Further Multi-cycle and Sub-cycle Schedulling for Bluespec

Dr David J Greaves

University of Cambridge
Computer Laboratory

Memocode 2019, UCSD, La Jolla, San Diego

Bluespec – Elaboration + Rules

- In the last decade Bluespec has received attention and raised the level of abstraction available for RTL design.
- A structural elaboration language creates a flat collection of parameterised FU instances.
- A flat list of exported methods and declarative rules is generated as well.
- Exported methods are for external invocation.
- Rules:
  - Fire atomically,
  - Fire at most once per clock cycle, and
  - Rule duration is less than one clock cycle.
- Rules are allocated a static schedule at compile time and some that can never fire are reported.

Baseline Bluespec: Further Details

- The synthesised scheduller is stateless.
- The mapping of operations to FU instances is manual.
- Syntactic sugar goodies:
  - An embedded FSM sub-language with SEQ/PAR constructs expands to regular rules,
  - rr <= rr + 1 expands to rr.write(rr.read() + 1).

- The commercial compiler must be licensed.
- In 2012 DJG wrote an open-source `toy' compiler.

Talk Outline

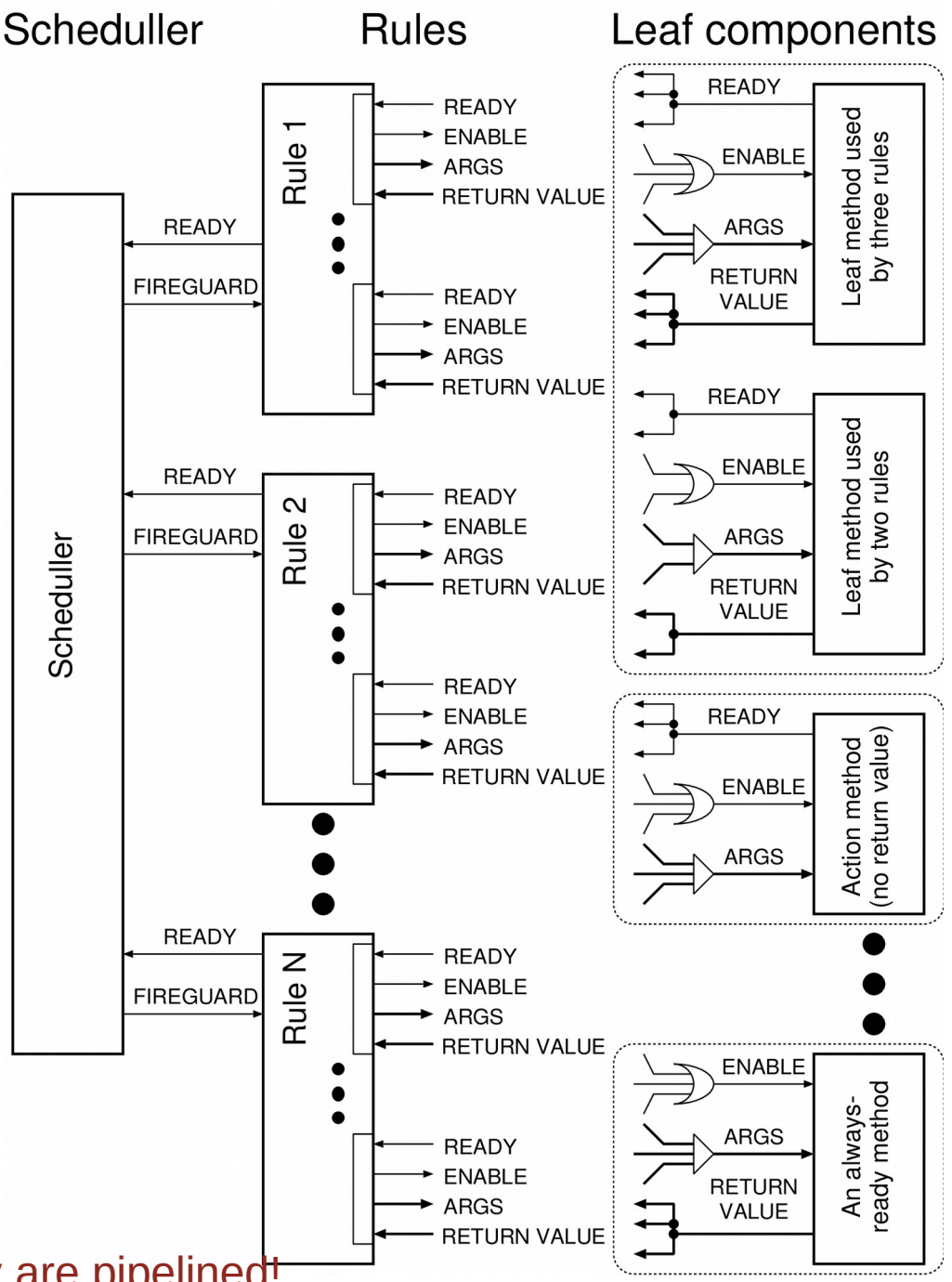
1) Tiny overview of how a Bluespec compiler works,
2) Report on several new fundamental extensions,
3) Mention of ongoing development.

