Bluespec Codesign Language: A Unified Language to Enable HW/SW Codesign

Nirav Dave
SRI International

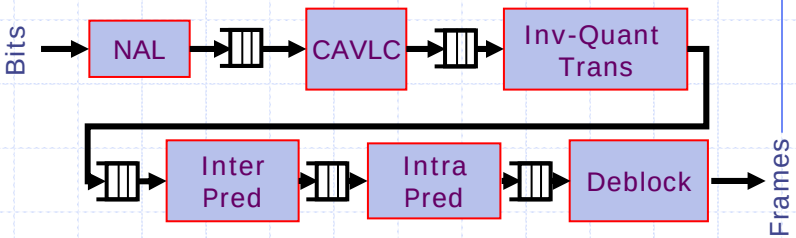
LAW
December 6, 2011

Modern SoCs

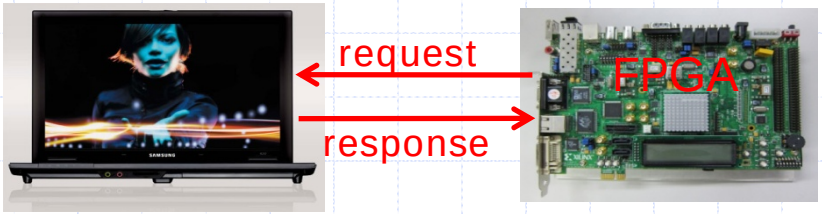
- Functionality is determined by power/energy issues
- Hardware accelerated solutions consume dramatically less power than pure software solutions
  - ⇒ Most SoC's have many specialized blocks
- Many SoCs have to support full software stacks which at the bottom must interact with special purpose hardware efficiently

Kinds of Hardware-Software Interactions

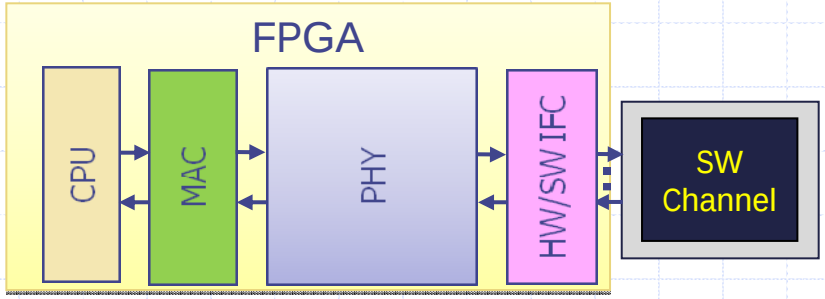
- Hardware accelerators
  - H.264: any block can be implemented in HW or SW



- Hardware calling software for functionality not supported in hardware



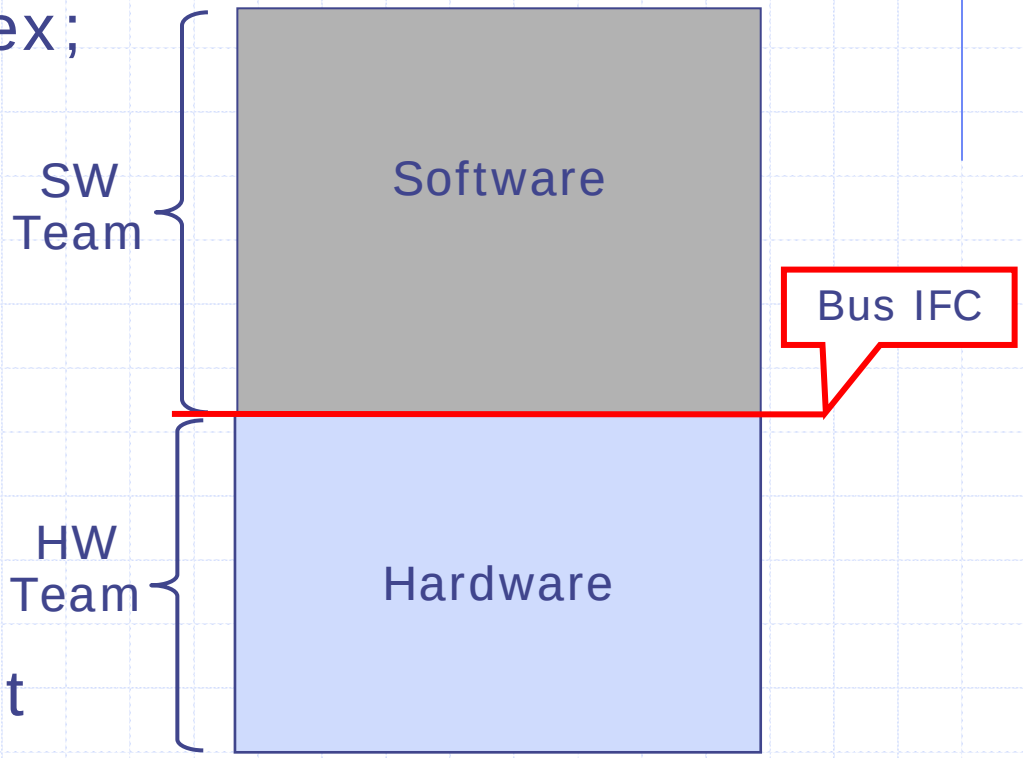
- Software driving hardware as in testing or simulation



Different scenarios but similar problems

Standard HW/SW Design

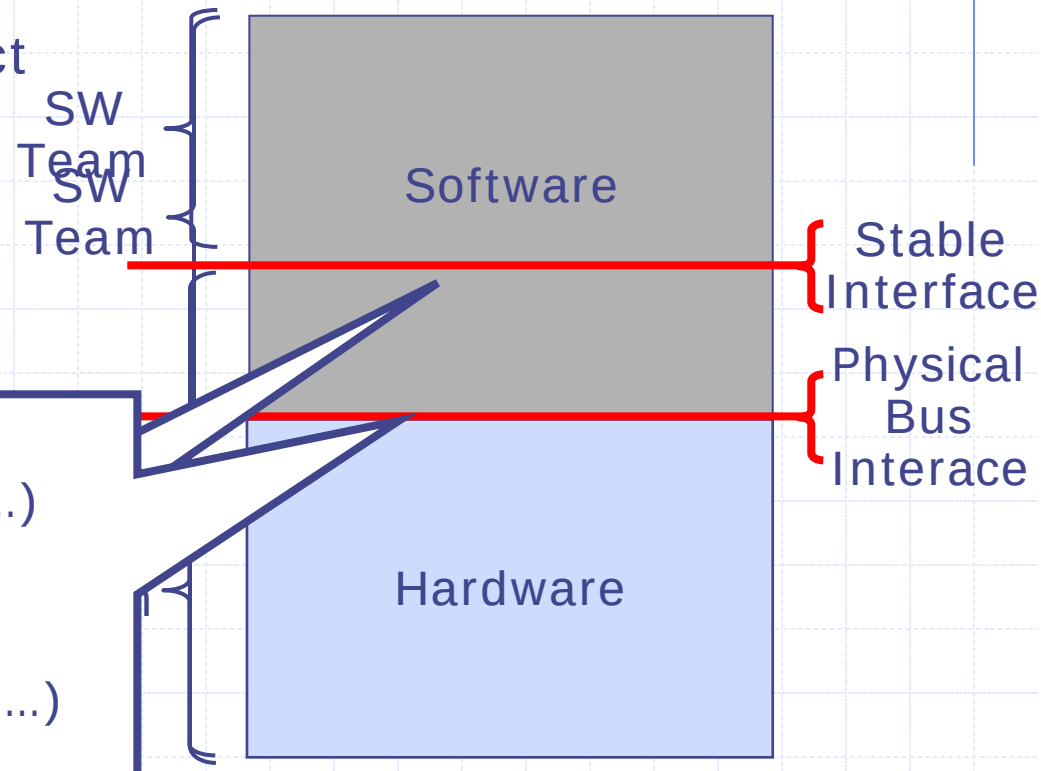
- HW/SW IFC is complex; must deal with:
  - Marshalling of data on fixed-datapath bus
  - Arbitration of requests
  - Low-level parallelism needed for efficient communication and control
- IFC needs to be set early in design



IFC is difficult to get right the first time around and changes in IFC later in the design is hard

Improving the Interface

- Define a more abstract interface and give the responsibility of low-level software to the HW team
- Isolated most of the



Interface:

writeDataToMem(0x00,...)

startFU(i)

resetFU(i)

getDataFromMem(0xA0,...)

...

interruptOperations()

...

