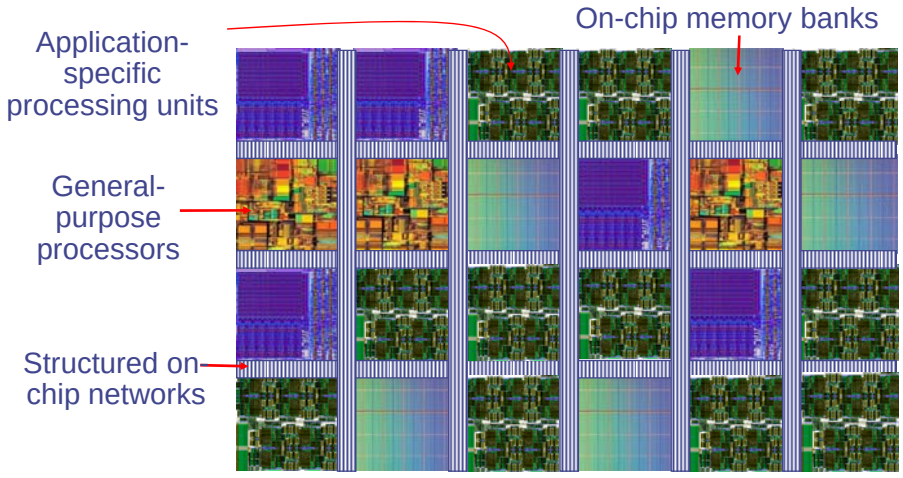
BSV[1] for ESL[2] Design

Rishiyur S. Nikhil, PhD.
CTO, Bluespec, Inc.

Lecture in 6.375, MIT
March 10, 2008

1: Bluespec SystemVerilog
2: Electronic System Level

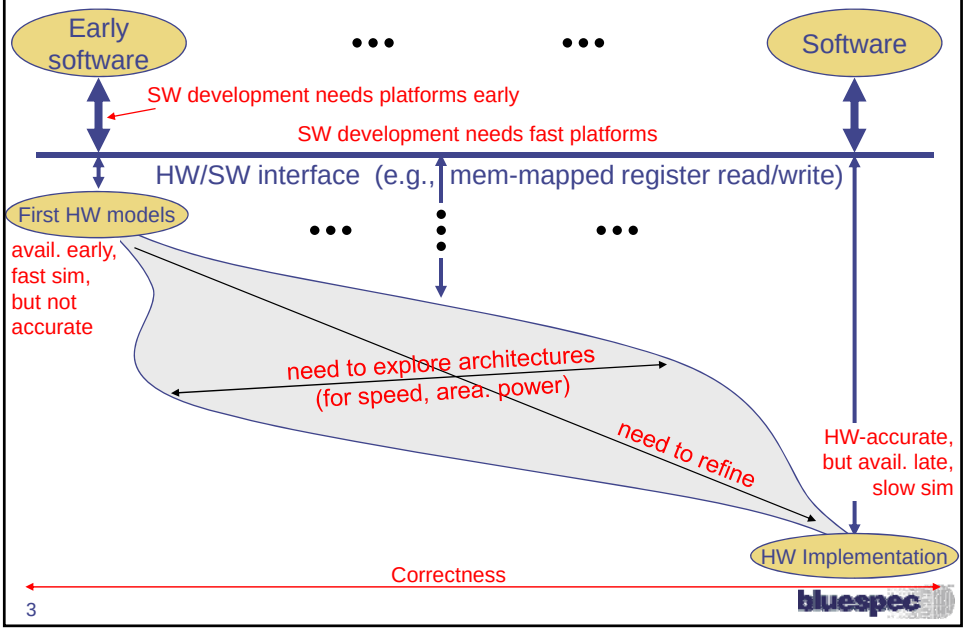
SoCs: complex assemblies of many application—specific blocks (IPs)



Many IPs must be implemented in HW to save power.
And, at many power/performance operating points