You might then:

1) Download and try yourself,
2) Join the development repo,
3) Help prove correctness when all the semantic extensions are deployed at once!

Baseline Synthesis Scenario

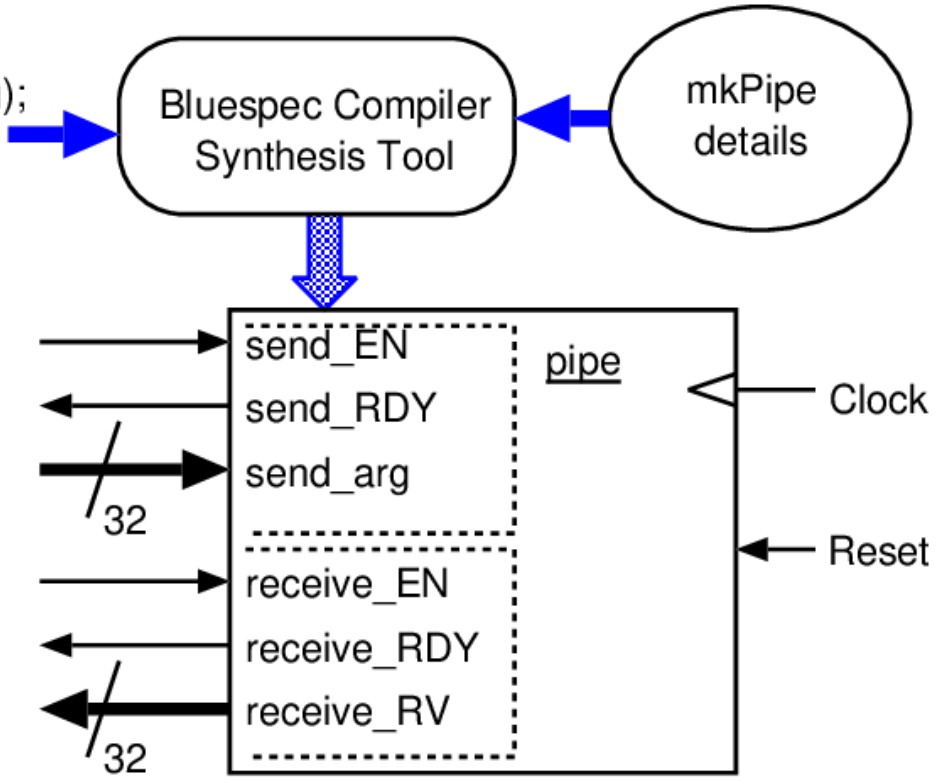
- Leaf component methods are shared by rules
  - Enable is disjunction
  - Arg sources are muxed
- Schedulling conflict if two rules try to use a method at once
- Synthesised scheduller avoids conflicts.



Notes:
- leaf methods are non-pipelined,
- dotted boundaries show an FU,
- many FUs, (RAMs, ALUs) typically are pipelined!