The Real problem

- The issue is that HW and SW do not have the same underlying semantic model
  - HW is gates and wires operating in parallel
  - SW is a sequence of instructions
- This mismatch leads to this confusion and errors
  - Also makes verification expensive and brittle

A unified Abstraction

- Represent HW and low-level SW in the same language:
  - Both need fine grained parallelism
  - “HW” team only has to deal with 1 language
  - Provide tools to generate both hardware and software
- Consequences:
  - Easy to move HW/SW boundary
  - No data representation/ interfacing issues from semantic mismatch of C and RTL
  - Complex bus interface is subsumed in a single language and can be abstracted cleanly

Panacea for HW-SW CoDesign

- A compiler that takes
  - A sequential program description
  - Some performance, cost and power constraints
- Automatically:
  - Identifies what part should be implemented in HW
  - Parallelizes HW part of design and parallelizes SW enough to exploit HW's parallelism
  - Insert proper HW-SW communication channels between HW and SW

This is an unrealistic expectation – it is difficult to convey high-level performance goals, constraints, suitable microarchitectures and come up with appropriate partitions

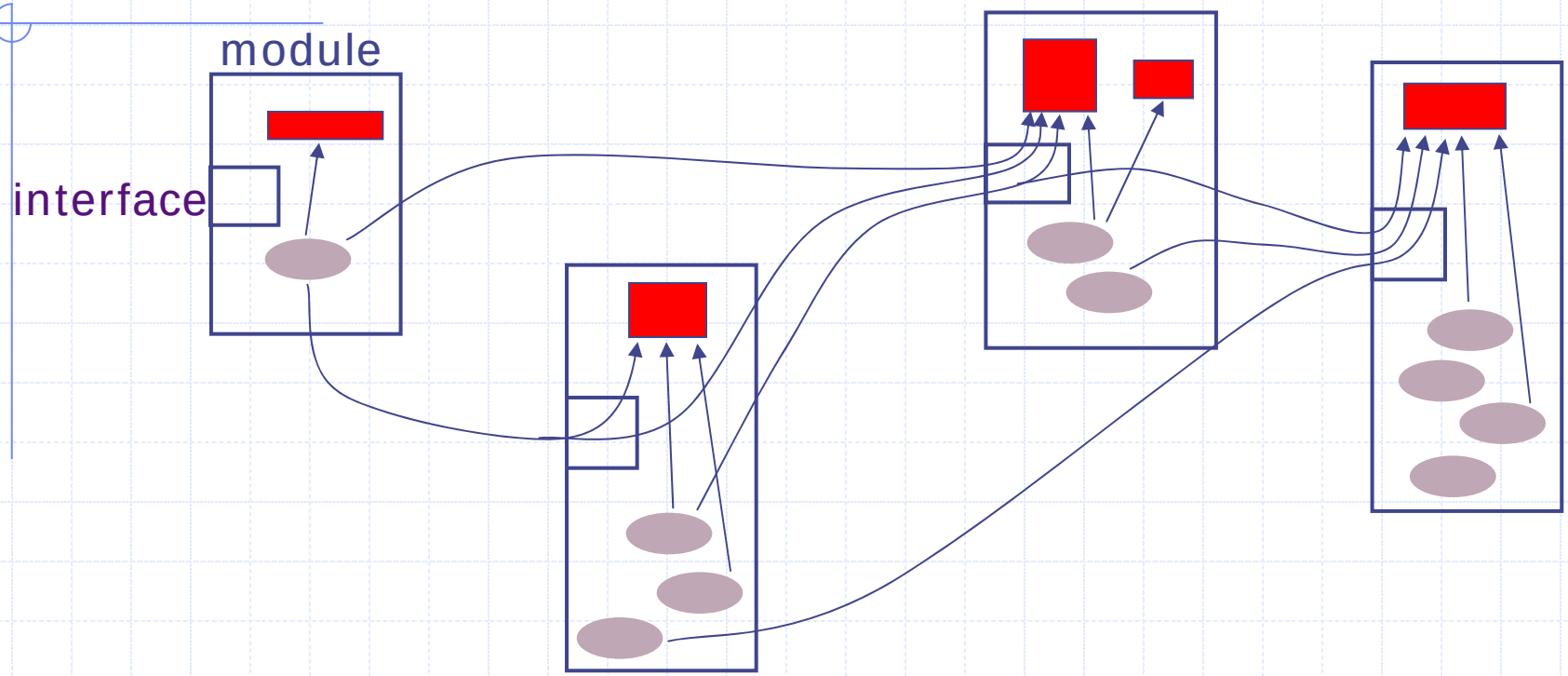
Our Goal: Facilitate Exploration

- A single parallel language to express both HW and parallel SW parts of an algorithm
  - Should be easy to express parallelism naturally
- Easy In-source specification of HW/SW partitioning
  - Should be easy to change partitioning
  - Designers should be able to reason about communication channel multiplexing directly in language.
- Compiler can easily separate the HW and SW compilation tasks and generate parts independently
  - Modularizes compilation and reasoning tasks
- Compiler should generate no-compromise hardware and high-quality software
- Clean model for fast verification / testing

Outline

- Motivation
- Bluespec Codesign Language
- Compilation
- Partitioning
- Encapsulating the Bus
- Compilation Results
- Verification

BCL: State and Rules organized into *modules*



All *state* (e.g., Registers, FIFOs, RAMs, ...) is explicit.

Behavior is expressed in terms of atomic actions on the state:

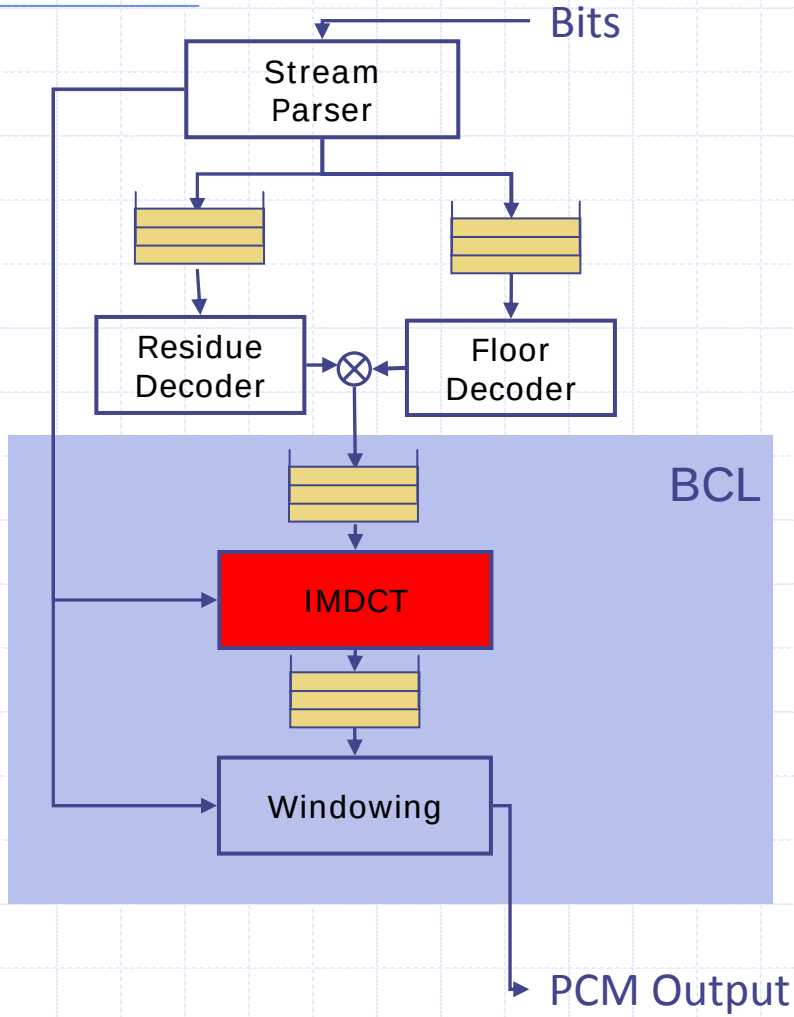
Rule: guard ➔ action

Semantics: Repeatedly *any* rule with valid guard & execute it

Bluespec Codesign Language (BCL)

- BCL is like Bluespec SystemVerilog (BSV)
  - Conditionals, guards, and action composition
  - Action and resource-level modularity
  - Expressing nondeterminsm/parallelism comes naturally via rules
- Extensions for efficient SW specification
  - Sequential composition and looping constructs
- BCL designs are partitioned into HW and SW domains which can be compiled separately and then integrated
- HW/SW compiler developed at MIT
  - HW translates to RTL via BSV Compiler
  - SW converted to C++

Example: Ogg Vorbis Decoder



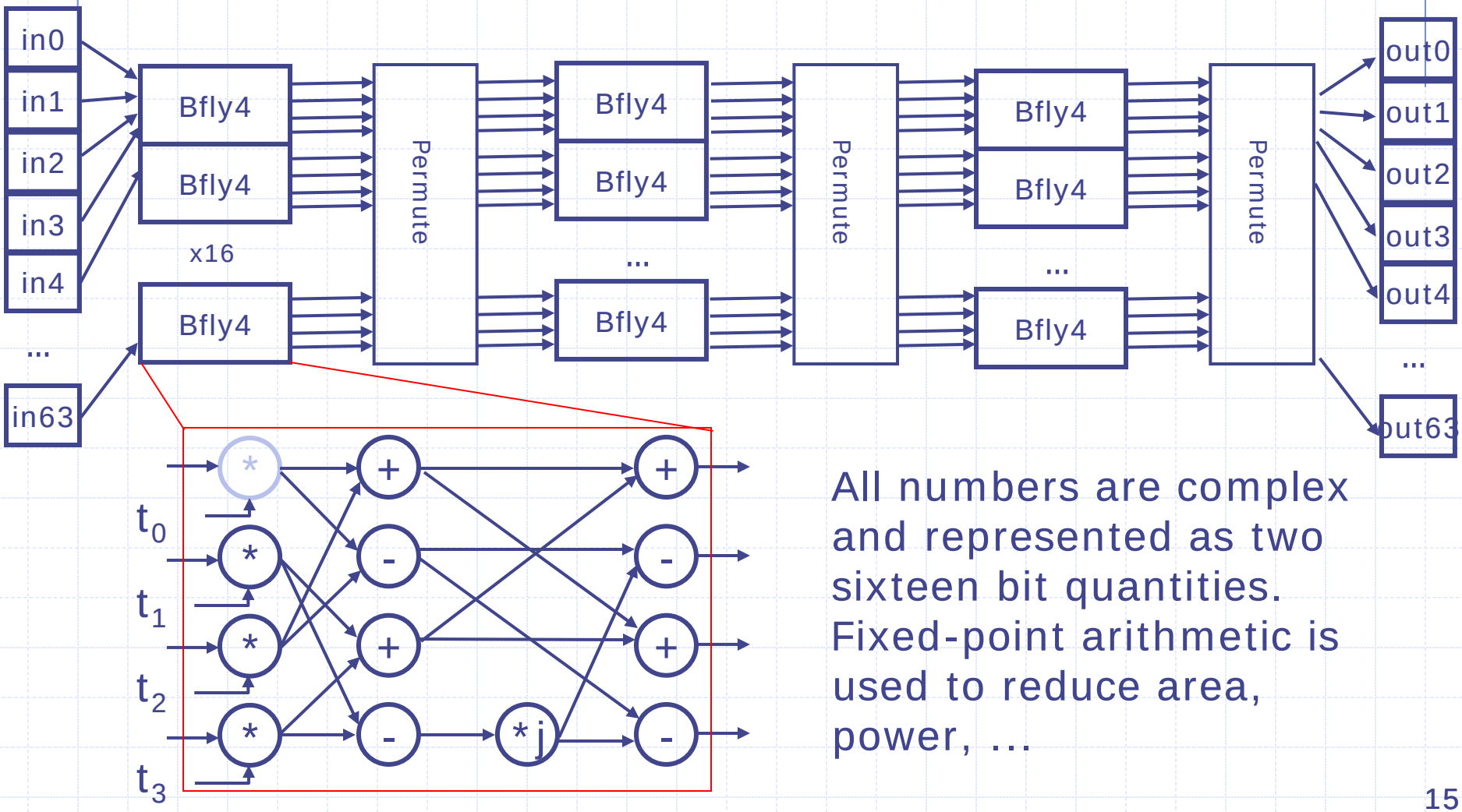
- Ogg Vorbis is a audio compression format roughly comparable to other compression formats: e.g. MP3, AAC, WMA.
- IMDCT takes the most computation
- Consider only the backend for BCL
  - Frontend is natural in SW