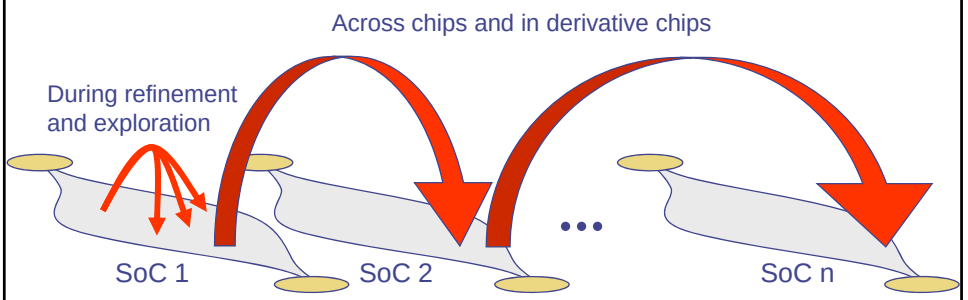
2

Central SoC design concerns



bluespec

Central SoC design concerns (contd.): reuse models, implementations, testbenches



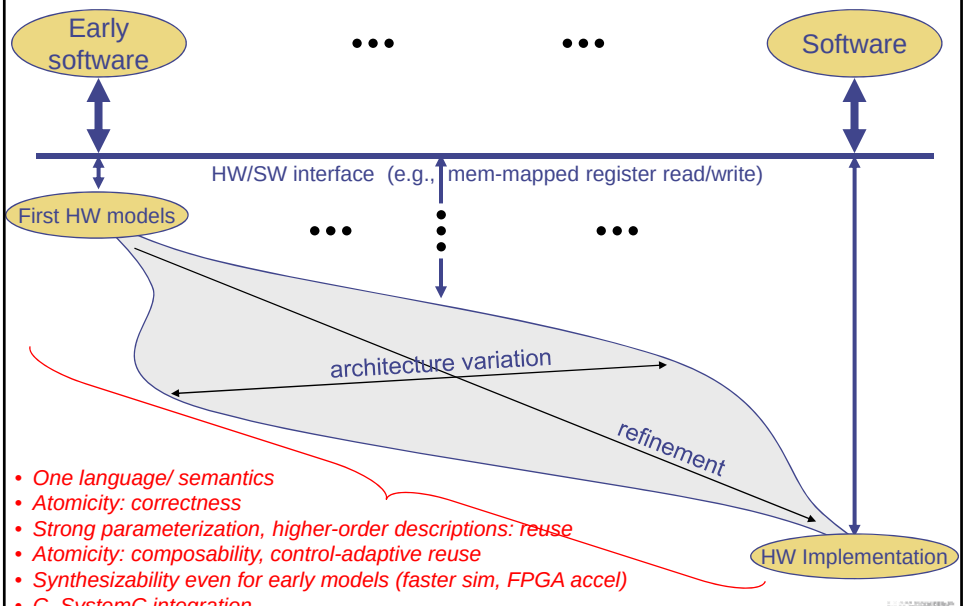
bluespec

The only feasible approach

- *Rapid prototyping for wireless designs: the five-ones approach*, M.Rupp, A.Burg and E.Beck, *Signal Processing 83:7, 2003, pp. 1427-1444*
  - "One code, one environment, one team, one documentation, one revision control"
- I.e., need a single, consistent language and environment for the whole ESL design process
  - But even this is not enough

bluespec

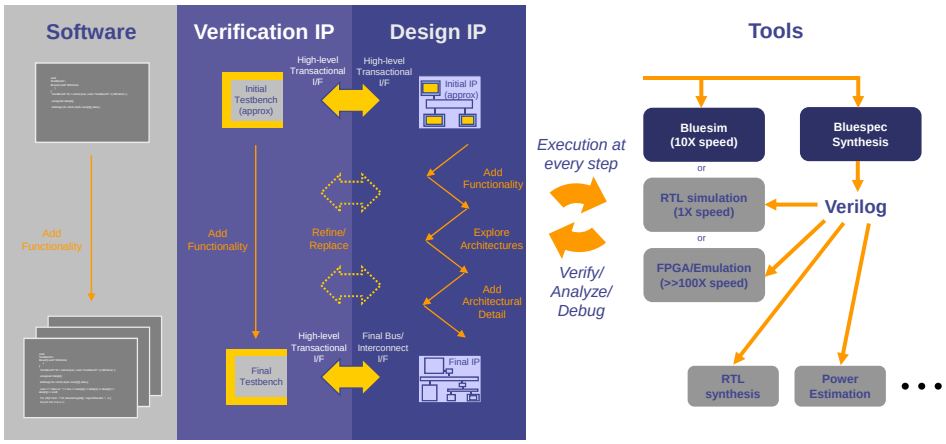
BSV addresses SoC design concerns



- *One language/ semantics*
- *Atomicity: correctness*
- *Strong parameterization, higher-order descriptions: reuse*
- *Atomicity: composability, control-adaptive reuse*
- *Synthesizability even for early models (faster sim, FPGA accel)*
- *C, SystemC integration*

bluespec

BSV enables a top-down refinement methodology



bluespec

Topics today

- Early models: executable, synthesizable
- FSMs
- Interface abstraction
- Parameterizable microarchitectures
- C/SystemC integration

bluespec

Early models (executable, synthesizable)

bluespec

Rules allow early models to be like executable and synthesizable specs: e.g., an ISS (instruction set simulator)

Excerpt from Assembler Manual

add
Add

add	rD, rA, rB	(OE=0, Rc=0)
add.	rD, rA, rB	(OE=0, Rc=1)
addo	rD, rA, rB	(OE=1, Rc=0)
addo.	rD, rA, rB	(OE=1, Rc=1)

XO Instruction Form

Description

The sum of the contents of register rA and register rB is loaded into register rD.

Pseudocode

(rD) ← (rA) + (rB)

Registers Altered

- rD.
- CR[CR0][LT, GT, EQ, SO] if Rc=1.
- XER[SO, OV] if OE=1.

If an overflow occurs, it is possible that the contents of CR0 do not reflect the infinitely precise result.

Exceptions

- None.

Compatibility

This instruction is defined by the PowerPC user instruction-set architecture (UISA). It is implemented by all PowerPC processors.