Basic High and Low-level Views

```
interface Pipe_ifc;
   method Action send(int arg);
   method int receive();
endinterface
```




```
module mkTb (Empty);

  Reg#(int) x <- mkReg ('h10);
  Pipe_ifc  pipe <- mkPipe;

  rule fill;
    pipe.send (x);
    x <= x + 'h10;
  endrule

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

We have a TLM-style invocation of each method of an FU.
This is non-blocking TLM.

Each TLM call must be complete within a clock cycle.
(The put/get interface type class nicely encapsulates the pair.
Ideal for FIFOs. Not ideal for pipelined operators...)

Memocode'19 Bluespec Enhancements

Available for download at the moment:

- Automatic instantiation of run-time arbiters for fairness,
- Multiple invocations of certain Action Methods (especially updates to registers) per clock cycle,
- Forwarding paths for efficient and easy use of pipelined operators,
- Multiple firing of rules within a clock cycle.

Ongoing work

- Dynamic binding of operations to stateless FUs,
- Rule fire count and arbiter shares set from a target firing rate.

A First Bluespec Shortcoming

- The standard scheduller is stateless.
- It *does not* generate multi-cycle schedules.
- It *does* report definite rule starvation.
- Arbitration mechanisms must be engineered manually.
- No global analysis or report of likely relative/abs firing rates.

Rule Starvation Example

```
interface BarFace;
   method Action orderDrink(int which, int no);
endinterface
```

```
module mkBarTender(BarFace);

  Reg#(Bit#(10)) beerdrink <- mkReg(20);
  Reg#(Bit#(10)) winedrink <- mkReg(12);

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

```
module mkTest1iBench();

  BarFace fbar <- mkBarTender();

  rule drinkBeer;
      fbar.orderDrink(1, 2);
  endrule

  rule drinkWine if (True);
      fbar.orderDrink(2, 10);
  endrule

endmodule
```

One rule would normally be starved under separate compilation. (If both halves compiled together, orderDrink would not be an external method and would not present a structural hazard).

Example Output (with & w/o arbiter).

Baseline compiler behaviour

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
Beer is 28 and wine is 12
...

Enhanced compiler behaviour

New simulation output:

Beer is 22 and wine is 12
Beer is 24 and wine is 12
Beer is 24 and wine is 22
Beer is 26 and wine is 22
Beer is 26 and wine is 32
Beer is 28 and wine is 32
Beer is 28 and wine is 42
Beer is 30 and wine is 42
...

An arbiter has been instantiated providing fairness between the rules.

Style of arbiter and number of shares per rule is currently by manual pragma.

```
assign Test1i_mkTest1iBench_drinkBeer_FIRE = RST_N && Test1i_mkTest1iBench_fbar_orderDrink_RDY &&
                    (32'sd0==ARXshedtree1_10);
assign Test1i_mkTest1iBench_drinkWine_FIRE = RST_N && Test1i_mkTest1iBench_fbar_orderDrink_RDY &&
                    (32'h1/*1:AUTB12*/==ARXshedtree1_10);
```

Further Bluespec Shortcomings

- Pipelined FUs are accessed via put/get interface.
- No support for infix use of pipelined operators.
  - Consider HLS of
    - foreach (i in 0..9) { ss += i * A[i] }
    - We typically need a synchronous array read and a pipelined multiplier.

- BSV requires named FUs and has no automatic load balancing over anonymous, stateless FU instances.
- Moreover, pipelined operators must have an output FIFO instantiated in case get() method is invoked late.

Solution: Specific FU Migration to Compiler Core

Certain functions and classes of method already call have support or are entirely hard-coded in the standard compiler instead of operating on regular leaf FUs, eg:

- Built-in functions (eg. sizeOf, $bitstoreal(), $finish),
- Combinational Wires (eg. PulseWire, Rwire).

We extend the categories to include:

- Multi-updateable (ephemeral history) registers,
- Synchronous read, stafeful components (RAMs, register files),
- Multiple writes to one location (per port) of such an SRAM,
- Anonymous, stateless, latency=1 ALU operations.

Register Modelling within Compiler Core

From the paper:

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

We use the same technique used in Verilog logic synthesis where expressions are evaluated in a *committed assignment environment* and pending writes are copied there at end of rule.

