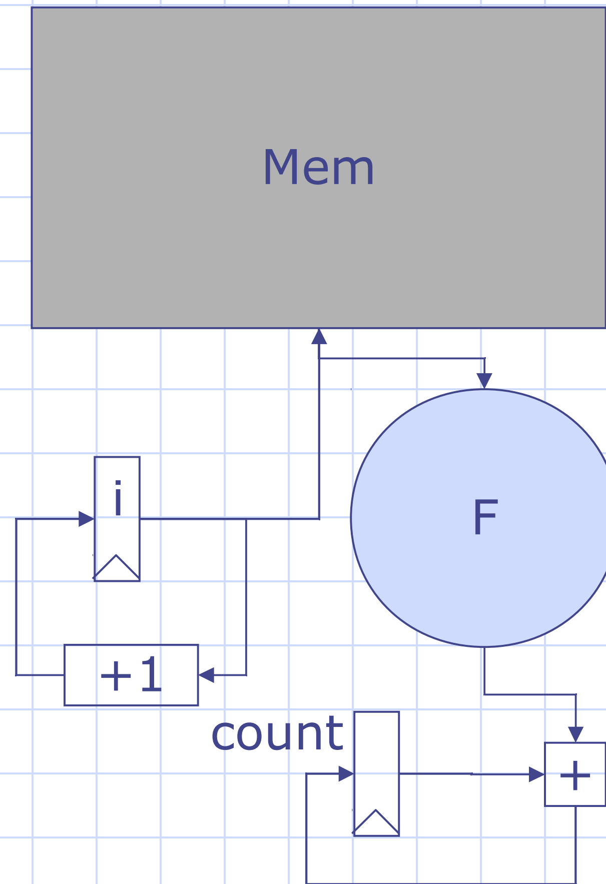
Checking Modular Refinements of Bluespec

Nirav Dave[1], Michael Katelman[2]
Massachusetts Institute of Technology[1]
University of Illinois at Urbana-Champaign[2]

Frequently BSV designers refine their designs

```
rule map(i<100);
  i <= i + 1;
  count <= count +
   f(mem.read(i));
```

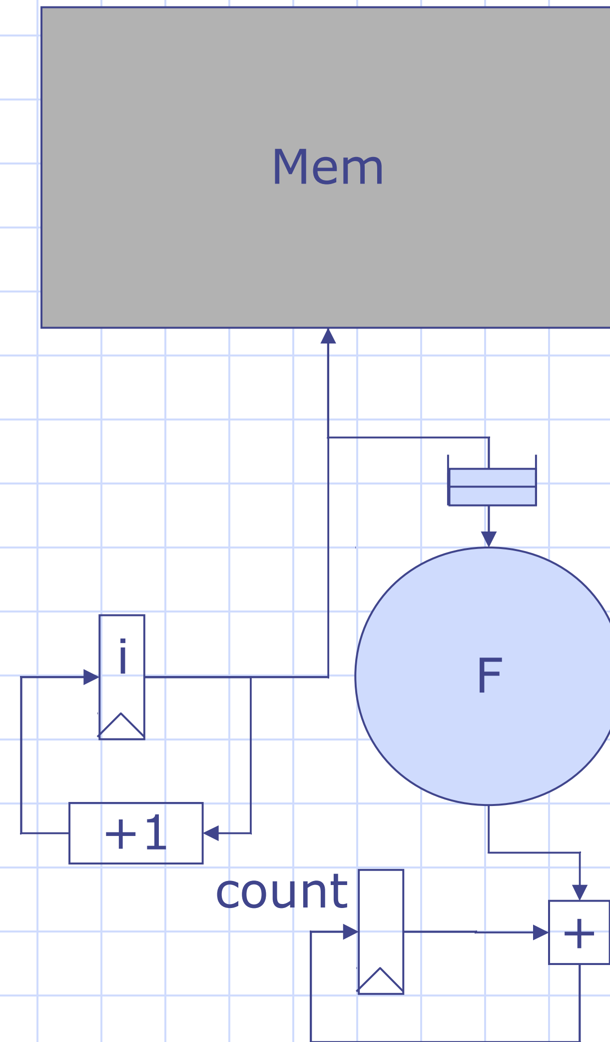


Refinement: Split lookup and modify

```
FIFO#(int) tempQ <- mkFIFO;

rule mapReq(i < 100);
 i <= i + 1;
 tempQ.enq(mem.read(i));

rule mapResp(True);
 count <= count +
      f(tempQ.first());
 tempQ.deq();
```



Pipelined! Better hardware, but is it correct?