BSV code:
- Close to spec
- No extraneous logic for managing shared resources
- Scalable – incrementally add functionality

```
rule add(instruction_D.first.op == Add && cpuState ==
    Execute);
  Instruction add = instruction_D.first;

  // Get the register values involved
  let a = regfile.sub(add.ra);
  let b = regfile.sub(add.rb);

  // Perform the addition
  let d = add32(a, b);

  // Writeback the results
  update_regfile(add.rd, d);

  // Calculate the overflow bit
  Bool overflow = (((a[31] == 1) && (b[31] == 1) &&
     (d[31] == 0)) || ((a[31] == 0) && (b[31] == 0)
    &&
     (d[31] == 1)));

  // Update Flags
  update_condition0_ovl(add.rc, d, overflow);
  update_carry_overflow(add.oe, overflow, False, 1'b0);

  increment_pc;
endrule
```


bluespec

FPGA acceleration

- Since all of BSV is synthesizable, even such early, functional models can be synthesized into RTL and run on FPGA platforms/emulators
  - Not optimized, but still faster than simulation
- Typical uses:
  - Architectural exploration
  - Fast emulation for earlier verification
  - Early software development

FSMs

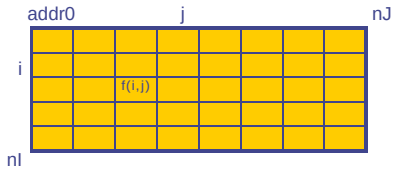
The StmtFSM package

- BSV's *rules* are of course powerful enough to express any FSM.
  - A rule's condition identifies a state
  - A rule's Action specifies the state transition and any other actions for that state
- However, there are some common idioms ("design patterns") for FSMs which are made simpler with syntactic support

bluespec

Example

- Initialize a memory with a 2-D pattern
  - An FSM is needed since the memory can only accept 1 write per cycle




```
typedef enum { Pre, Loop, Post, Done} Phases
   deriving (Bits, Eq);

Reg#(Phases) p  <- mkReg (Pre);
Reg#(int) i     <- mkRegU;
Reg#(int) j     <- mkRegU;
Reg#(Addr) addr <- mkRegU;
```

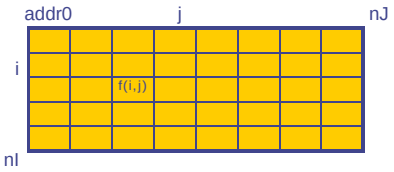
```
rule prelude (p == Pre);
    ... pre initialization actions ...
    p <= Loop; i <= 0; j <= 0; addr <= addr0;
endrule

rule loop (p == Loop && (i < nI) && (j < nJ));
   mem.write (addr, f (i, j));
   addr <= addr + 1;
   if (j < nJ-1)
      j <= j + 1;
   else begin
      j <= 0;
      if (i < nI-1)
         i <= i + 1;
      else
         p <= Post;
   end
endrule

rule postlude (p == Post);
   ... post-initialization actions ...
   p <= Done;
endrule
```

"Messy "bureaucracy" to sequence the FSM

bluespec

Example

- Initialize a memory with a 2-D pattern
  - (must be written as an FSM since only memory can only accept 1 write per cycle)




```
import  StmtFSM::*

Reg#(int) i          <- mkRegU;
Reg#(int) j          <- mkRegU;
Reg#(Addr) addr <- mkRegU;
```

```
Stmt s = seq
            action
               ... pre initialization actions ...
               addr <= addr0;
            endaction
            for (i <= 0; i < nI; i <= i + 1)
               for (j <= 0; j < nJ; j <= j + 1) action
                  mem.write (addr, f (i, j));
                  addr <= addr + 1;
               endaction;
            ... post-initialization actions ...
         endseq;

FSM fsm <= mkFSM (s);

rule ...
   fsm.start();
endrule
```


bluespec

StmtFSM sublanguage

- The Stmt sublanguage is signalled by 'seq-endseq' or 'par-endpar' brackets
  - Representing expressions of type 'Stmt'
  - A Stmt is a *specification* of an FSM
- Sublanguage constructs (cf. Ref Guide C.5):
  - Primitives: any Action (including composite actions)
  - *seq s1...sN endseq*: sequential composition
  - *par s1...sN endpar*: parallel composition
    - All *sJ*'s initiated in parallel; completes when all have completed
  - *if-then*, *if-then-else*
  - *for-*, *while-*, *repeat(n)-* loops, with optional *break*, *continue*

bluespec

Using and composing FSMs

- Various modules take Stmt specs as parameters and generate the specified FSMs
- FSMs have a standard BSV interface:

```
interface FSM;
   method Action  start          ();
   method Bool    done           ();
   method Action  waitTillDone   ();
endinterface: FSM
```

FSM interface

```
module mkFSM #( Stmt  s ) ( FSM );
```

mkFSM module

- Note: fsm methods can be used from other FSMs
  - fsm.done() can be used in rule conditions
  - fsm.waitTillDone() can be used in a rule body to block the rule (implicit condition is same as fsm.done())

bluespec

StmtFSM semantics, implementation

- Integrated cleanly into the language
  - mkFSM takes the Stmt spec and generates the corresponding Rules—the Actions in the spec become rule bodies
  - Compiler manages all state book-keeping
    - Automatic generation of state variables, state registers, state update
  - Standard rule semantics for FSM actions (e.g., shared counters, queues)
    - Blocks on implicit conditions, if necessary
    - Atomic (arbitrated in usual way w.r.t. other actions)
- Fully synthesizable

bluespec

Modules with FSM interfaces

- mkFSM#(Stmt s) (FSM)
- mkFSMWithPred#(Stmt s, Bool b) (FSM)
  - Global 'disable/suspend' control on the FSM
- mkAutoFSM#(Stmt s) (Empty)
  - Build the FSM, plus boilerplate that runs it exactly once and quits
- See Ref Guide C.5 for more discussion

