state === s_odd)  { state := s_even }  
  }  
  io.out := (state === s_odd)  
}
```

*The 'when' key word is Chisel's main run-time IF operator, but there are other variants including a switch/case statement.*

*The === operator is used so that Scala's == remains usable.*



# Adding TLM to Chisel

We store function entry points in the I/O bundle

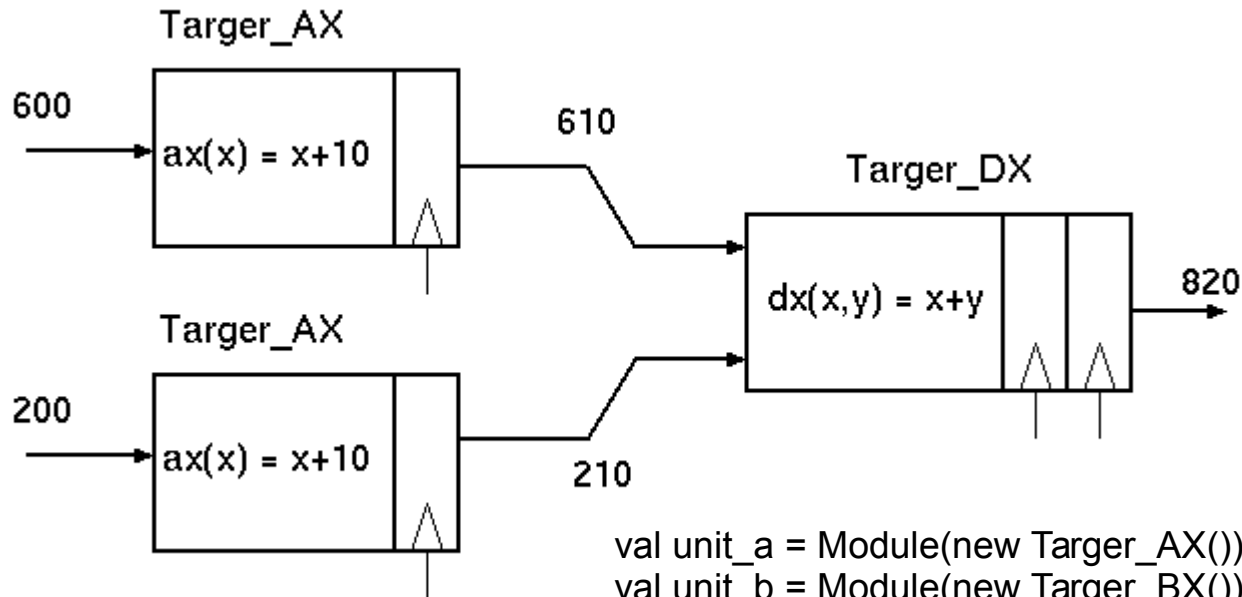
Each function is annotated with its fixed pipeline delay or else can use handshake nets Request/Valid (not shown here).

```
class Targer_AX extends Module
{
  val io = new TLM_bundle_Ic
  { // Register TLM callable function with one pipeline
    delay.
    tlmBind_a1(ax_fun __, 1)
  }
  def ax_fun(x:UInt) = Reg(UInt(32), x + UInt(10))
}

class Targer_DX extends Module
{
  val io = new TLM_bundle_Ic
  { // TLM callable diadic function with 2 pipeline delays.
    tlmBind_a2(dx_fun __, 2)
  }
  def dx_fun(x:UInt, y:UInt)= Reg(Reg(UInt(32), x + y))
}
```

TLM = Transaction Level Modelling – although here we are not modelling, but doing.

# TLM in Chisel (2)



```
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 answer = unit_d.io.run2(unit_a.io.run1(arg1K), unit_b.io.run1(arg2K))
```