Correctness depends on context

i[0], c[0]		i[1], c[1]		i[2], c[2]		…
i[0]	c[0]	i[1]	c[1]	i[2]	c[2]	…
i[0]	i[1]	c[0]	i[2]	c[1]	c[2]	…

```
rule bad(True);
 if (p(count))
  $display(i,count);
```

```
rule good(tempQ.empty);
 if (p(count))
  $display(i,count);
```

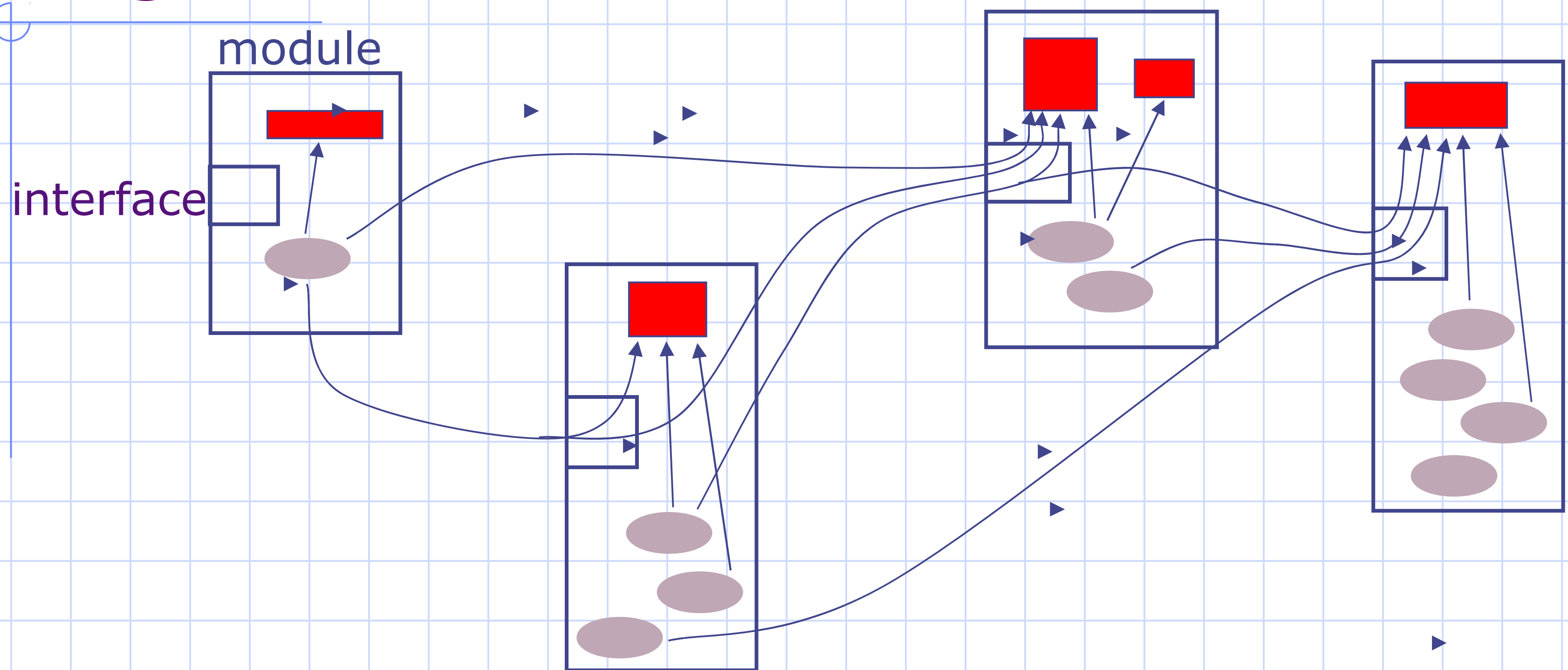
- New design can observe partially updated state. Rest of system Can't count on i and count to be in sync
- If we were given the whole design can we say if this is okay?

Can a tool solve this?

- At least for a reasonable class of refinements
  - Convert BSV design to TRS
  - Translate BSV rules in to pure functions
  - Use functions to form queries to an bitvector SMT solver

First a bit more about the language

Bluespec: 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

Rules can manipulate state in other modules only *via* their interfaces.