- You can also write your own module with an FSM interface
  - See "FSMs from lists", generating an FSM from a list of (current-state,action,next-state) triples, in Small Examples Suite, Example 12g http://www.bluespec.com/wiki/SmallExamples/

bluespec

StmtFSM - example

```
Stmt specfsm =
  seq
     write( 15, 51 );
     read( 15 ) ;
     ack ;
     ack ;
     write( 16, 61 ) ;
     write( 17, 71 ) ;
     // a memory operation and
     // an acknowledge can occur
     // simultaneously
     action
        read( 16 ) ;
        ack ;
     endaction
     action
        read( 17 ) ;
        ack ;
     endaction
     ack ;
     ack ;
  endseq ;

FSM testfsm <- mkFSM (specfsm);
```

- seq / endseq define a sequence (multi-cycle)

- action / endaction define a simultaneous block (in one cycle)

- par / endpar defines a parallel block (multi-cycle)

```
rule run ( True );
   testfsm.start ;
endrule
rule done (testfsm.done);
   $finish(0);
endrule
```


bluespec

StmtFSM - example

```
Stmt specfsm =
  seq
     write( 15, 51 );
     read( 15 ) ;
     ack ;
     ack ;
     write( 16, 61 ) ;
     write( 17, 71 ) ;
     // a memory operation and
     // an acknowledge can occur
     // simultaneously
     action
        read( 16 ) ;
        ack ;
     endaction
     action
        read( 17 ) ;
        ack ;
     endaction
     ack ;
     ack ;
  endseq ;

mkAutoFSM (specfsm);
```

- Same FSM, using mkAutoFSM

21

State Machine Generation

- Example

```
// Specify an FSM generating a test seqence
Stmt test_seq =
  seq
    for (i <= 0; i < nI; i <= i + 1)                //ede
      for (j <= 0; j < nJ; j <= j + 1) begin        //edestinatn
         let pkt <- gen_packet ();
         send_packet (i, j, pkt);                   // tesi-jpahinidtn
      end

    // ten, tsaitatinbysadingpdessimultandytandst
    action
       send_packet (0, 1, pkt0);                    // tods1
       send_packet (1, 1, pkt1);                    // tods1(sdisn)
    endaction
  endseq

mkAutoFSM (fsm);        // GenateFSMaddnitamtally
```

22

On efficiency of generated FSMs

- In general, compositional implementation of FSMs will not be as efficient as an implementation with:
  - State-encoding optimizations based on global knowledge
    - E.g., use single wide register for (i,j)
    - E.g., one-hot encodings
  - State-encoding optimizations based on domain knowledge
    - E.g., state can be inferred from other data registers
- Nevertheless, BSV's StmtFSM package is useful most of the time
  - Rarely fall back to express FSMs directly as rules
  - Because FSM is a standard BSV interface, you can also write your own FSM generators
    - See "FSMs from lists", Small Examples Suite, Example 12g http://www.bluespec.com/wiki/SmallExamples/

FSM summary

- Stmt sublanguage captures certain common and useful FSM idioms:
  - sequencing, parallel, conditional, iteration
- FSM modules automatically implement Stmt specs
- FSM interface permits composition of FSMs
- Most importantly, *same Rule semantics*
  - Actions in FSMs are atomic
  - Actions automatically block on method implicit conditions
  - Parallel actions, whether in the same FSM or different FSMs, automatically arbitrated based on atomicity semantics

Interface abstraction

bluespec

Transactional interfaces: Get/Put

- Provides simple handshaking mechanism for getting data from a module or putting data into it
- Easy to connect together

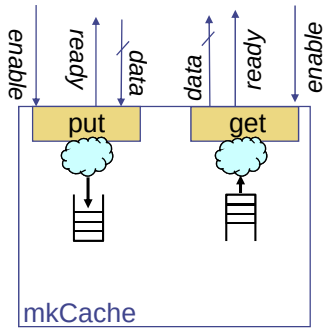
```
interface Get#(type t);                          // polymorphic
   method ActionValue#(t) get();
endinterface: Get

interface Put#(type t);
   method Action put(t x);
endinterface: Put
```

bluespec

Get and Put Interfaces

- Example: interface provided by a cache (c) to a processor (p)

```
interface CacheIfc;
  interface Put#(Req_t)   p2c_request;
  interface Get#(Resp_t)  c2p_ response;
  ...
endinterface
```




```
module mkCache (CacheIfc);
  FIFO#(Req_t)   p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;

  ... declarations and logic ...

interface p2c_request;
    method Action put (Req_t  req);
      p2c.enq (req);
    endmethod
  endinterface

  interface c2p_response;
    method ActionValue#(Resp_t) get ();
      let resp = c2p.first; c2p.deq;
      return resp;
    endmethod;
  endinterface

endmodule
```

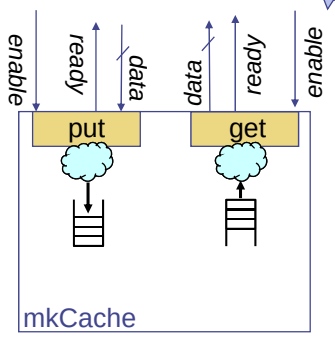

bluespec

Get and Put Interfaces

```
function Put#(Req_t) fifoToPut(FIFO#(Req_t) fifo);
return (
  interface Put;
    method Action put (a);
      fifo.enq (a);
    endmethod
  endinterface);
endfunction
```