A Closer look at IMDCT

```
Array imdct(int N, Array vx){
  // preprocessing loop
  for(i = 0; i < N; i++){
    vin[i]   = convertLo(i,N,vx[i]);
    vin[i+N] = convertHi(i,N,vx[i]);
  }
  // do the IFFT
  vifft = ifft(2*N, vin);

  // postprocessing loop
  for(i = 0; i < N; i++){
    int idx = bitReverse(i);
    vout[idx] = convertResult(i,N,vifft[i]);
  }
  return vout;
}
```

Suppose we want to use hardware to accelerate FFT/IFFT computation

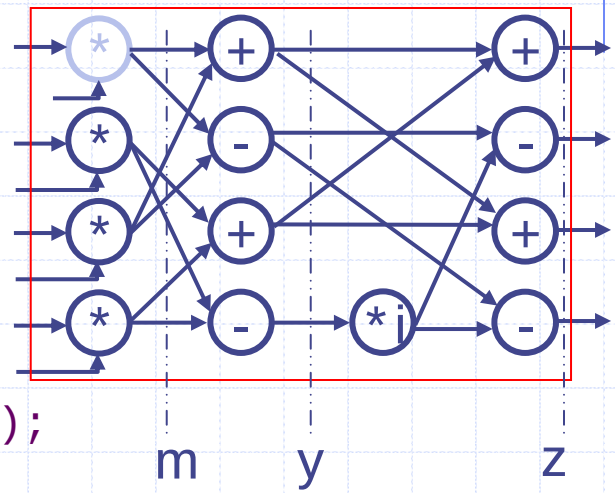
Combinational IFFT



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

4-way Butterfly

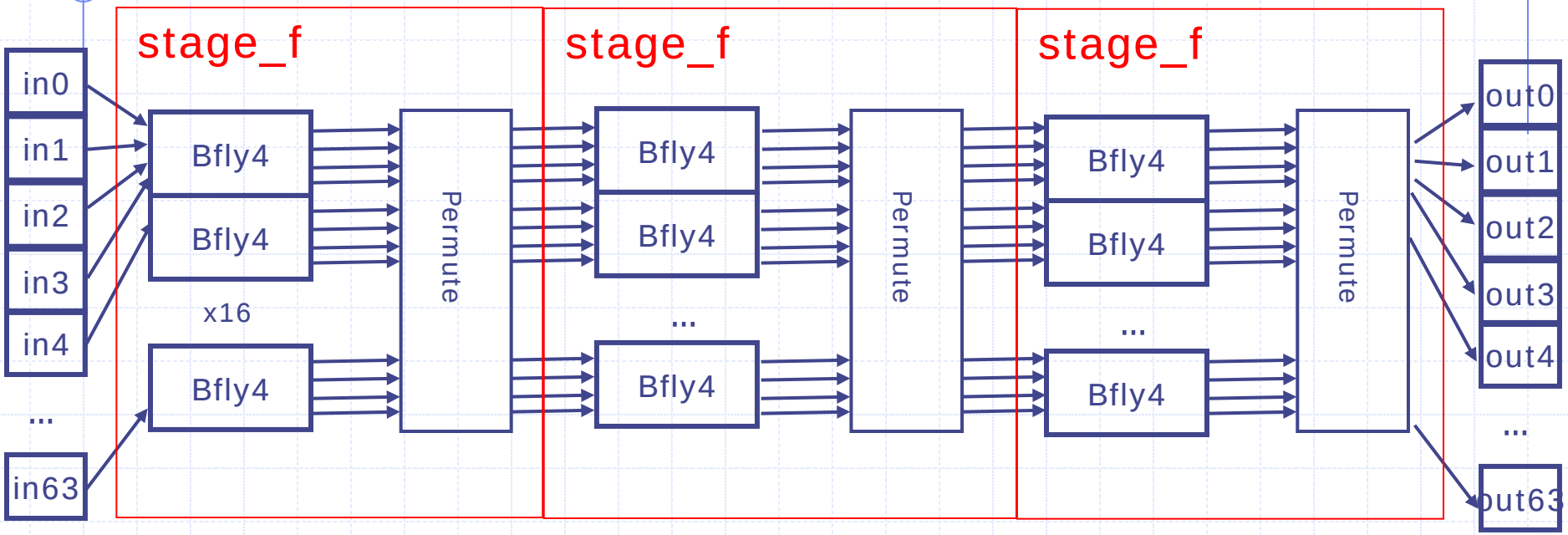
```
function Vector#(4,Complex) bfly4
            (Vector#(4,Complex) t,  Vector#(4,Complex) k);

  Vector#(4,Complex) m, y, z;

  m[0] = k[0] * t[0]; m[1] = k[1] * t[1];
  m[2] = k[2] * t[2]; m[3] = k[3] * t[3];

  y[0] = m[0] + m[2]; y[1] = m[0] – m[2];
  y[2] = m[1] + m[3]; y[3] = i*(m[1] – m[3]);

  z[0] = y[0] + y[2]; z[1] = y[1] + y[3];
  z[2] = y[0] – y[2]; z[3] = y[1] – y[3];

  return(z);
endfunction
```



Polymorphic code: works on any type of numbers for which *, + and - have been defined

Note: Vector does not mean storage

Stage_f Function



Code for stage_f

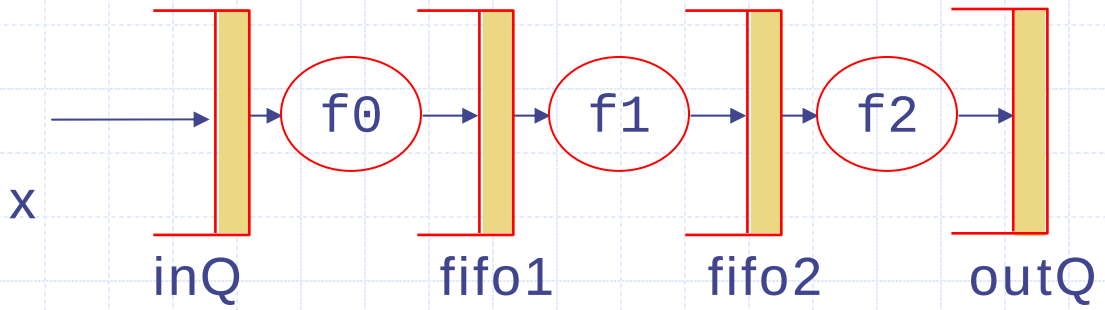
```
function Vector#(64, Complex) stage_f
        (Bit#(2) stage, Vector#(64, Complex) stage_in);
  begin
   for (Integer i = 0; i < 16; i = i + 1)
    begin
      Integer idx = i * 4;
      let twid = getTwiddle(stage, fromInteger(i));
      let y = bfly4(twid, stage_in[idx:idx+3]);
      stage_temp[idx]   = y[0]; stage_temp[idx+1] = y[1];
      stage_temp[idx+2] = y[2]; stage_temp[idx+3] = y[3];
    end
  //Permutation
   for (Integer i = 0; i < 64; i = i + 1)
      stage_out[i] = stage_temp[permute[i]];
   end
return(stage_out);
```

Code: Combinational IFFT

```
function Vector#(64, Complex) ifft
                        (Vector#(64, Complex) in_data);
   //Declare vectors
   Vector#(4,Vector#(64, Complex)) stage_data;
   //Define stage results
   stage_data[0] = in_data;
   for (Integer stage = 0; stage < 3; stage = stage + 1)
    stage_data[stage+1] = stage_f(stage,stage_data[stage]);
   return(stage_data[3]);
endfunction
```