Rule: As a State Transformer

A rule may be decomposed into two parts $\pi(s)$ and $\delta(s)$ such that

$$s_{next} = \textit{if } \pi(s) \textit{ then } \delta(s) \textit{ else } s$$

$\pi(s)$ is the guard (predicate)

$\delta(s)$ is the "state transformation" function, i.e., computes the next-state values from the current state values

Execution model

Repeatedly:

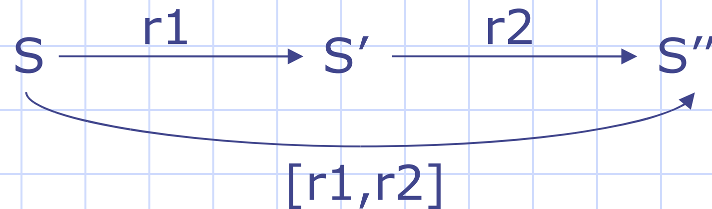
- Select a rule to execute
- Compute the state updates
- Make the state updates

Highly non-deterministic

- Compilation involves deciding how we select rules
  - Multiple rules in a cycle
  - Tradeoff between parallelism and cycle-level depth
  - A lot of flexibility in choice

Rule Traces

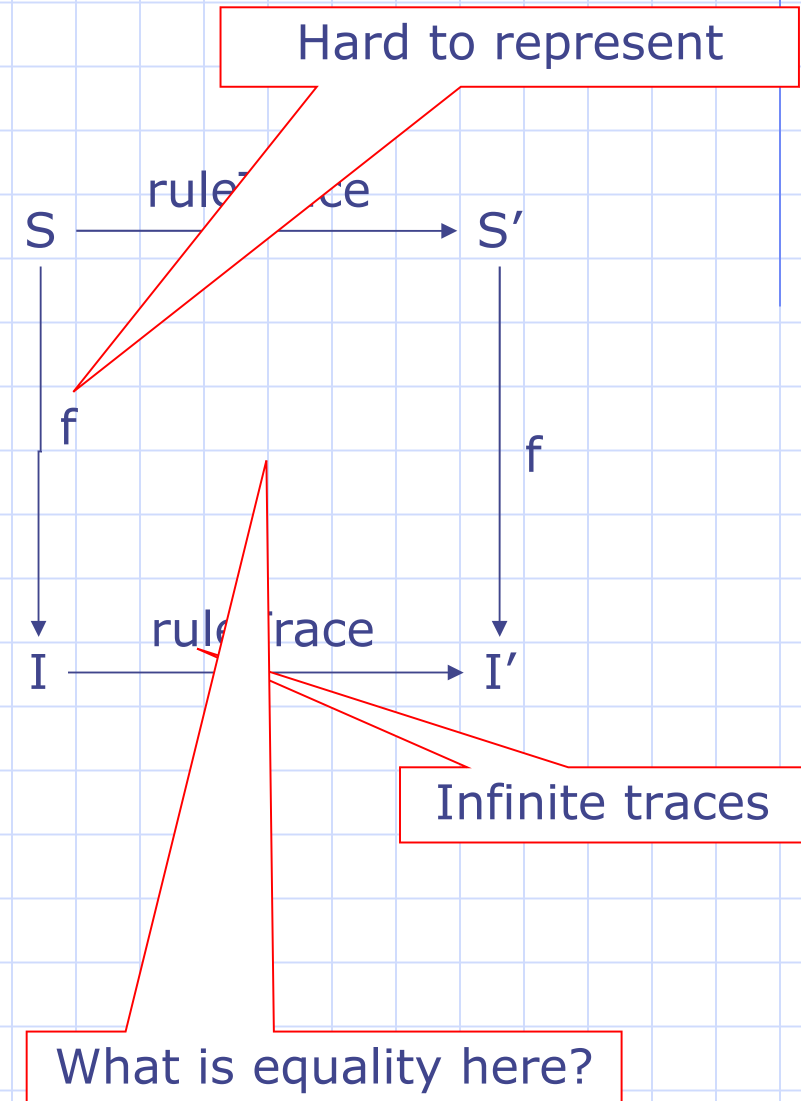
- Rules takes us from State to State



- A rule trace is a sequence of rules, executed in order
  - run(t,s) runs trace t on initial state s
  - Like rules may not be valid to apply guard to a state (a rule in the chain fails)