- (c) to a processor (p)

```
mkCache (CacheIfc);
(Req_t)   p2c <- mkFIFO;
(Resp_t) c2p <- mkFIFO;

s expressing cache logic ...

interface p2c_request;
    method Action put (Req_t  req);
      p2c.enq (req);
    endmethod
  endinterface

  interface c2p_response;
    method ActionValue#(Resp_t) get ();
      let resp = c2p.first; c2p.deq;
      return resp;
    endmethod;
  endinterface

endmodule
```

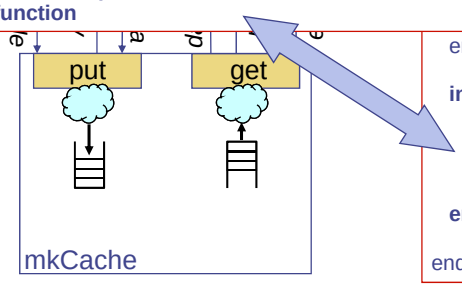


bluespec

Get and Put Interfaces

- Example: interface provided by a cache (c) to a processor (p)

```
function Get#(Resp_t) fifoToGet(FIFO#(Resp_t) fifo);
return (
  interface Get;
    method ActionValue#(Resp_t) get ();
      let a = fifo.first;
      fifo.deq;
      return a;
    endmethod;
  endinterface);
endfunction
```



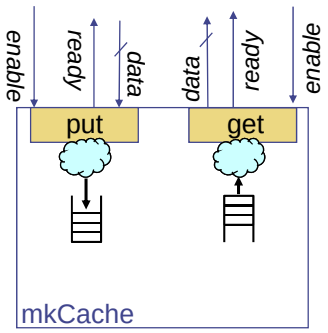

```
mkCache (CacheIfc);
(Req_t)   p2c <- mkFIFO;
(Resp_t) c2p <- mkFIFO;

s expressing cache logic …

ce p2c_request;
od Action put (Req_t  req);
2c.enq (req);
method
endinterface

interface c2p_response;
  method ActionValue#(Resp_t) get ();
    let resp = c2p.first; c2p.deq;
    return resp;
  endmethod;
endinterface

endmodule
```


bluespec

"Interface transformers": Converting FIFOs to Get/Put

- The BSV library provides functions to convert FIFOs to Get/Put

```
interface CacheIfc;
  interface Put#(Req_t)    p2c_request;
  interface Get#(Resp_t)  c2p_ response;
  …
endinterface
```




```
module mkCache (CacheIfc);
  FIFO#(Req_t)   p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;

  … rules expressing cache logic …

interface p2c_request = fifoToPut (p2c);

  interface c2p_response = fifoToGet (c2p);
  …
endmodule
```

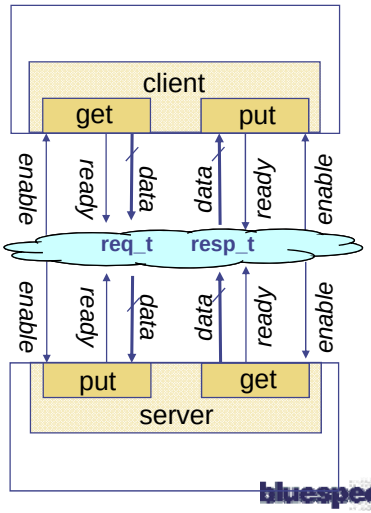
Absolutely no difference in the HW!

bluespec

Client/Server interfaces

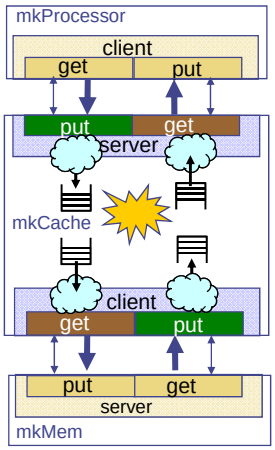
- Get/Put pairs are very common, and duals of each other, so the library defines Client/Server interface types for this purpose

```
interface Client #(req_t, resp_t);
   interface Get#(req_t)    request;
   interface Put#(resp_t)  response;
endinterface

interface Server #(req_t, resp_t);
   interface Put#(req_t)    request;
   interface Get#(resp_t)  response;
endinterface
```



Client/Server interfaces

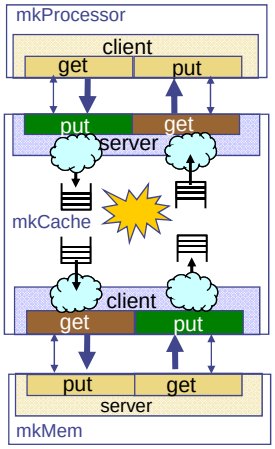
```
interface CacheIfc;
   interface Server#(Req_t, Resp_t)  ipc;
   interface Client#(Req_t, Resp_t)   icm;
endinterface
```




```
module mkCache (CacheIfc);
   FIFO#(Req_t)   p2c <- mkFIFO;
   FIFO#(Resp_t) c2p <- mkFIFO;

   FIFO#(Req_t)   c2m <- mkFIFO;
   FIFO#(Resp_t)  m2c <- mkFIFO;

   … rules expressing cache logic …

   interface Server ipc
      interface Put put(a);
         method Action put(a);
            p2c.enq(a);
         endmethod
      endinterface

      interface Get get;
         method ActionValue#(Resp_t) get;
            c2p.deq;
            return c2p.first;
         endmethod
      endinterface
   endinterface        // Server
```