```
// Diadic - reuse of same component AX
```

```
val answer = unit_d.io.run2(unit_a.io.run1(arg1K), unit_a.io.run1(arg2K))
```

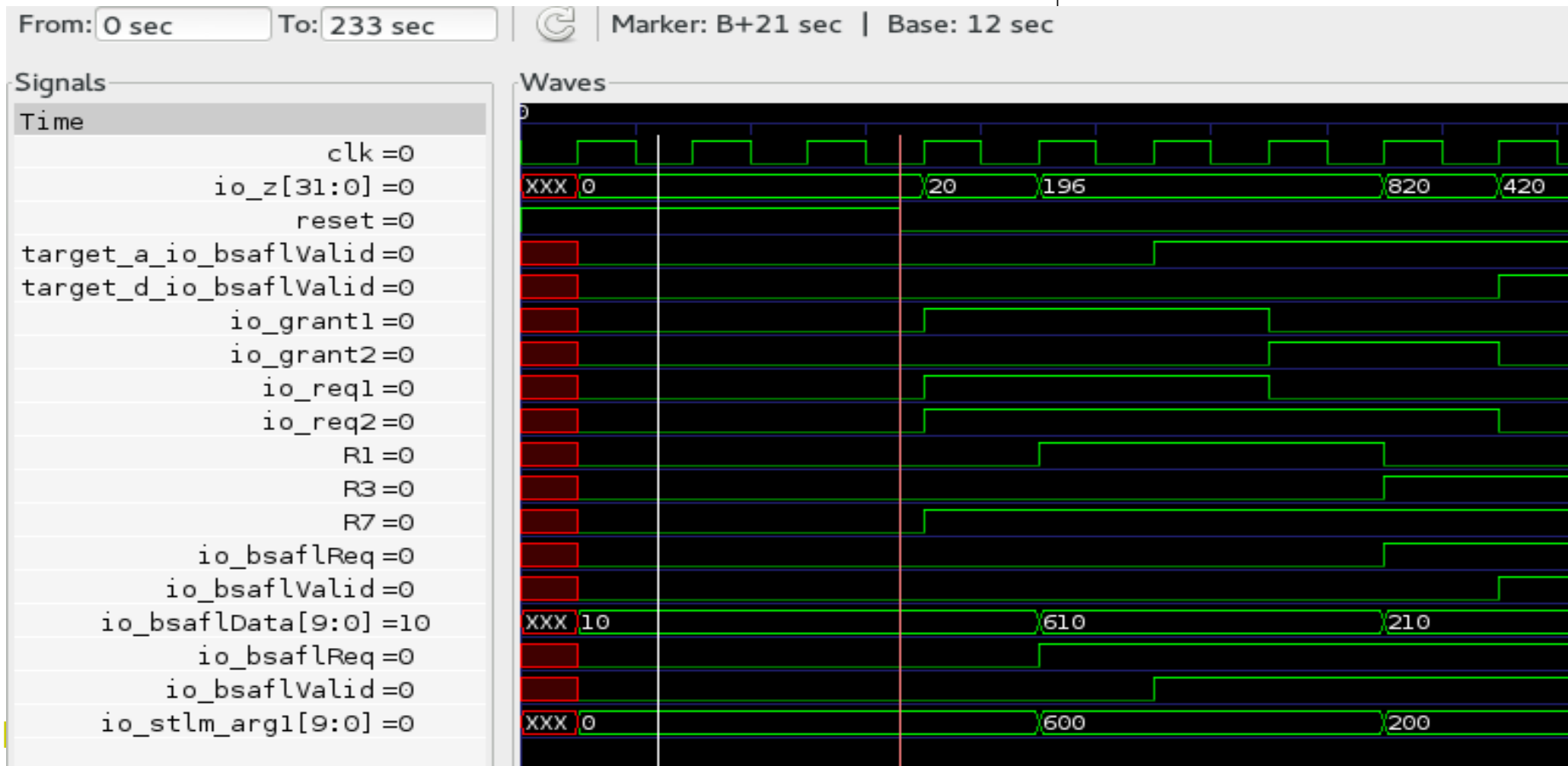
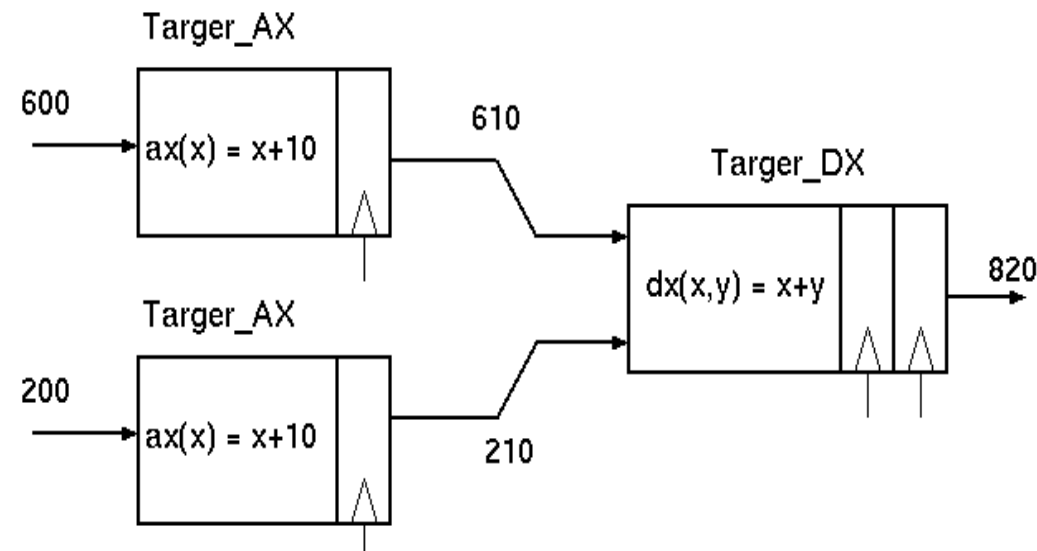
```
// Invoke and downconvert to unguarded for rest of design
```