The Query

- Completeness: Every rule trace in the Spec has a corresponding trace in the Implementation
- Soundness: Every rule trace in the implementation has a corresponding trace in the Spec



Handling Infinite Traces

- There are an infinite # of traces
- Can we just handle a finite set of finite traces?
  - If we have a prefix of a trace, we can reduce the problem to solving it for the tail
    - If [A,B,C] is fine, then [A,B,C,D,E] reduces to [D,E]
  - If we have a prefix cover for all possible traces of sufficient size, we can always make progress

Handling Infinite Traces

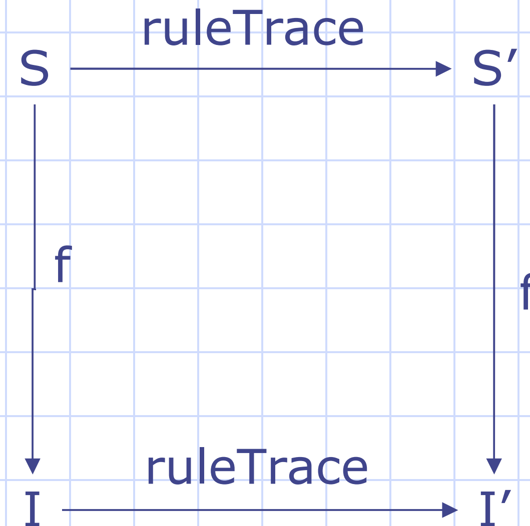
- Still may need infinite prefix traces
  - May never reach a comparable state
- Show prefix is equivalent to a “safe” trace + a smaller prefix
  - If [A,B] is safe, and we can show [A,C,D] is the same as [A,B,E], we reduce to [E].
- Can get away with considering finite traces

Algorithm to find prefix cover

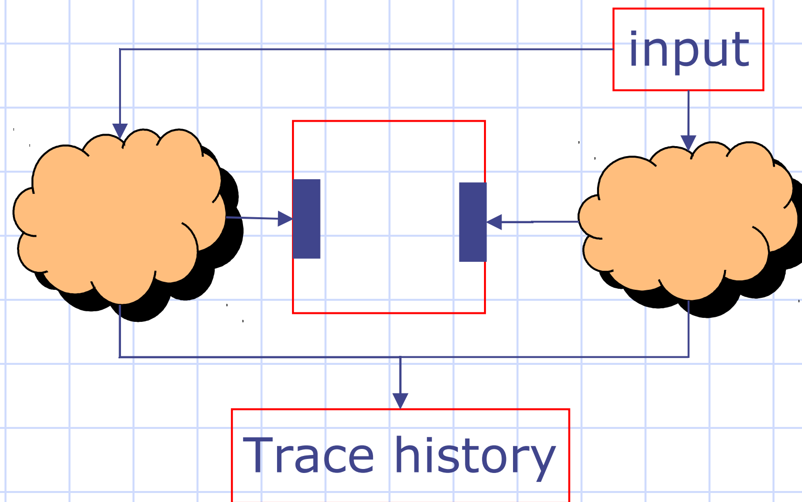
- Start with T = traces of length 1
- Repeatedly:
  - Remove smallest t from T
  - Check if we always represent t using safe traces (had a matching point to a spec trace)
  - If not add extend t with all possible 1 rule prefix and add to T
- Bail after some size N
  - Remaining traces are interesting to designers

Trace equality

- We wanted the traces to be "equivalent"
- For modular refinement it's what we can observe about the module
  - Method calls – existence & output



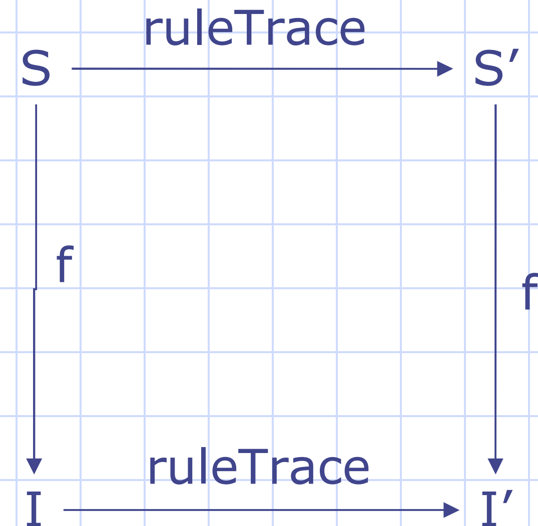
Method Calls to State



- All visible history stored in trace
  - Just look at history for equivalence
- Need to consider all input systems