Client/Server interfaces and more interface transformers

```
interface CacheIfc;
  interface Server#(Req_t, Resp_t)  ipc;
  interface Client#(Req_t, Resp_t)  icm;
endinterface
```



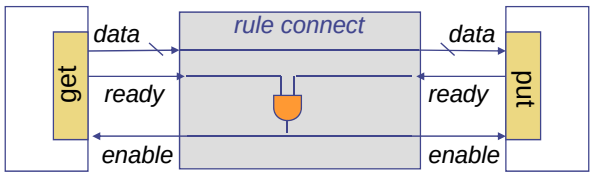

```
module mkCache (CacheIfc);
  // from / to processor
  FIFO#(Req_t)   p2c <- mkFIFO;
  FIFO#(Resp_t) c2p <- mkFIFO;

  // to / from memory
  FIFO#(Req_t)   c2m <- mkFIFO;
  FIFO#(Resp_t)  m2c <- mkFIFO;

  … rules expressing cache logic …

interface ipc = fifosToServer (p2c, c2p);

  interface icm = fifosToClient (c2m, m2c);
endmodule
```


bluespec

Connecting Get and Put

- A module m1 providing a Get interface can be connected to a module m2 providing a Put interface with a simple rule

```
module mkTop (…)
  Get#(int) m1 <- mkM1;
  Put#(int) m2 <- mkM2;

  rule connect;
    let x <- m1.get();  m2.put (x);          // rule implicit conditions
  endrule
endmodule
```



bluespec

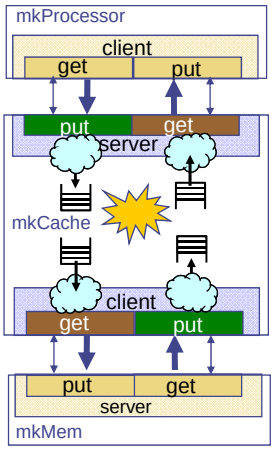
Abstracting connections: Connectables

- There are many pairs of types that are duals of each other
  - Get/Put, Client/Server, YourTypeT1/YourTypeT2, …
- The BSV library defines an *overloaded* module **mkConnection** which encapsulates the connections between such duals
  - The BSV library predefines the implementation of mkConnection for Get/Put, Client/Server, etc.
- Because overloading in BSV is extensible, you can overload mkConnection to work on your own interface types T1 and T2
- The BSV library also recursively defines mkConnection for *tuples* and *vectors* of interfaces that are already individually connectable (including your own interface types)

bluespec

mkConnection

- Using these interface facilities, assembling systems becomes very easy




```
interface CacheIfc;
  interface Server#(Req_t, Resp_t)  ipc;
  interface Client#(Req_t, Resp_t)  icm;
endinterface

module mkTopLevel (…)
  // instantiate subsystems
  Client #(Req_t, Resp_t)       p  <- mkProcessor;
  Cache_Ifc #(Req_t, Resp_t)    c  <- mkCache;
  Server #(Req_t, Resp_t)       m <- mkMem;

  //  instantiate connects
  mkConnection (p, c.ipc);   // Server connection
  mkConnection (c.icm, m); // Client connection
endmodule
```

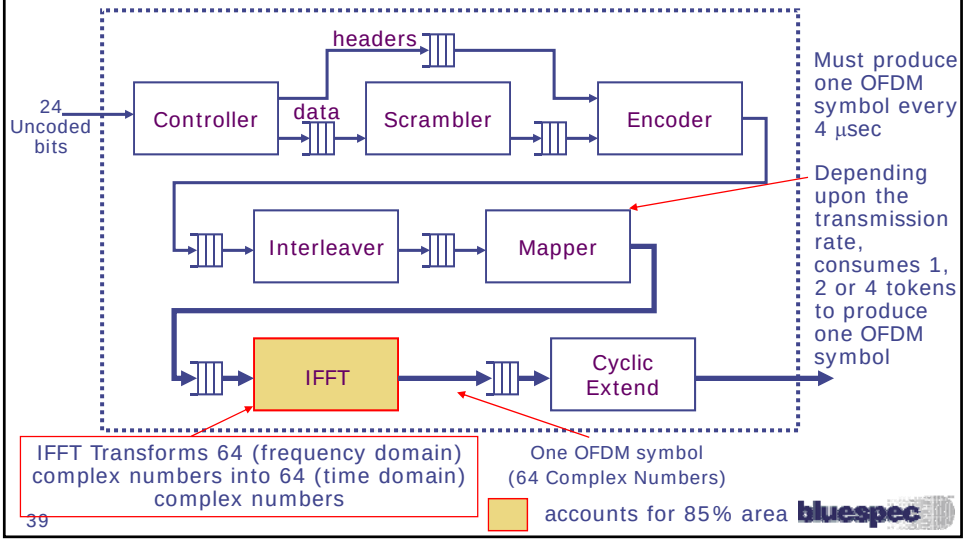

bluespec

Summary of interface abstraction

- BSV allows writing very high level transactional interfaces that remain synthesizable and efficient
- We can very quickly, succinctly and correctly define very complex interfaces, and connect them, using interface abstraction, and BSV library elements Get/Put, Client/Server, Connectables, etc.