Paper has details: σ_c is the committed updates, and σ_p is the un-committed (pending write) assigns/updates/writes.

Multiple Action Method Invocations Notes

Commercial compiler has this behaviour for registers flagged as `ephemeral'.

In our approach, the multiple invocation can be applied to any FU who's behaviour can be modelled inside the main body of the compiler up to the extent that there is sufficient hardware write-back bandwidth to flush to real hardware at end of cycle.

For instance, it works with our SRAM extension (next section) provided the number of concurrently written addresses is statically determinable to be no greater than the number of write ports on the SRAM.

In the future it can be applied to FIFOs that have a multi-word broadside interface where queue or dequeue of two or more words per clock cycle is supported.

(We also have an extension where writes of the same value by different rules to the same location or RAM location do not conflict. In general we mark up Action Methods of idempotent or not … cannot go into detail here)

Multiple Action Method Invocations (eg. register writes) in One Clock Cycle.

```
module mkTest1f2();

    Reg#(Bit#(10)) vodka <- mkReg(30);
    Reg#(Bool) grd <- mkReg(0);

    rule test_1f_inc1 if (True);
      vodka <= vodka + 1;
    endrule

    rule test_1f_inc3 if (grd);
      vodka <= vodka + 3;
    endrule

    rule shower if (True);
      grd <= !grd;
      $display("Vodka is %1d", vodka);
    endrule
endmodule
```

This code fragment has two rules writing to one register.

Normal scheduller will assign the inc3 rule higher priority than inc1 since it has a tighter guard condition:

Vodka is 31
Vodka is 34
Vodka is 35
Vodka is 38
Vodka is 39
...

Enhanced compiler overcomes write conflict on shared variable with both updates committing where possible:

Vodka is 31
Vodka is 35
Vodka is 36
Vodka is 40
Vodka is 41
...

Super-scalar Rule Firing

Standard semantic is a rule fires at most once per clock cycle.

Ability to write a register more than once or do two FIFO reads/writes on a single FIFO within a clock cycle makes super-scalar rule firing attractive.

Currently we manually attach rule repeat count to a rule using a pragma, but are exploring replication that aims to hit rule firing rate targets.

Compiler essentially copies out the parsed rule (or group of rules with wire resets) as though macro-expanded in a preprocessor.

Super-scalar SimpleProcessor (1)

SimpleProcessor is a standard Bluespec demo.

The instruction set looks very nice in Bluespec, using its tagged union and associative structure constructs.

```
// ---- Instructions
typedef union tagged {
   struct { RegNum rd; Value v; }                MovI;    // Move Immediate
   InstructionAddress                          Br;      // Branch Unconditionally
   struct { RegNum rs; InstructionAddress dest; }  Brz;     // Branch if zero
   struct { RegNum rd; RegNum rs1; RegNum rs2; }   Gt;      // rd <= (rs1 > rs2)
   struct { RegNum rd; RegNum rs1; RegNum rs2; }   Minus;   // rd <= (rs1 - rs2)
   RegNum                                      Output;
   void                                      Halt;
} Instruction deriving (Bits);
```

The implementation is a simple pattern match against these forms.

Super-scalar SimpleProcessor (2)

SimpleProcessor uses simple register files for I and D memory.

Hence easy to go super-scalar by repeating the fetch_and_execute rule.