```
val answer1 = SAFLImplicitx.ex_drop_chisel_data_from_guarded(answer)
```

```
io.z := answer1
```

```
io.v := answer.isValid()
```

# Running the TLM Example with SAFL semantics



# HardCaml

ML is the perhaps the best-known functional language.

ML + Objects + Better syntax + more advanced types = OCAML

OCAML is the ultimate programming language ?

(Well some think so - Mirage operating system is an OCAML linux kernel. I'm beginning to prefer Scala ...)

HardCaml: *An open-source domain specific language embedded in OCaml for designing and testing register transfer level hardware designs. --- The HardCaml library provides an API roughly consistent with the structural subset of VHDL and Verilog.*

Also: has a snazzy front end embedded in Javascript.

# HardCaml Small Example

```
/* Verilog counter */  
module counter  
  #(parameter bits = 8)  
  (  
    input clock, clear, enable,  
    output reg [bits-1:0] q  
  );  
  
  always @(posedge clock)  
    if (clear) q <= 0;  
    else if (enable) q <= q + 1;  
endmodule
```

```
(* HardCaml counter *)  
let q = reg_fb r_sync enable bits (fun d -> d +: 1)
```

# Rule-based hardware generation (Bluespec)

- Recently Bluespec System Verilog has successfully raised the level of abstraction in RTL design:
- A Bluespec design is expressed as a list of declarative rules that fire atomically and which last less than one clock cycle,
- Shared variables are mostly replaced with one-place FIFO buffers with automatic handshaking,
- Rules are allocated a static schedule at compile time and some that can never fire are reported,.
- The wiring pattern of the whole design is elaborated using a powerful embedded functional language (as per Lava).

# Bluespec: Background + Example

```
module mkTb (Empty);
```

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

```
  rule countup (x < 30);
```

```
    int y = x + 1;
```

```
    x <= x + 1;
```

```
    $display ("x = %0d, y = %0d", x, y);
```

```
  endrule
```

```
  rule done (x >= 30);
```

```
    $finish (0);
```

```
  endrule
```

```
endmodule: mkTb
```

*But, imperative expression using a conceptual thread is also useful to have, so Bluespec has a behavioural sub-language compiler built in.*

# Bluespec: Background + Example (2)