Relating States

- Simplification: We only add state between Spec and Implementation
- Relation from Spec to Implementation clear
  - Add new state in initial state
- Leave Implementation to Spec partial
  - Reason about longer rule traces



More simplification

- Only consider systems where break one rule into two
  - Rules in Spec: r12,r3,r4,r5, …
  - Rules in Impl: r1, r2, r3′,r4′,r5′,…
- r12 should correspond to {r1,r2}
- Makes completeness easy to prove

Queries

- Each question takes the form:

$$\forall i \in I, s \in S(i).\ \text{isSpecState}(s) \Rightarrow \text{run}(t, s) \in \{\text{run}(t', s) \mid t \in T\}$$

  - I is the set of possible input
  - t is trace we're considering
  - T is the set of "safe" traces we want to check against

- This is easy to cast in SAT

Reducing the number of Queries

- We can find impossible rule traces:
  - Many rules cannot fire twice concurrently (FIFOs fill up)
  - e.g. [req,req,req]) is impossible
- Many rule sequences are equivalent:
  - e.g. Rules don’t touch same state
  - Do not have to check traces T1+[A,B]+T2 since we’ll check T1+[B,A]+T21

Current Status

- Simple simple programs : 4 rules
  - Correct design: 10 seconds (N = 7)
  - Added an error: 2 seconds (N = 4)

- 6 stage SMIPS pipeline
  - refine to 7 stage (N = 14)
  - Many Days of compute

Improvements

- Trace verification is ridiculously parallel
  - Parallel execution
- Currently, we Represent state as a bitvector
  - Does not scale (especially Memories)
  - Should move to uninterpreted functions/arrays
- Call SMT via file system (write file)
  - Significant overhead (>50%)
  - Direct interfacing significantly cheaper

Summary

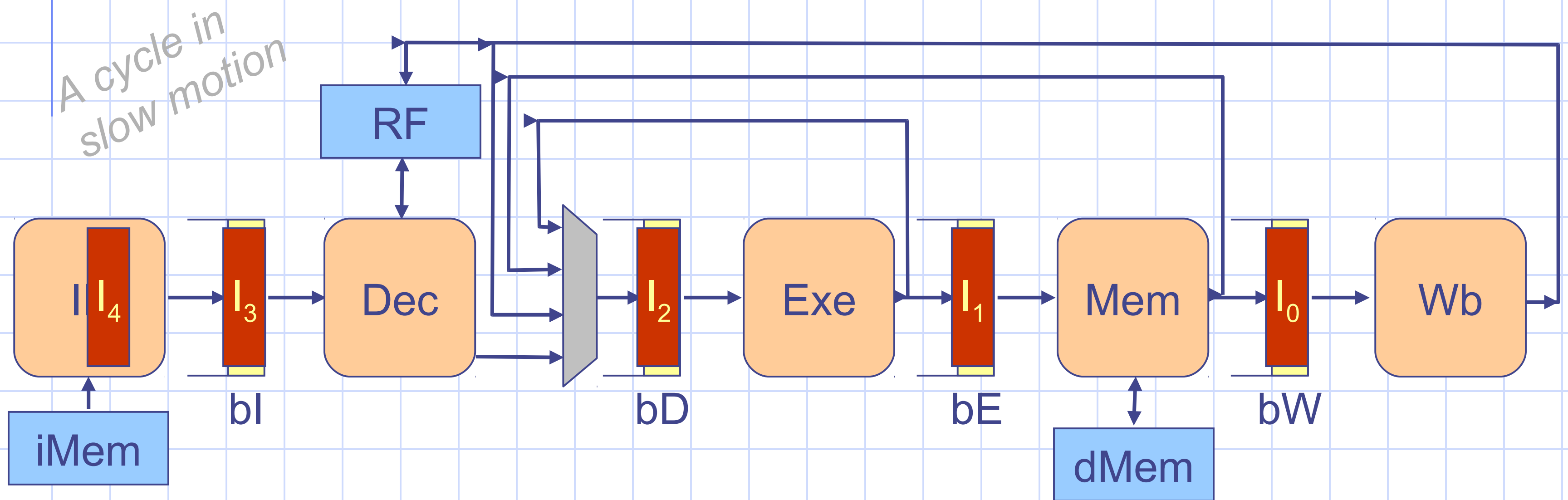
- Can answer interesting questions about traces in BSV systems
- Initial implementation seems pretty reasonable
- Efficiency improvements needed to be practical