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.**

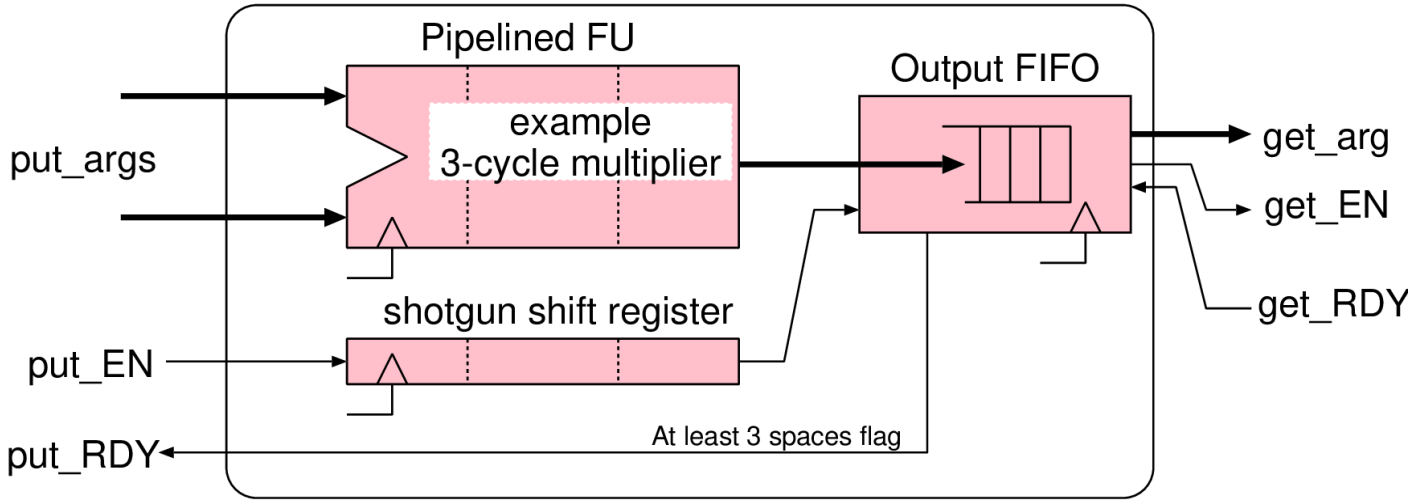
Sadly, overall performance goes down on this example – but useful in general we hope!

Forwarding Paths: Motivation 1

Put/Get is not always a good interface for BRAMs or fully-pipelined FUs.

Consider a 3-cycle multiplier with II=1.

Wrapping up a pipelined FU inside Bluespec Put/Get interface.
(Requires an output FIFO of sufficient depth.)



Without clock-enable, an additional 3-entry FIFO is needed.
With clock-enable, one-entry FIFO is needed.

Forwarding Paths: Motivation 2

Consider the following RTL style code in a rule/method body, where A and B are arrays held in different, latency=1, SSRAMs:

```
if (gg)  pp <= A[qq] ^ B[rr]
ss <= pp + ss
```

Observations:

- The values read are minimally processed before storing in pp.
- The value in pp will at earliest be used in the next clock cycle if code runs in the next cycle, otherwise later still.
- Value forwarding can be used so that readers of pp sometimes use the XOR of the read busses of the two SSRAMs instead of pp contents.

Term `minimally processed' can relate to anything that is `free' in FPGA. This is any logic function (identity, bit-range-select, AND, OR, XOR, NOT).

Forwarding Paths: Implementation

1) A static scan of rule/method bodies after all static elaboration finds:

   - Array subscriptions of SSRAMs (stateful)

   - Pipelined stateless operators with latency=1

2) Pattern matching checks that results are `minimally processed' and then stored in a register.

3) A score-board flip-flop is instantiated to name and record activation of the associated forwarding path.

4) Values to be written in next cycle, guarded by such flops, are simply always pre-added to the committed updates environment.