Suppose we wanted to pipeline the IFFT…

Elastic pipeline



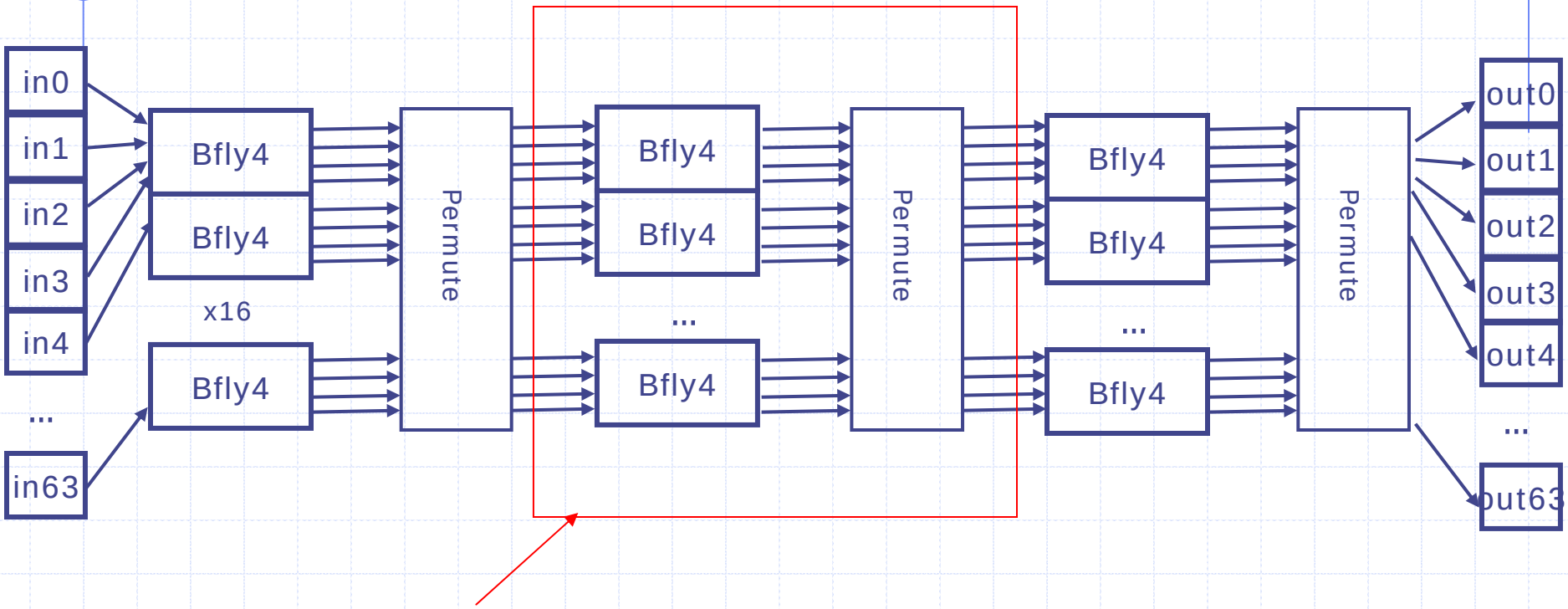

```
rule stage1 (True);
  fifo1.enq(f0(inQ.first());
  inQ.deq();        endrule
rule stage2 (True);
   fifo2.enq(f1(fifo1.first());
  fifo1.deq();      endrule
rule stage3 (True);
  outQ.enq(f2(fifo2.first());
  fifo2.deq();      endrule
```

Represent each stage as a rule

Firing conditions?

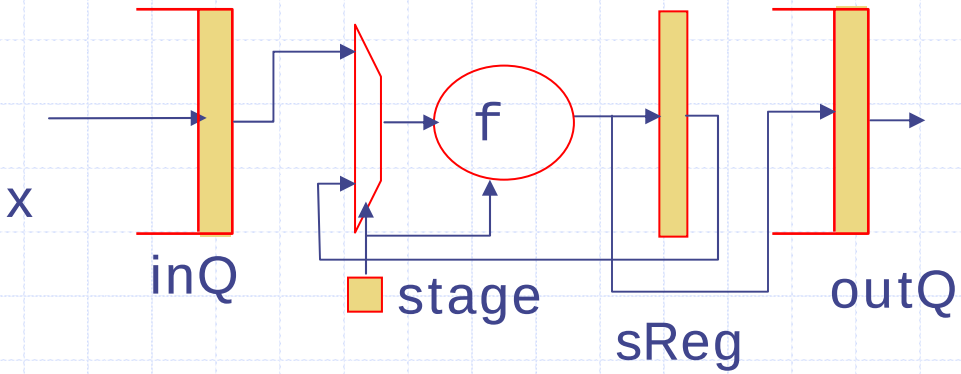
Can only fire when it has data and space in next buffer

Suppose we want to reduce the area



Reuse the same circuit three times

Folded pipeline




```
rule folded-pipeline (True);
 if (stage==0)
   begin sxIn= inQ.first(); inQ.deq(); end
 else     sxIn= sReg;
 sxOut = f(stage,sxIn);
 if (stage==n-1) outQ.enq(sxOut);
 else sReg <= sxOut;
 stage <= (stage==n-1)? 0 : stage+1;
endrule
```

Dilemma: Designing to the substrate

- HW generation is well understood and automatable
  - Combinational – single-cycle large combinational cloud
  - Elastic Pipeline – Full throughput pipeline
  - Folded Pipeline – Multi-cycle FSM
- SW generation from rule is new
  - Good conversions from combinational is trivial
  - Generating good code from the pipelines is a little tricky (selecting a schedule to factor out bookkeeping operations)
- In general different algorithms are more appropriate for hardware than software (or vice versa)
  - Designers should be aware of these tradeoffs and reason about them as a first-order concern

Outline

- Motivation
- Bluespec Codesign Language
- Compilation
- Partitioning
- Encapsulating the Bus
- Compilation Results
- Verification

Generating HW

- Each rule in the design is executed in a single clock cycle
- Convert each rule into (State ➔ Bool x State) function
- Greedy algorithm to select which rules to fire each cycle:
  - Static total ordering of rules
  - Must have a total ordering of rules executed
  - Considers port resources
- As good as hand-written RTL!

SW Compilation: Syntax Directed Scheme

- Each Module definition becomes a C++ class
  - Each BCL method/rule is a class method
  - Methods to create and merge shadow copies (parallel and sequence)
- Lazy STM-style conversion of rules
  - Keep a mapping to current notion of state which is committed when complete
  - Actions translate to C++ statements
  - Expressions to an C++ expression and a statement which must be evaluated before expression is valid
  - Rule translate to methods which return a boolean signifying success of execution
- Runtime instantiates top-level design and decides how run our rule procedures

SW Optimizations

- Sequentialization – Convert Parallel compositions to sequential ones
- Guard Lifting – Exit early on failing executions
- Shadow Minimization – higher granularity shadow generation reduces copying
- Shadow Reuse – Reuse the same shadow across different rule executions

Future Work: Scheduling

- Choice of what order to execute rules is a key decision for efficiency
- In HW rules execute in parallel ("free" parallelism)
- In SW, we must balance between:
  - Locality – Merge rules into larger rule (StreaMIT-style synchronous data flow)
  - Thread Parallelism – pipeline parallelism
- Lots of work remaining (Myron King's PhD)

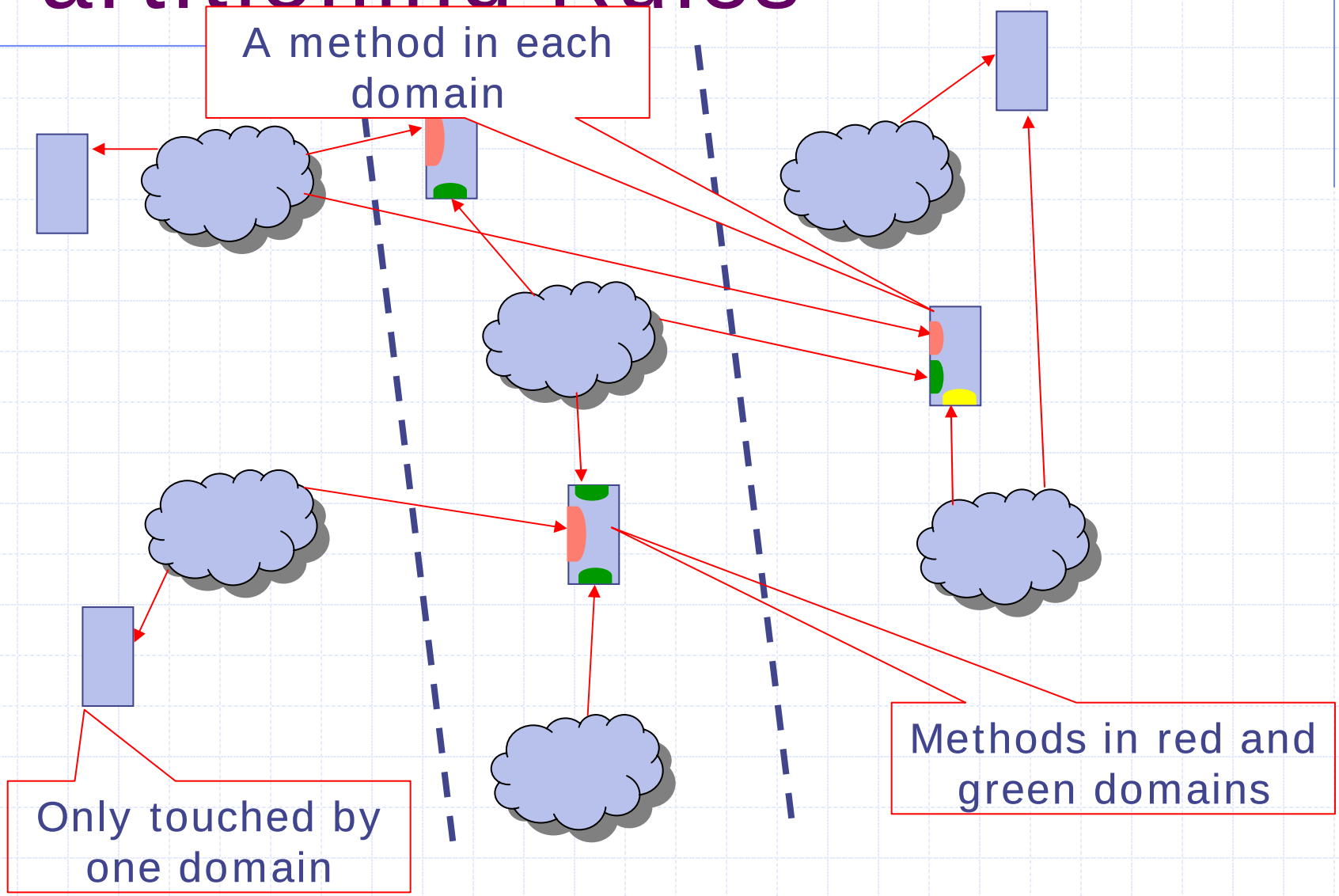
Outline

- Motivation
- Bluespec Codesign Language
- Compilation
- Partitioning
- Encapsulating the Bus
- Compilation Results
- Verification