The End

Scheduling Flexibility

- *What order do we want?*

Wb < Mem < Exe < Dec < IF

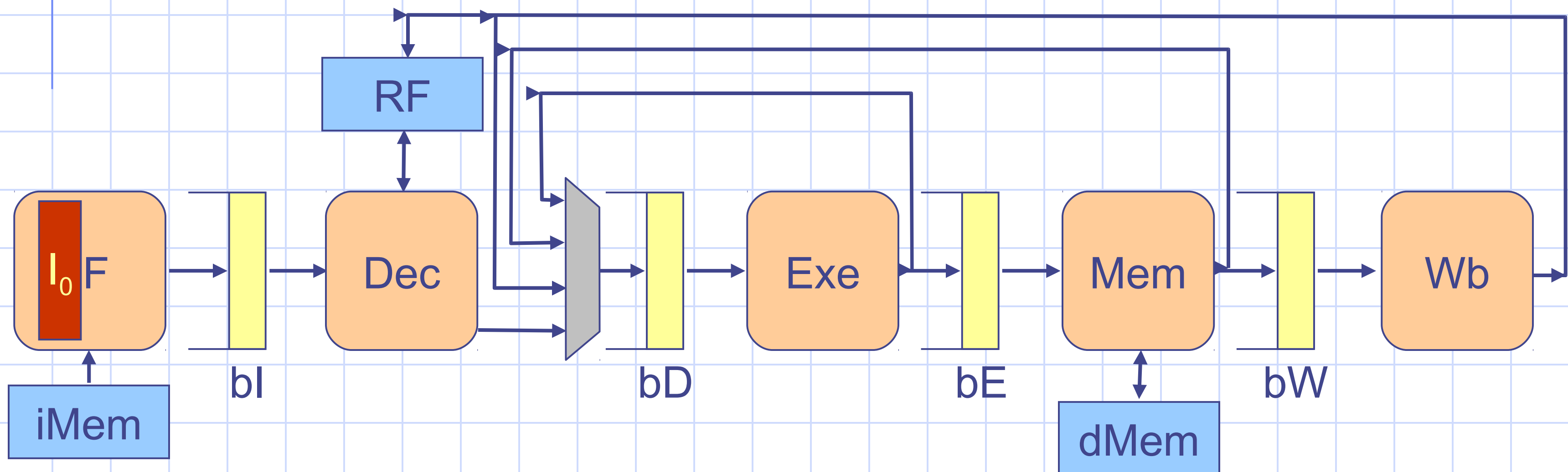


Scheduling Flexibility

- What if flip the order?