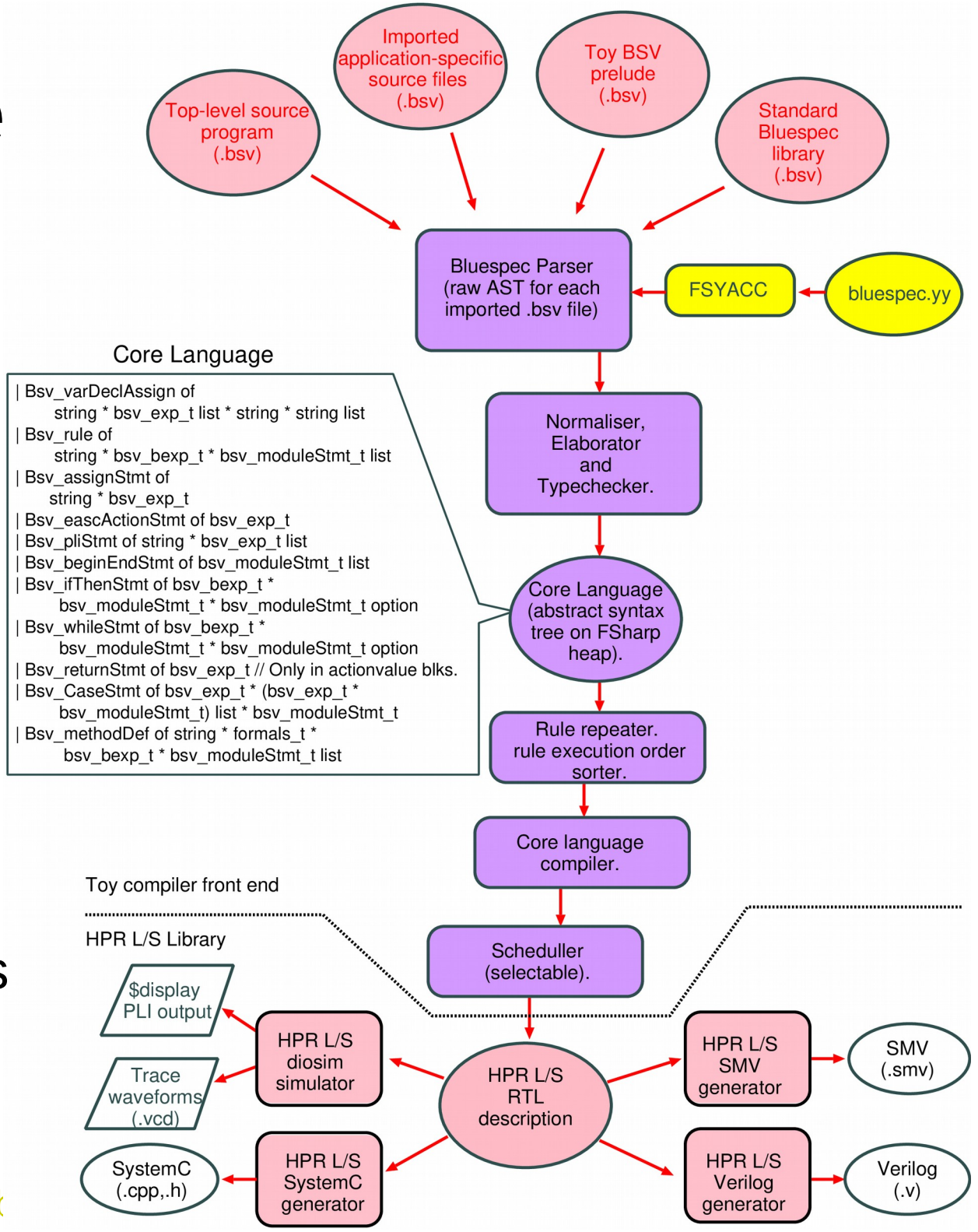
5) As mentioned before, readers read values from committed updates in preference to real FU result bus.

Consequences:

- Multiple forwarded and non-forwarded expressions are storable in a single register,
- Each result that needs forwarding can be stored in more than one register and with arbitrary surrounding control flow.

Open-Source Bluespec Compiler

- Called ‘Toy Compiler’
- Uses HPR L/S framework
- Written in F#.
- Has significant language coverage.
- Hundreds of downloads since 2012.
- Basis for today’s work.



Conclusions

- Various mechanisms for multi-cycle schedulling demonstrated on open-source toy compiler.
- Support for more natural use of certain infix operators.
- Access to pipelined FUs, but without II degradation (cf Karczmarek and Arvind 2008).
- Output FIFOs can be avoided.
- Automatic insertion of arbiters.
- Super-scalar rule firing.
- *Assertion*: Our extensions can all be applied at once without conflicts (further testing/formal proof required).

Thank you for you attention.