Computational Domains

- Any expression/action can be implemented in either HW or SW
- To ease implementation burden and understandability we insist that each rule operates in a single domain
  - Same follows for methods and expressions

Partitioning Rules



We need primitive domain-crossing modules

- Simple directional channel
  - enq in one domain and deq/first/pop in another
  - full/empty signals are conservative approximations (e.g. may not be full when full signal is true)
  - Can be modeled precisely by 3 FIFOs in sequence
- Other interfaces are possible
  - e.g. split-phase memory

We call such modules synchronizers

Partitioning the Ogg Vorbis Example

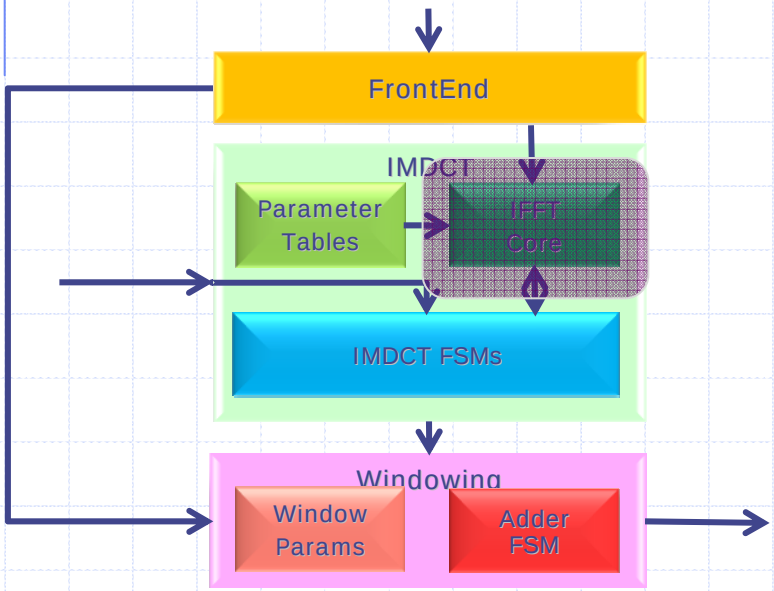
Ogg Vorbis Backend code

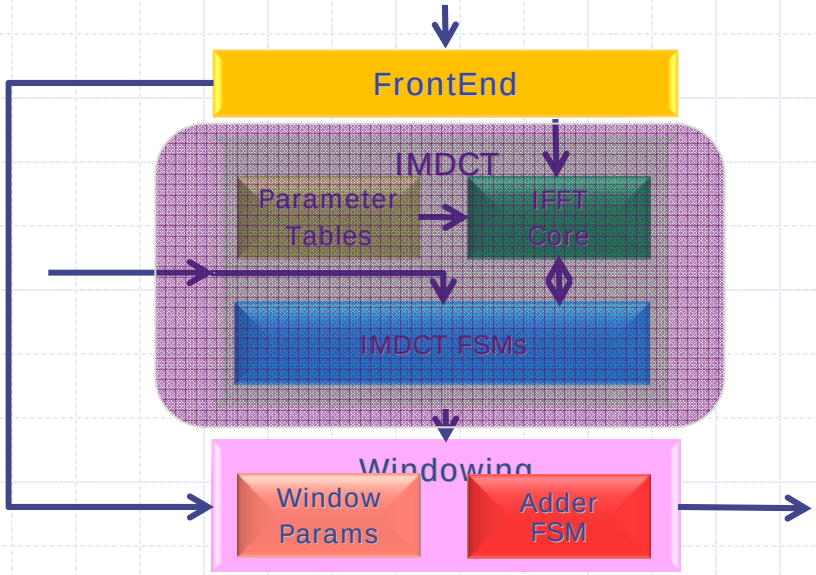
```
//State
FIFO#(IMDCTReqData) toimdctQ <- mkFIFO;
IFFT  ifft   <- mkIFFT(); Windower window <- mkWindow;
Reg   icnt   <- mkReg(0);  Reg   ocnt   <- mkReg(0);
Array iFrame <- mkArray(); Array oFrame <- mkArray();
//Rules & Methods
method addData(x) = toimdctQ.enq(x);
rule setConfiguration;…ifft.config…window…
rule putFrameInIFFT when (icnt < n);
  ifft.inputData(iframe[icnt]);
  icnt <= icnt+1;
rule getFrameFromIFFT when (ocnt < n);
  rv <- ifft.outputData();
  oframe[ocnt] <= rv;
  ocnt <= ocnt+1;
rule enterWindow(ocnt == n);
  window.enter(oframe); ocnt <=0;
method pcmoutput() = window.getOutput();
```

How do we partition this?

Partitions

- Two partitions
  - HW IFFT, SW IMDCT FSM & Windower
  - HW IMDCT (IFFT & FSM), SW Windower





Partitioning the Design IFFT in HW

Red is HW

```
//State
FIFO#(IMDCTReqData) toimdctQ <- mkFIFO;
IFFT  ifft   <- mkIFFT(); Windower window <- mkWindow;
Reg   icnt   <- mkReg(0);  Reg   ocnt   <- mkReg(0);
Array iFrame <- mkArray(); Array oFrame <- mkArray();
//Rules
method addData(x) = toimdctQ.enq(x);
Rule setIMDCTData;
rule setConfiguration;…ifft.config
rule putFrameInIFFT when (icnt < n);
  ifft.inputData(iframe[icnt]);
  icnt <= icnt+1;
rule getFrameFromIFFT when (ocnt < n);
  rv <- ifft.outputData();
  oframe[ocnt] <= rv;
  ocnt <= ocnt+1;
rule enterWindow(ocnt == n);
  window.enter(oframe); ocnt <= 0;
method pcmoutput() = window.getOutput();
```

Fails our 1 domain per rule restriction

Making Single Domain Rules IFFT in HW

```
Synchronizer#(2,1) sync <- mkBusSynchronizer(SW, HW);
rule sendDataToIFFT when (icnt < n);
  let x <- sync.toHW[0].put(iframe[icnt]);
  icnt <= icnt+1;
rule putFrameInIFFT;
  let x <- sync.toHW[0].get();
  ifft.inputData(x);
rule sendDataFromIFFT();
  let v <- ifft.outputData();
  sync.toSW[0].put(v);
rule getFrameFromIFFT when (ocnt < n);
  rv <- sync.toSW[0].get();
  oframe[ocnt] <= rv; ocnt <= ocnt+1;
rule setConfigurationSW …
  … sync.toHW[1].put(…)… window.config(…)
rule setConfigurationHW …
  … sync.toHW[1].get(…)… ifft.config(…)
```

- Add bidirectional synchronizer (2 virtual channels SW to HW, 1 HW to SW)
- Split rules to fix domain restrictions

Partitioning the Design IMDCT in HW

```
//State
FIFO#(IMDCTReqData) toimdctQ <- mkFIFO;
IFFT  ifft   <- mkIFFT(); Windower window <- mkWindow;
Reg   icnt   <- mkReg(0);  Reg   ocnt   <- mkReg(0);
Array iFrame <- mkArray(); Array oFrame <- mkArray();
//Rules
method addData(x) = toimdctQ.enq(x);
rule setConfiguration;…
rule setIMDCTData;…
rule putFrameInIFFT when (icnt < n);
  ifft.inputData(iframe[icnt]);
  icnt <= icnt+1;
rule getFrameFromIFFT when (ocnt < n);
  rv <- ifft.outputData();
  oframe[ocnt] <= rv;
  ocnt <= ocnt+1;
rule enterWindow(ocnt == n);
  window.enter(oframe); ocnt <=0;
method pcmoutput() = window.getOutput();
```

Breaks Domain Restriction

Making Single Domain Rules IMDCT in HW

Red is HW

```
//State
Synchronizer#(2,1) sync <- mkBusSynchronizer(SW, HW);

rule setIMDCTDataSW(hasDatainQ)
  imdctQ.deq(); sync.toHW[1].put(imdctQ.first);
rule setIMDCTDataHW;…
  let x <-  sync.toHW[1].get(…);
  iframe[idx] = …
rule setConfigurationSW(hasConfiginQ);…
  sync.toHW[1].put(…);
rule setConfigurationHW;…
  sync.toHW[1].get(…);
rule enterWindowHW(ocnt == n);
 sync.toSW[0].put(oframe); ocnt <= 0;
rule enterWindowSW;
 let x <- sync.toSW[0].get();
 window.enter(x);
```