```
module mkTb (Empty);
```

```
  Reg#(int) xx  <- mkReg ('h10);  
  GCD_ifc pipe <- mkGCD;
```

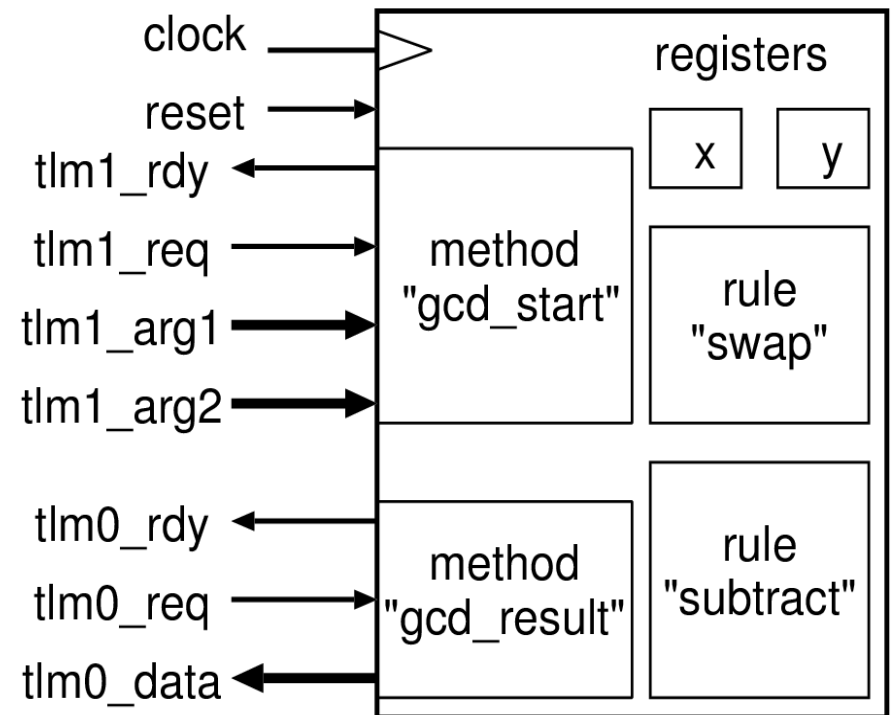
```
  rule sendwork;  
    dut.gcd_start (xx, 40);  
    xx <= xx + 'h10;  
  endrule
```

```
  rule drain;  
    let y = dut.gcd_result();  
    $display ("    y = %0h", y);  
    if (y > 'h80) $finish(0);  
  endrule
```

```
endmodule
```

Bluespec uses interface definitions imported by both caller and callee with an established mapping to H/W.

```
interface GCD_ifc;  
  method Action start(int a, int b);  
  method int result();  
endinterface
```





# GCD in Chisel

*Top is a standard example  
from the Chisel tutorial  
material.*

*Bottom is my SAFL TLM  
binding thereof.*

*This makes it  
nicely callable.*

```
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  
}  
}
```

```
class GCDunit extends Module {  
  val io = new TLM_bundle {  
    tlmBind_a2(gcd_fun _, -1)  
  }  
}
```

# Laying Bluespec on Chisel

(Bluespec left, Chisel-Bluespec right)

```
//http://csg.csail.mit.edu/6.375
//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

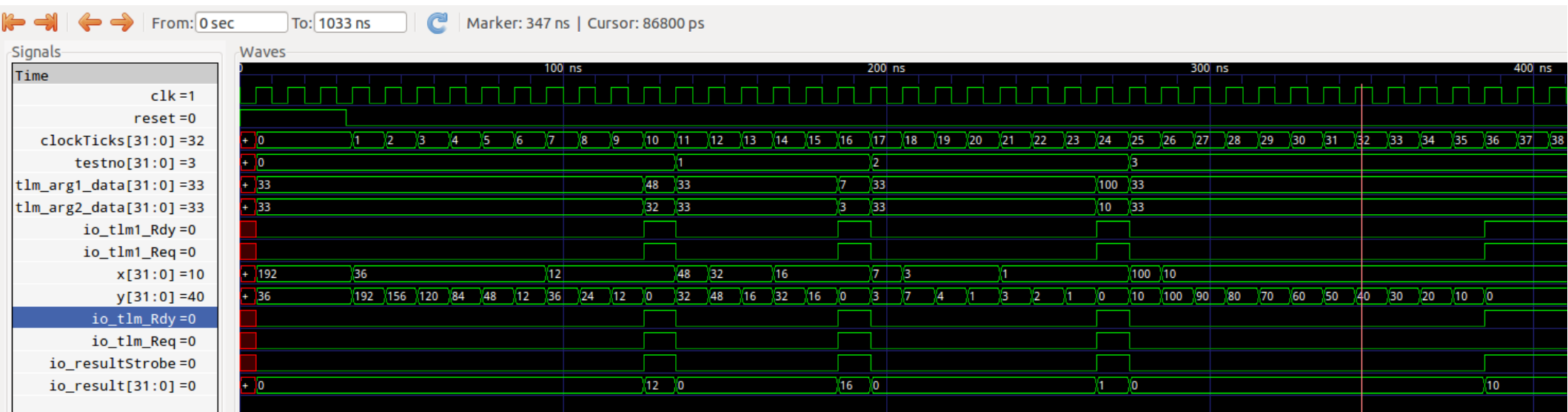
  method int result() if (y==0);
    return x;
  endmethod
endmodule
```