Source Tarball Link

www.cl.cam.ac.uk/~djg11/wwwhpr/toy-bluespec-compiler.html

Bluespec: Tiny 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
```

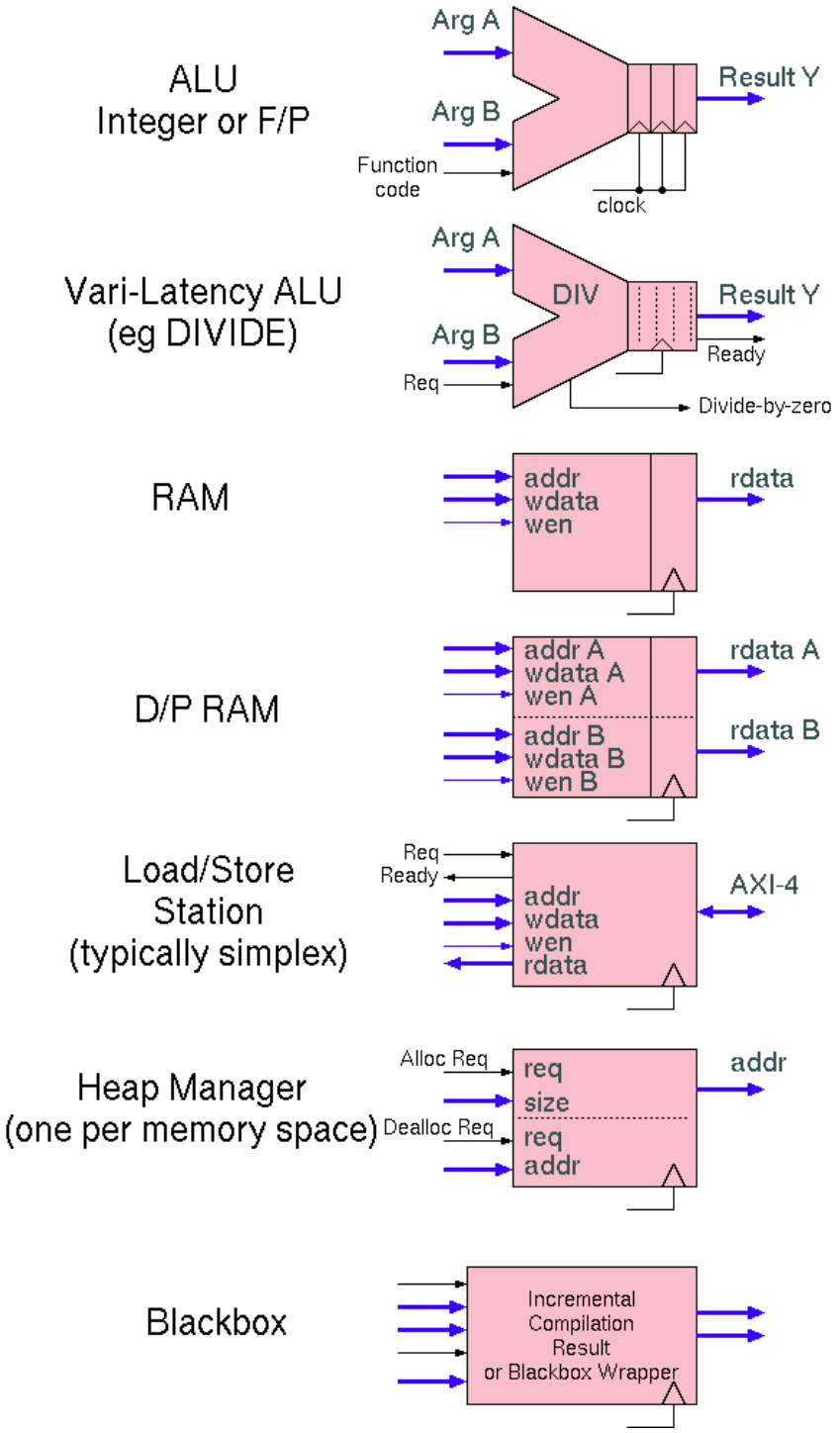
But, imperative expression using a conceptual thread is also useful to have, so Bluespec has a behavioural sub-language compiler built in.

Functional Units

As well as simple arithmetic and logic for back-end synthesis, HLS tool makes structural instantiation of major FUs in output netlist.