Split different rules but same procedure

Making Single Domain Rules

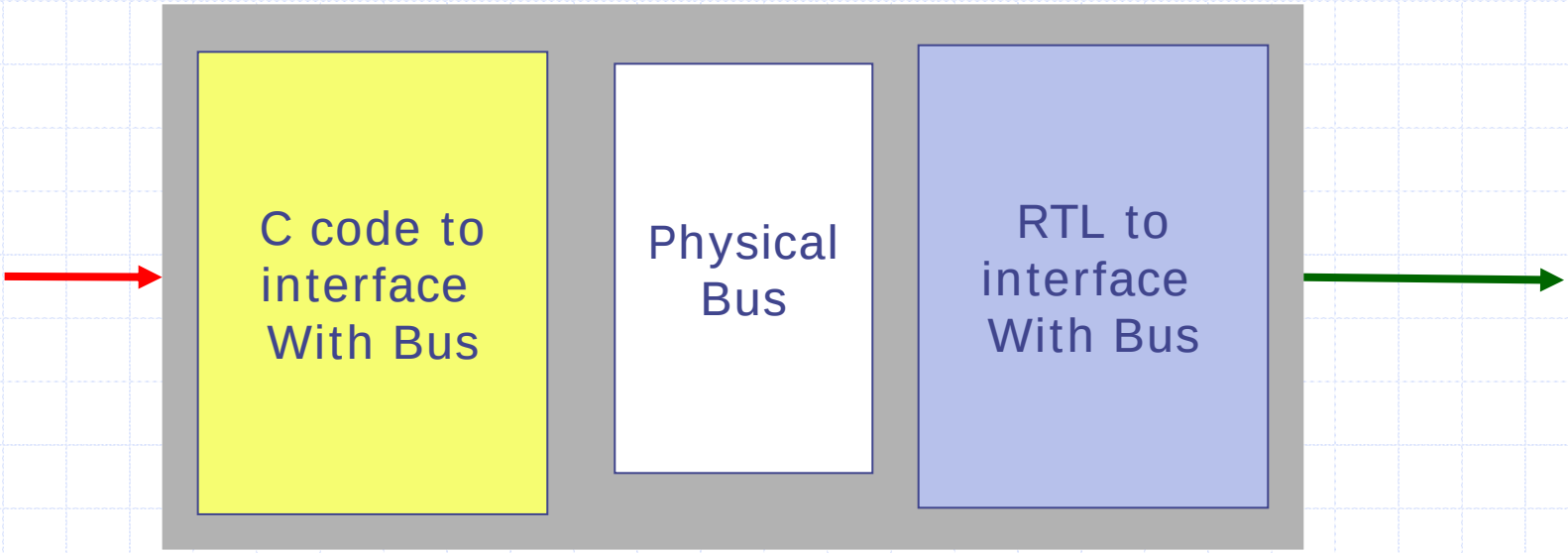
```
Synchronizer#(2,1) sync <- mkBusSynchronizer(SW, HW);
rule sendDataToIFFT;
  let x <- sync.toHW[0].put(iframe[icnt]);
  icnt <= icnt+1;
rule putFrameInIFFT when (icnt < n);
  let x <- sync.toHW[0].get();
  ifft.inputData(iframe[icnt]);
rule sendDataFromIFFT();
  let v <- ifft.outputData();
  sync.toSW[0].put(v);
rule getFrameFromIFFT when (ocnt < n);
  rv <- sync.toSW[0].get();
  oframe[ocnt] <= rv;
  ocnt <= ocnt+1;
rule setConfigurationSW …
  … sync.toHW[1].put(…)… window.config(…)
rule setConfigurationHW …
  … sync.toHW[1].get(…)… ifft.config(…)…
```

Same idea. Different rules need to be split

Outline

- Motivation
- Bluespec Codesign Language
- Compilation
- Partitioning
- Encapsulating the Bus
- Compilation Results
- Verification

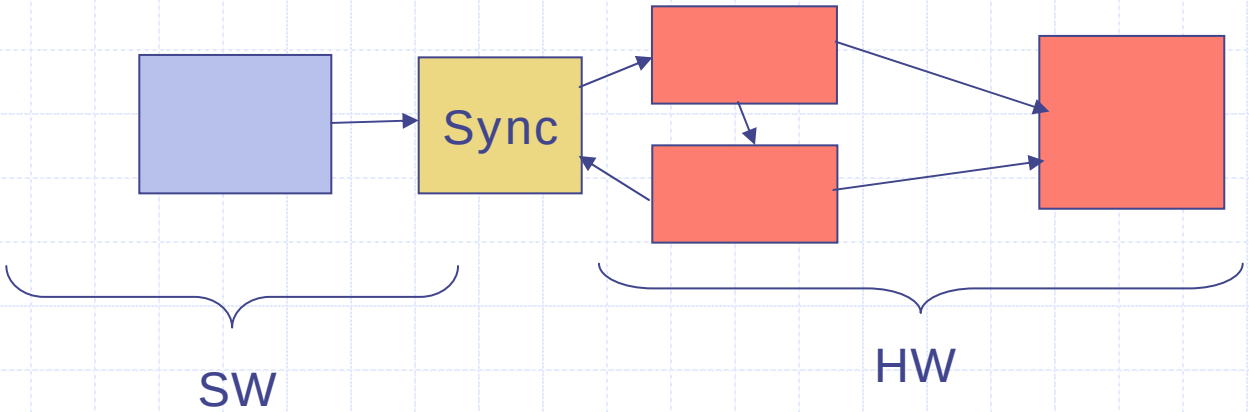
Implementing a Bus-based Synchronizer



This is a natural abstraction for the communication channel

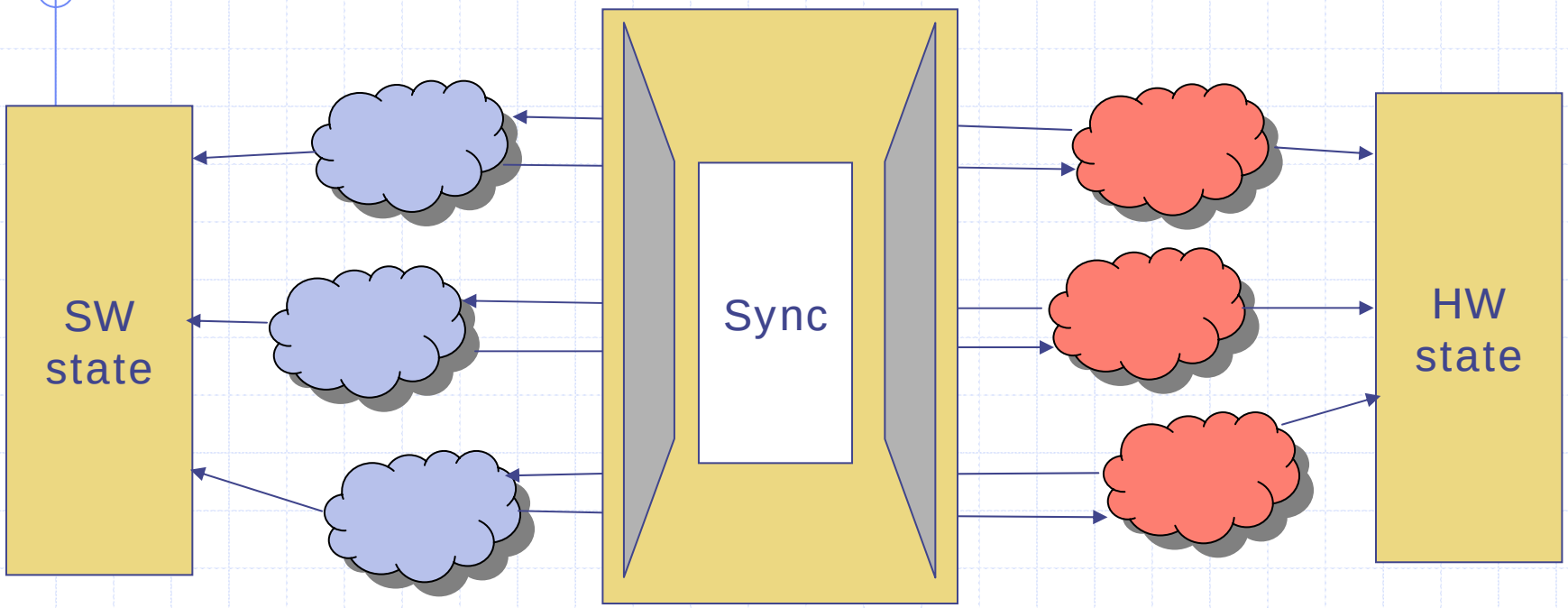
May be complicated by burst reads/writes, Random access memory-like communication, failing writes, multiplexing onto same bus.

Retargeting Design



- Initial: “Embedded FPGA” design
  - HW: FPGA
  - SW: on FPGA PowerPC
  - Communication via Processor Local Bus (PLB)
- New: Accelerator for General Purpose Processor
  - HW: FPGA
  - SW: On CPU
  - Communication via PCI-Express
- All design specific aspects limited to synchronizer
  - Easy to replace. Some variations due to bursts, etc.

Making it more Complicated: Sharing a Bus Interface



- 3 bus interfaces on the same bus is generally not reasonable
  - Large HW cost, redundant
- User can multiplex onto a single synchronizer
  - We can represent the Multiplexing logic in BCL directly
  - Query: How does this affect performance/correctness?

Outline

- Motivation
- Bluespec Codesign Language
- Software Compilation
- Partitioning
- Encapsulating the Bus
- Compilation Results
- Verification

Ogg Vorbis Implementations: Platforms

- FPGA Accelerator
  - HW: XUPv5-LX110T FPGA
  - SW: 2.8GHz Nehalem Westmere CPU with 3GB of RAM
  - Communication: PCI-Express using the Standard Co-Emulation and Modeling Infrastructure (SCE-MI)
- Embedded System:
  - HW: XUPv5-LX110T FPGA
  - SW: Microblaze softcore implemented in FPGA
  - Communication: point-to-point communication channels called the Fast Simplex Links (FSLs).

Results

FrontEnd	SW					
IMDCT	HW		Hybrid		SW	
Windowing	HW	SW	HW	SW	HW	SW
FPGA Accelerator Platform						
Speed (s)	8.9	28.1	84.9	102	316	38.9
FPGA (Regs)	36%	32%	40%	36%	34%	0
FPGA (Slices)	22%	22%	20%	21%	23%	0
Embedded FPGA Platform						
Speed (s)	114	231	414	424	876	896
FPGA (Regs)	36%	35%	38%	37%	35%	33%
FPGA (Slices)	36%	35%	33%	34%	37%	39%

All done in a matter of minutes!