```
//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
  }
}
```

# Bluespec on Chisel: Three GCD runs.



- So, it works!
- Scala/Chisel gave us all the Bluespec elaboration combinators, and
- The rule/method core h/w generation is relatively simple to lay on Chisel.

# Handel-C

Handel-C uses explicit Occam/CSP-like channels  
(**'!' to write, '?' to read**):

<b>// Generator (src)</b>	<b>// Processor</b>	<b>// Consumer (sink)</b>
<b>while (1)</b>	<b>while(1)</b>	<b>while(1)</b>
<b>{</b>	<b>{</b>	<b>{</b>
<b>ch1 ! (x);</b>	<b>ch2 ! (ch1? + 2)</b>	<b>\$display(ch2?);</b>
<b>x += 3;</b>	<b>}</b>	<b>}</b>
<b>}</b>		

Using channels makes concurrency explicit and allows synthesis to re-time the design.

Banning shared variables avoids RaW and WaW hazards.

Handshaking wires within a synthesis unit may disappear during compilation if they would have constant values owing to certain components being always ready.

# Laying Handel-C on Chisel

```
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!  
  }  
}
```

Handel-C also supports  
SEQ and PAR block keywords  
where imperative commands  
enclosed run sequentially  
Or in parallel.

We use our RTL always for the  
outermost SEQ.

# Laying RTL on Chisel