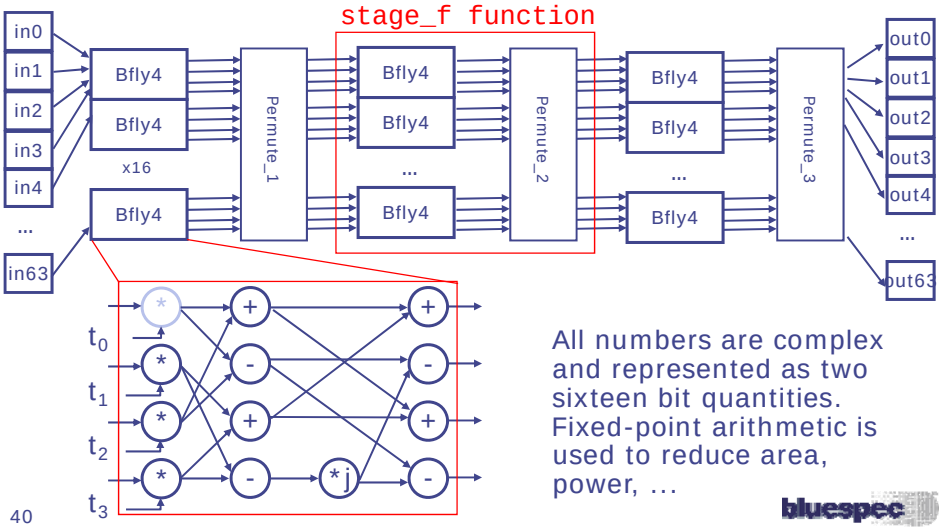
Parameterizable microarchitecture

Example: 802.11a Transmitter



39

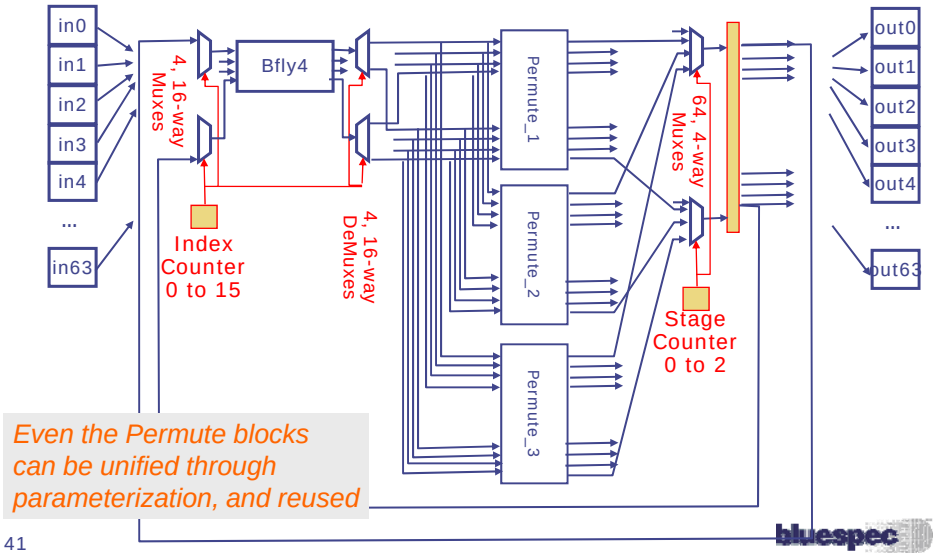
Combinational IFFT



All numbers are complex and represented as two sixteen bit quantities. Fixed-point arithmetic is used to reduce area, power, ...

40

Superfolded circular pipeline: Just one Bfly-4 node!



Even the Permute blocks can be unified through parameterization, and reused

bluespec

Different microarchitectural choices ➔ different power/area/speed operating points

- Higher-order descriptions in BSV
  - Can write functions that compute with *any* types: modules, rules, actions, interfaces
- Parameterization in BSV
  - Parameters can be of *any* type
  - Can conditionally instantiate modules/ state elements
- Thus: can write a single source which, depending on parameters, "unfolds" to different microarchitectures:
  - degree of "data parallelism",
  - degree of pipelining,
  - degree of resource sharing, etc.
- Power/area/speed tradeoff not easy to predict without actual synthesis of different microarchitectures, because of complex interaction with optimization
- E.g., see MIT's experiences with 802.11xx, OFDM, H.264

bluespec

C and SystemC integration

Encapsulating existing C code

- Well-defined mechanism to incorporate existing C code (for both Bluesim and Verilog sim)
  - "import BDPI"
- Typical uses:
  - Initial rough models of IP behavior, to eventually be replaced by BSV code
  - Initial reuse of reference codes for standard algorithms (wireless, audio/video, image processing, security, ...)
  - Testbenches (may never be replaced by BSV code)
    - Generate input/output vectors
    - Generate random numbers
    - Use C to do complex arithmetic
  - Instrumentation: write log files of model activity

The *import-BDPI* statement

- Importing C function “add32” as BSV function with the same name:

```
import “BDPI”
  function Bit#(32) add32 (Bit#(32) v1, Bit#(32) vs);
```

- Importing C function “add32” as BSV function called “add”:

```
import “BDPI” add32 =
  function Bit#(32) add (Bit#(32) v1, Bit#(32) vs);
```

- Imported C functions can also have Action and ActionValue#() return types

Argument/result Types

BSV Type	C Type
String	char*
Bit#(0) – Bit#(8)	unsigned char
Bit#(9) – Bit#(32)	unsigned int
Bit#(33) – Bit#(64)	unsigned long long
Bit#(65) -	unsigned int*
Bit#(n)	unsigned int*