Software Overhead

- The rule-based abstraction introduces overhead:
  - Overhead of running incorrect rules
  - Shadowing state for rule which may fail partway through execution
- 4 implementations of a Ray Tracer:
  - Hand coded C++ software model (1x)
  - Our automatically generated HW/SW design (1.7x)
  - Automaticallt generated HW/SW with ideal schedule (1.2x)
  - Handtuned schedule on generated design and optimized to remove unneeded shadowing (**~1.01x**)
- Initial performance results are promising.

Outline

- Motivation
- Bluespec Codesign Language
- Software Compilation
- Partitioning
- Encapsulating the Bus
- Compilation Results
- Verification

The problem with Verification

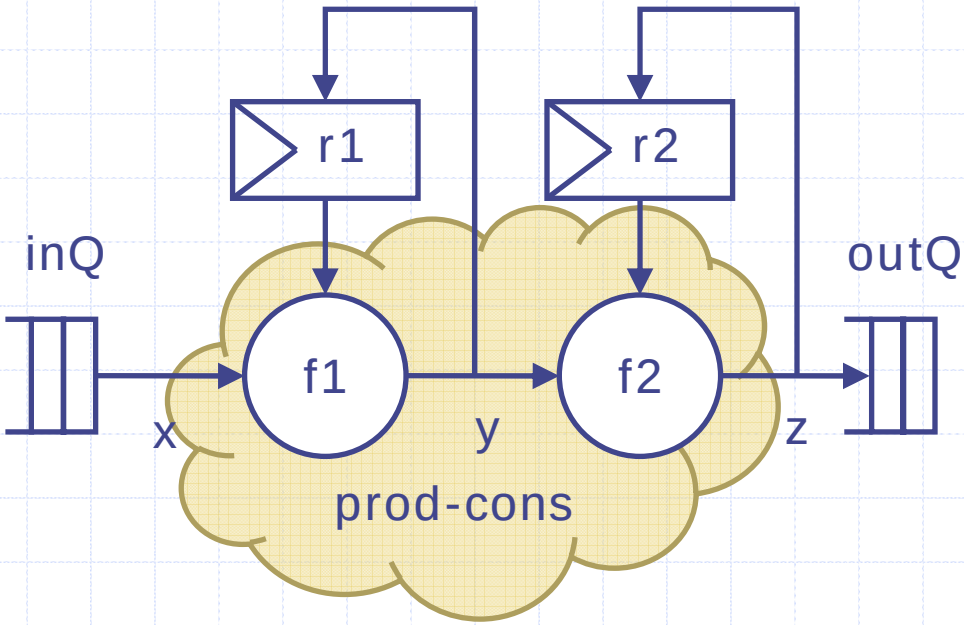
- Formal abstractions are too complicated for most users
  - Temporal logic spec of a FIFO
- Most successes from models with simple abstractions & modularity
  - Type checking
  - Language-level equivalence (FSM)
- Leverage these in the BCL context

BCL Verification

- Consider the equivalence of BCL programs
  - Can be converted to module equivalence
- BCL is inherently nondeterministic at this level
  - Can represent more “spec”-like behavior in the language itself
  - Intuitively, we expect to find errors with shorter number of rules executions (minimal trace)
- All the awkwardness of HW/SW timing is abstracted
  - Safe over approximation
  - Can reach closer to the realized schedule in implementation by representing scheduling as program transform

KEY: Verifying the actual design

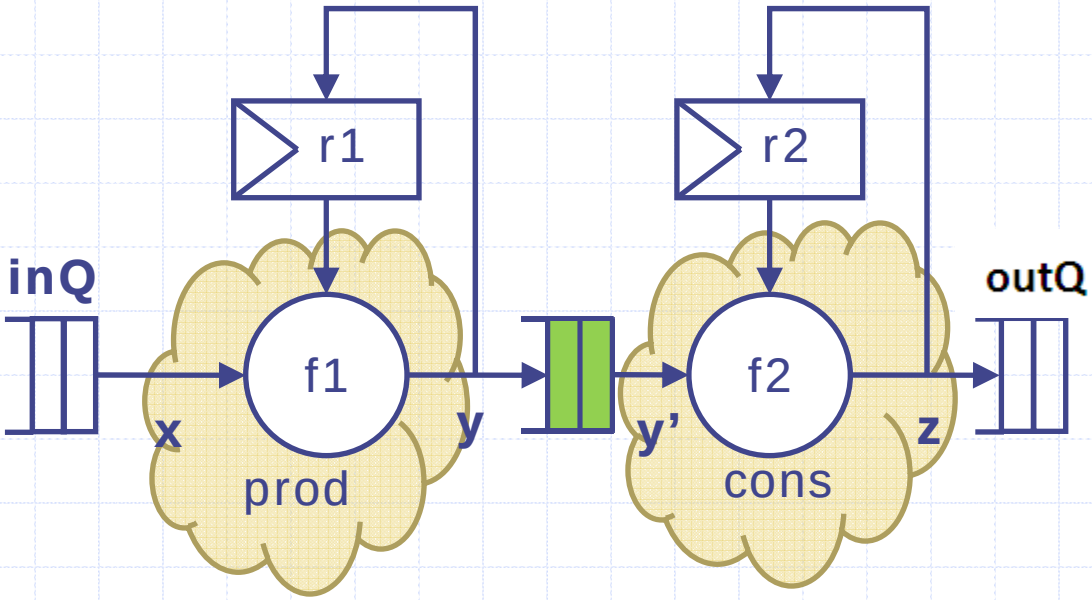
A rule-based description



register r1 = 0, r2 = 0
fifo inQ, outQ

```
rule producer-consumer when (!inQ.empty && !outQ.full):
    let x = inQ.first;
    let y = f1(x,r1);
    let z = f2(y,r2);
    r1 := y; r2 := z
    outQ.enq(z); inQ.deq;
```

Rules for the Refined System



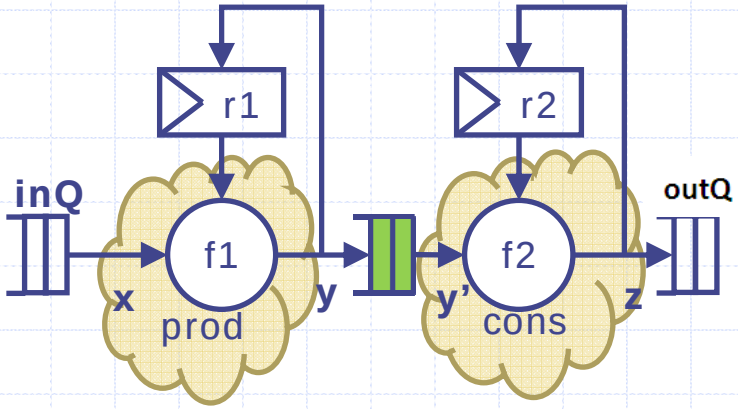
register r1 = 0, r2 = 0
fifo q, inQ, outQ

```
rule produce when
        (!q.full && !inQ.empty):
    let x = inQ.first;
    let y = f1(x,r1);
    q.enq(y); inQ.deq;
    r1 := y
```

```
rule consume when
        (!q.empty && !outQ.full):
    let y = q.first;
    let z = f2(y,r2);
    outQ.enq(z); q.deq;
    r2 := z;
```

Possible executions of the two rule system

```
rule produce when
        (!q.full && !inQ.empty):
    let x = inQ.first;   let y = f1(x,r1);
    q.enq(y); inQ.deq; r1 := y;

rule consume when
        (!q.empty && !outQ.full):
    let y = q.first;   let z = f2(y,r2);
    outQ.enq(z); q.deq; r2 := z;
```



Some schedules worth considering:

prod; cons; prod; cons; prod; cons;...

prod; prod; cons; prod; cons; prod; cons;...

prod; prod; cons; cons; prod; prod;...

In what sense are these two systems the same?

```
rule producer-consumer when (!inQ.empty && !outQ.full):
    let x = inQ.first;
    let y = f1(x,r1);
    let z = f2(y,r2);
    r1 := y; r2 := z;
    outQ.enq(z); inQ.deq;
```

```
register r1 = 0, r2 = 0
inQ, outQ
```

Original System ↑

Refined System ↓

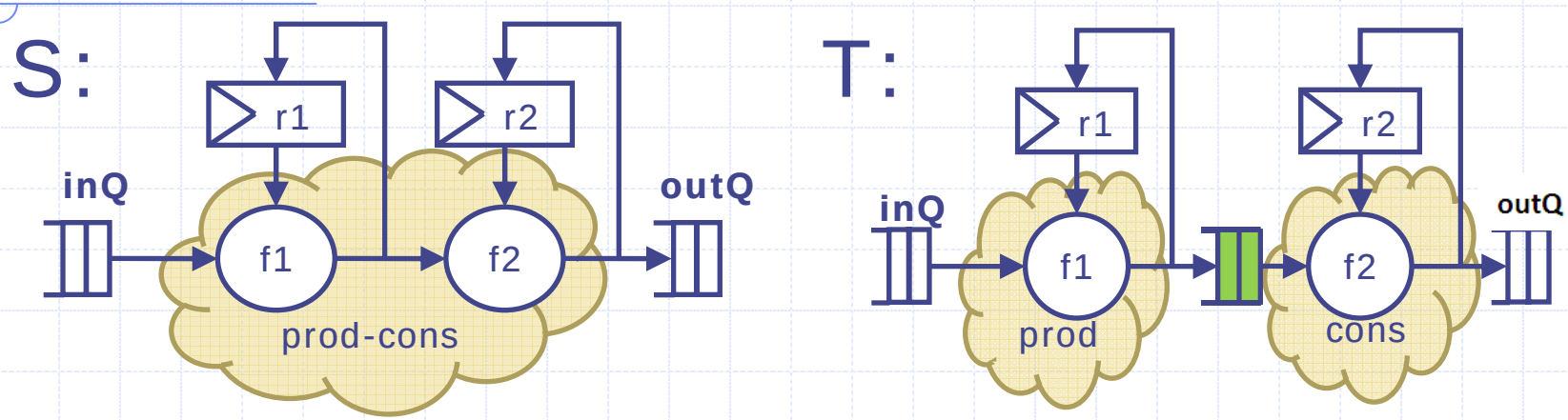
same set of state transitions ?

```
register r1 = 0, r2 = 0
fifo q, inQ, outQ
```

```
rule produce when
        (!q.full && !inQ.empty):
    let x = inQ.first;
    let y = f1(x,r1);
    q.enq(y); inQ.deq;
    r1 := y
```

```
rule consume when
        (!q.empty && !outQ.full):
    let y = q.first;
    let z = f2(y,r2);
    outQ.enq(z); q.deq;
    r2 := z;
```

When are states equivalent?