```
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); }  
  }  
}
```

RTL primitives added:

- state machine inference,
- run time control flow IF statements,
- assignment packing.

# Interoperation Issues ?

All fragments show are freely embedded in general Scala source code.

The libraries for all coding styles are enabled all at once.

All Chisel features are also remain available without change.

Q1. Are there any restrictions on how we intermix these design styles?

A1. Hardly any (see next slide).

Q2. Is it a good idea to freely mix them ?

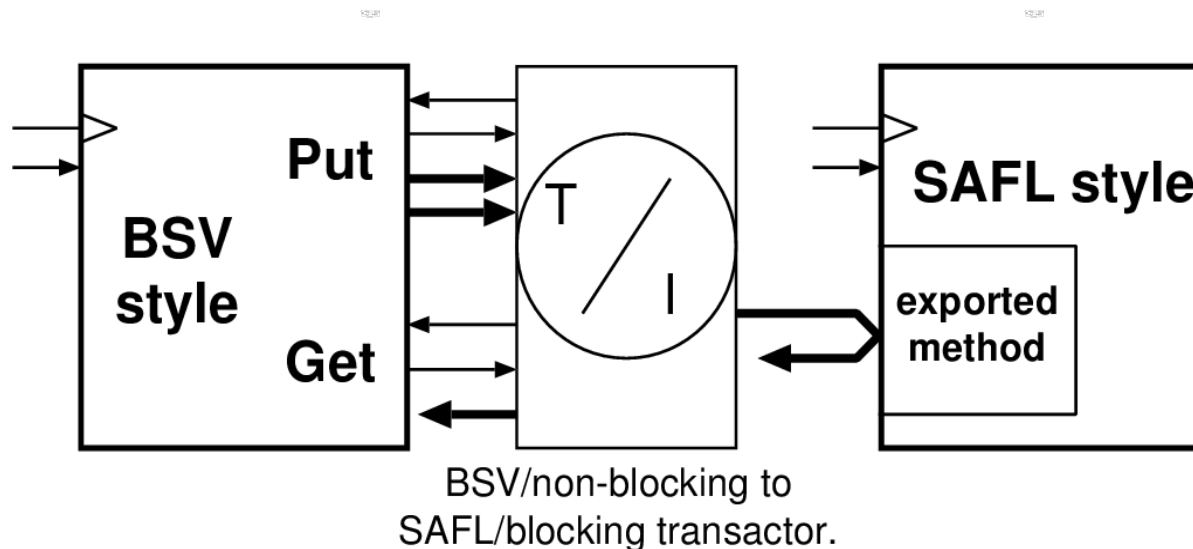
A2a. Horses for courses ?

A2b. *Probably not!*

# BSV to SAFL Interworking

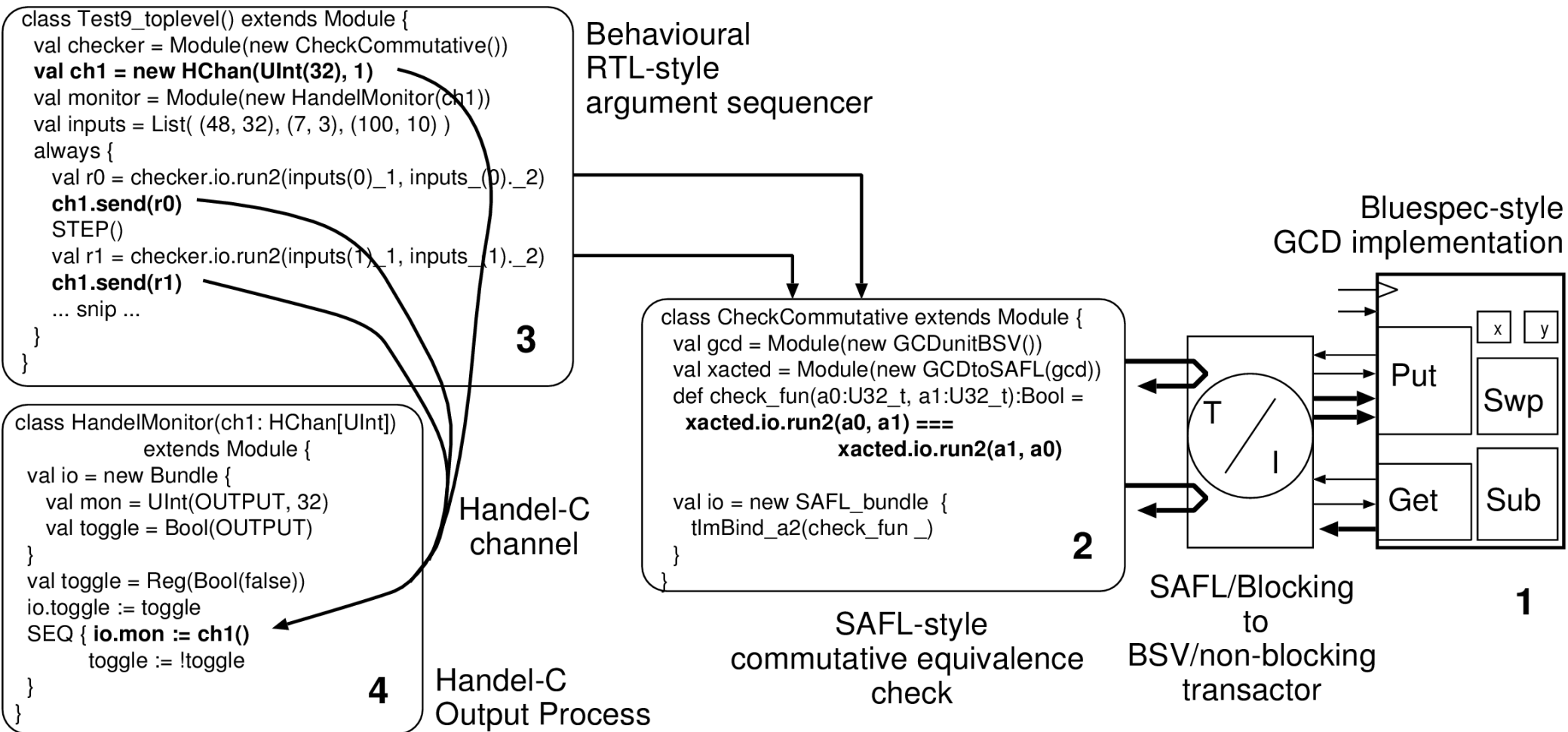
*Bluespec insists that all rules and methods take at most one clock cycle to execute.*

- SAFL can freely call Bluespec methods, but
- Bluespec may need a transactor to call SAFL.





# Combining all four design styles



- 3. State machine stimulus generation
- 1. Bluespec GCD unit
- 4. Handel-C output logging channel.
- 2. Invoked twice from SAFL via a transactor.

# Summary Features

	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 Scheduler	no	YES	no	YES	YES

TABLE I: Comparison of Language Features

# Do we need HCL's ?

- The Chisel and HardCaml baselines are fairly simple yet provide all the 'structural' resources for emitting validated netlists and cycle-accurate simulation.
- Yet they leverage the full power of their parent language for elaboration.
- They provide interworking with RTL designs in Verilog and VHDL.
- They provide a 'power platform' for supporting your own favourite expression style ...

# Conclusions and Views

- Functional elaboration language gives expressivity and supports folding – rich and modern.
- People vary in the expression form they prefer.
- Future hardware languages will be richer, support concurrency better and amenable to repipelining post synthesis.
- Future styles will perhaps be more explicit on state edges and support associative assignments.
- ...

# Thankyou for you attention.

## Open Source Links

- Chisel: <https://chisel.eecs.berkeley.edu>