- Other types are allowed if they can be packed to bits

Wide and Polymorphic Data

- Wide and poly data are passed by reference: `unsigned int*`
- For wide data, no size is passed to the C function; the size should be known
- For poly data, BSV takes no position on how the size is passed; left to the user
  - Size can be passed as an additional argument (see next slide)

Polymorphic Example

```
// This function computes a checksum for any size
// bit-vector.  The second argument is the size of
// the input bit-vector.

import "BDPI" checksum =
   function Bit#(32) checksum_raw (Bit#(n), Bit#(32));


// This wrapper handles the passing of the size

function Bit#(32) checksum (Bit#(n) vec);
   return checksum_raw(vec, fromInteger(valueOf(n)));
endfunction
```

Implicit pack/unpack

- Example:

```
import "BDPI“
   function Bool my_and (Bool, Bool);
```

- Since `Bool` packs to a `Bit#(1)`, it would connect to a C function such as the following

```
unsigned char
my_and (unsigned char x, unsigned char y);
```


bluespec

Compilation and Linking

- Imported C functions are compiled like any other C source file
- See *BSV User Guide* for details on how to link in the .o files
- The bsc compiler generates all the auxiliary “glue” for linking in the compiled C code, whether in Bluesim or in Verilog sim

bluespec

Encapsulating BSV into SystemC models

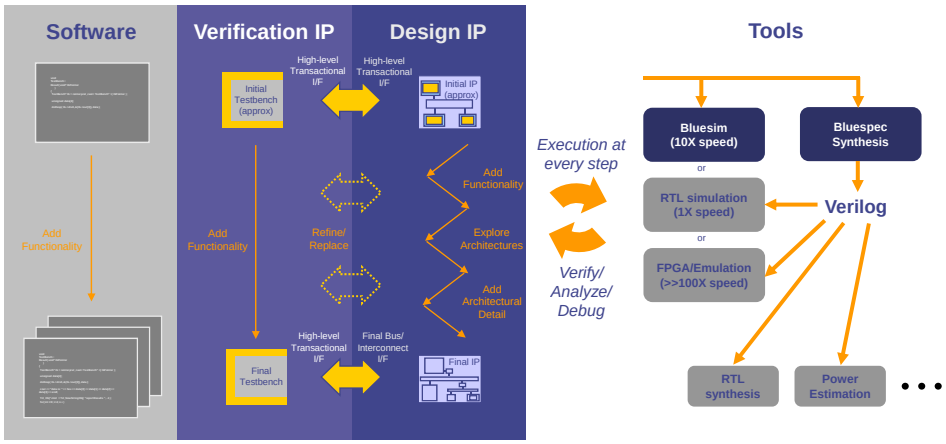
- SystemC (www.systemc.org) is very popular for system modeling and testbenches
- The *bsc* compiler can encapsulate a BSV module into a SystemC module, which can then be used in a standard SystemC environment
  - "-systemc" flag to bsc (cf. *User Guide* Sec 4.2)
  - Incorporates the fast Bluesim engine
- Typical uses:
  - Plug into existing SystemC models and environments to reuse existing testbench

bluespec

Wrapup

bluespec

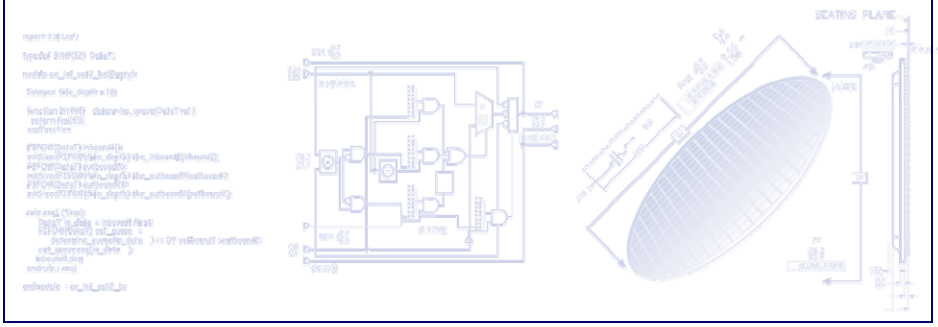
BSV enables a top-down refinement methodology



bluespec

ESL SYNTHESIS

End



Extras

bluespec

Traditional (C/C++ based) High Level Synthesis

- Source code: standard C/C++
  - Sequential source semantics to specify highly parallel implementations!
- Works only for compute-intensive "loop-and-array" algorithms (which can be automatically parallelized)
  - Yes: computation kernels of 802.x, H.264, MIMO, ...
  - No: Processors, caches, DMAs, interconnects, I/O peripherals, ..., and even the rest of 802.x, H.264, ...
- Even acceptable algorithms need significant "massaging" to produce good synthesis
- Plus, need tool-specific, and sometimes technology-specific "constraints" guiding synthesis (loop unrolling, resource sharing, ...)
  - Observation: these essentially constrain/specify the microarchitecture (which is explicit in BSV)
- Non-trivial integration into rest of system

BSV, in contrast: *microarchitecture precisely specified by designer, but with such strong* functional abstraction *and* parameterization *that:*

- *often shorter than C/C++ codes*
- *single source can represent family of microarchitectures (degree of pipelining, degree of data parallelism, degree of ad-hoc parallelism, ...)*
- *transparent, predictable, controllable*
- *Applications demonstrated: H.264 decode, OFDM, MIMO decode, CIECAM, and more*

bluespec