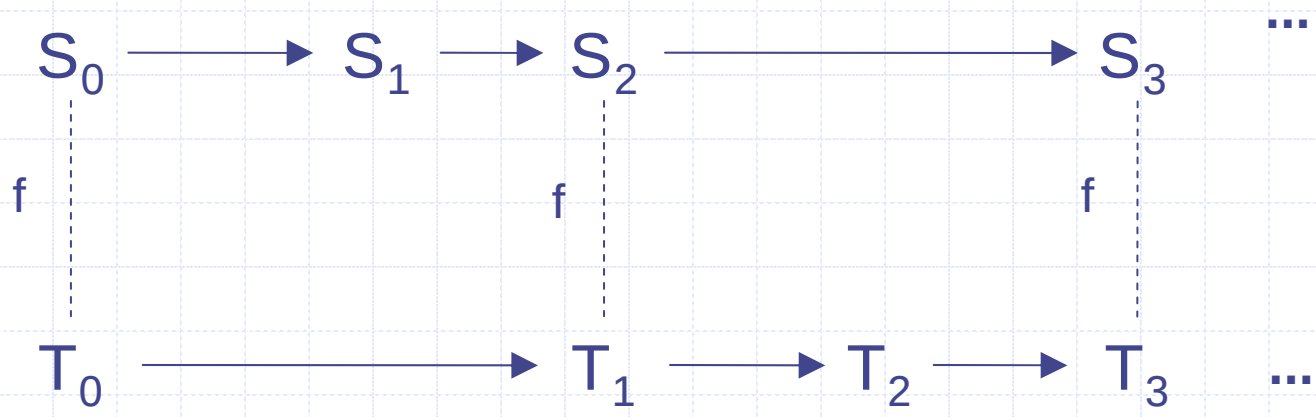


- If q is empty: the remaining state matches
- If q is non-empty: many choices (Run cons, Undo prod, etc.) Tricky to encode in function
- Practically, user can only give a partial condition

In our example: (f: T -> S) elides q and is defined only when q is empty.

Equivalence: Intuition

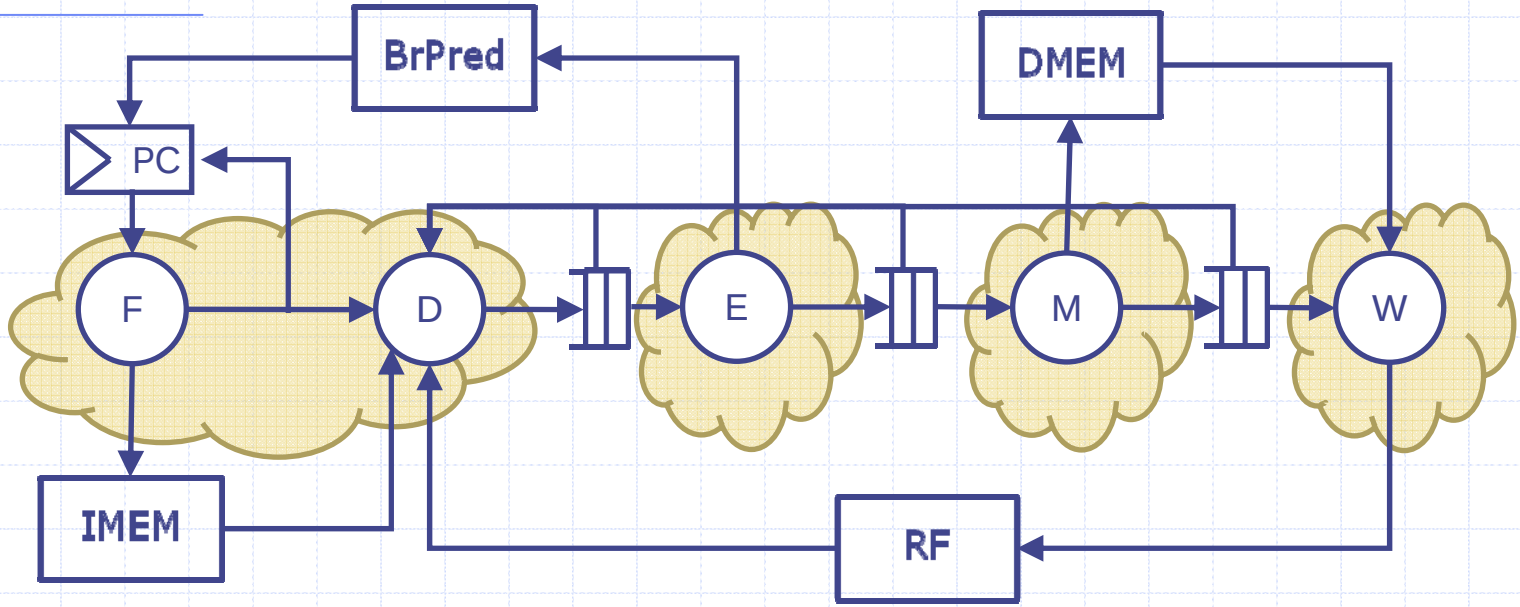
- Programs S and T equal if for each schedule of S, there's a schedule of T where each state that could match, does.
  - (i.e. if S reaches a state with a match, then T reaches the corresponding state and vice versa)



Automatic Verification

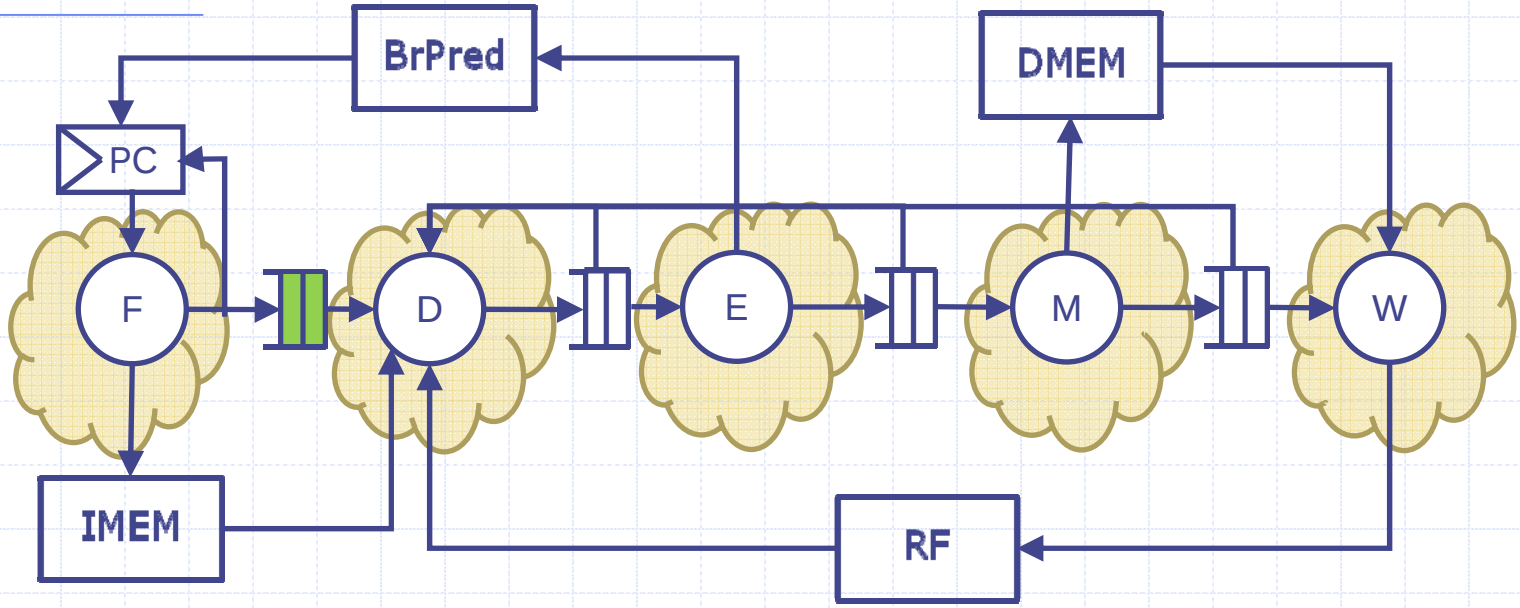
- This can be verified by coinduction on the schedules
  - Find finite prefix cover for schedules of T which has a correspondence to a schedule in S
  - Can be reduced to relatively small set of simple SMT queries

A more interesting example: 4-stage Pipeline



- Speculative w/ multi-cycle memory
- Unpipelined: (FD;E;M;W)*
- Pipelined: (W;M;E;FD)*

Split Fetch and Decode



- Initial tool verifies this refinement in a matter of minutes
  - 21 queries considering length 3 prefixes
  - Can be made much faster (>10x)

Summary

- BCL provides:
  - A clean abstraction allowing HW and SW to be unified
  - Has enough precision in result to allow the user to partition, evaluate, and tune a design easily.
    - HW is already there, SW is within reach
  - Provides good abstraction for verification purposes

Thanks

ndave@csl.sri.com