For each FU, scheduller needs basic data:

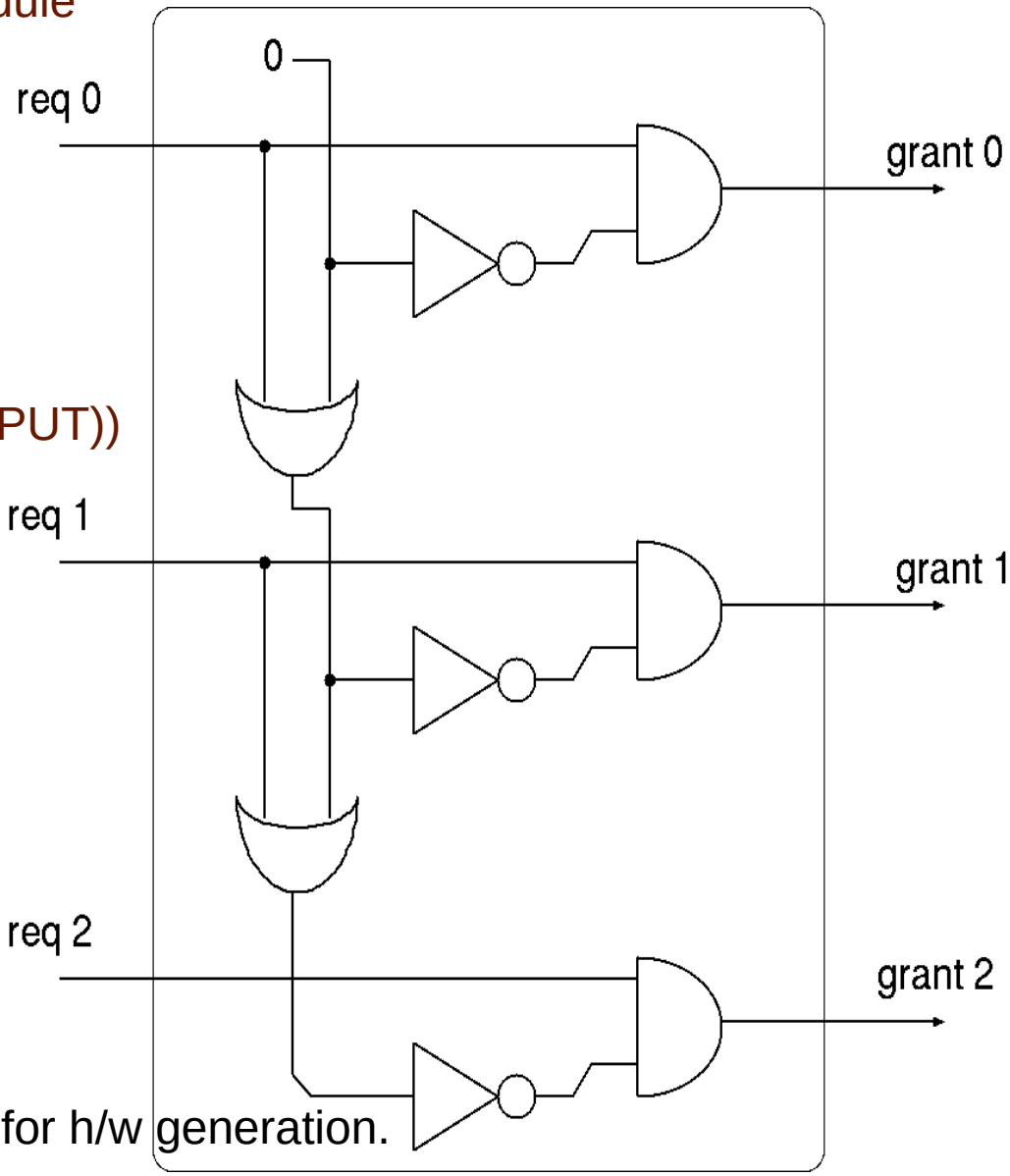
- EIS
- Ref-transparent
- Speculatively harmless
- Fixed/Vari delay
- Average Latency
- Re-initiation interval
- Costs Area/Energy.



A varadic Priority Arbiter in Chisel

```
class genPriEncoder(n_inputs : Int) extends Module
   {
     val io  = new Bundle {  }
     val terms = (1 to 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.