- HARDCAML:  
[www.ujamjar.com/open-source/ocaml/2014/06/17/hardcaml.html](http://www.ujamjar.com/open-source/ocaml/2014/06/17/hardcaml.html)
- Build on chisel <http://www.cl.cam.ac.uk/users/djg11/cbgboc>
- Kiwi HLS from C#: <http://www.cl.cam.ac.uk/~djg11/kiwi>
- Toy Bluespec: [www.cl.cam.ac.uk/~djg11/wwwhpr/toy-bluespec-compiler.html](http://www.cl.cam.ac.uk/~djg11/wwwhpr/toy-bluespec-compiler.html)

**Backup Slides Follow**

# C-to-Gates: Classical HLS

Take one thread and a body of code:

generate a custom datapath containing registers, RAMs and ALUs  
and a custom sequencer that implements an efficient, static schedule  
that achieves the same behaviour.

Creates a precise schedule of addresses on register file and RAM ports  
and ALU function codes.

Typically unwinds inner loops by some factor.

All current EDA/FPGA vendors now support C++ to gates.

Leading free tool is LegUp from U Toronto.

Profiling or datapath description hints are needed for a sensible datapath  
structure since sequencer states are not equiprobable and we do not want  
to deploy resource on seldom-used data paths.

# C-to-Gates: Classical HLS

For example, best mapping of the record fields x and y to RAMs is different in the two foreach loops:

```
class IntPair
{
    public bool c;    public int x, y;
}

IntPair [] ipairs = new IntPair [1024];

void customer(bool qcond)
{
    int sum1 = 0, sum2 = 0;
    if (qcond) then foreach (IntPair pp in ipairs)
    {
        sum1 += pp.x + pp.y;
    }
    else foreach (IntPair pp in ipairs)
    {
        sum2 += pp.c ? pp.y: pp.x;
    }
    ...
}
```

The fields x and y could be kept in separate RAMs or a common one. If qcond rarely holds then a common RAM will serve since there is little contention. Whereas if qcond holds most of the time then keeping x and y in separate RAMs will boost performance.



# What is emitted by elaborate ?

- Gates, wires, flip-flops and RAMs (all).
- TLM arg mux and body mutexes (Bluespec + SAFL).
- Channel FIFOs (Handel-C).
- Current state PC register + logic enable decoder (RTL + Handel-C).