IF < Dec < Exe < Mem < Wb

An in-order processor

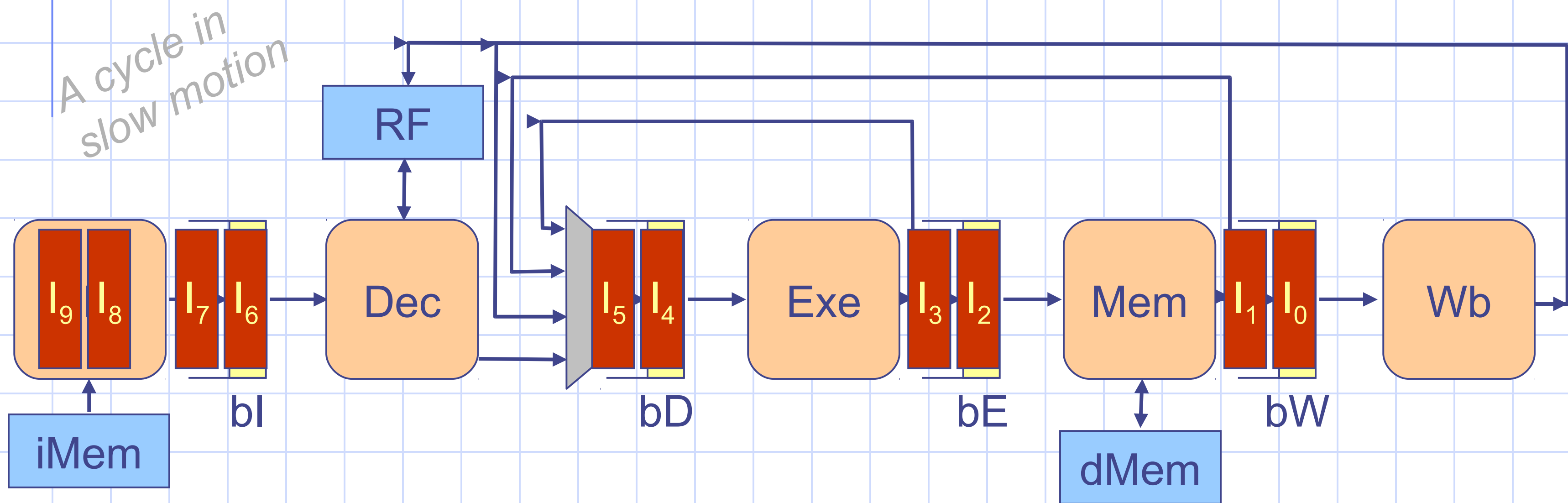


Scheduling Flexibility

- What happens if the user specifies:

Wb < Wb < Mem < Mem < Exe < Exe < Dec < Dec < IF < IF

No change in rules *a superscalar processor!*



Executing 2 instructions per cycle requires more resources but is functionally equivalent to the original design

Checking Completeness

- Given our constraints this should hold
- forall s. isSpec(s) => run([R12],s) = run([r1,r2],s)
- Forall r in {r3,rN}. forall s. isSpec(s) => run(r,s) = run(r',s)

Checking Soundness

- This takes a bit more work as:
  - We don’t really know what traces to compare against. R1,r3?
    - Can hazard some guesses (permutations? Elisions?)
  - Some implementation traces do not end in a state the spec can reach:
    - Extend the sequence and try again

Real Question: How much work is this?

- What do we have?
  - BSV Parser (from BSV-SW Compiler)
  - SMT solver w/ focus on bitvectors
- First step – verify scheduling properties
  - BSV ATS -> Lambda Calculus -> SMT
  - 2 weeks of time
- Okay. Maybe we this won't be so bad

So what exactly does it mean to show things are correct?

Bluespec Specification

- Bluespec designs are closer to specifications
  - Schedule makes it an implementation
  - Guaranteed safe
- Spec and Implementation in the same language
- Designers mostly do spec. refinement

What sort of questions can we ask of our solver?

- Convert rule R into π_R and δ_R
- Use this to ask questions about rule traces:
  - [A, B] = [B,A]
  - forall s. $\pi_A(s)$ & $\pi_B(\delta_A(s))$ =>

    $\pi_B(s)$ & $\pi_A(\delta_B(s))$ &

    $\delta_A(\delta_B(s)) = \delta_B(\delta_A(s))$

Bluespec - Origins

- Started from work modeling Cache coherence engines and processors in a Term Rewriting System (TRS) for verification [Stoy, Shen, Arvind]
- Precise enough to compile into hardware
  - TRAC compiler [Hoe]
  - Bluespec Compiler [Augustsson]

How do we get designers to formally verify?

- Reason in the design language
  - Inputs and Results have to be natural
- Low burden
  - Cannot ask for complex properties
  - Simple predicates / statements
- Fast feedback
  - Useful in testing

Correctness depends on the context

- We've broken the atomicity invariant
  - i and count are no longer in sync

```
rule safeRead(i==100 &&
              tempQ.first);
  if (p(count))
     $display(i, count);
```

Okay

```
rule unsafeRead(True);
  if (p(count))
        $display(i, count);
```

Not Okay

Can we verify such changes are safe?

Example: modifying memory

```
Mem      mem          <- mkMemory;
Reg#(int) i           <- mkReg(0);
Reg#(int) count       <- mkReg(0);


rule map(i<100);
    i <= i + 1;
    count <= count + f(mem.read(i));
```

What does it mean for two modules to be equivalent?

- Bisimularity:
  - Every rule trace in A has a corresponding rule trace in B which has the same “observable” effects and vice versa
- Observations – Method calls
  - Existence + output value

Split lookup and modify

```
Mem           mem   <- mkMem;
Reg#(int)     i     <- mkReg(0);
Reg#(int)     count <- mkReg(0);
FIFO#(int)    tempQ <- mkFIFO;

Rule mapReq(i < 100);
   i <= i + 1;
   tempQ.enq(mem.read(i));

rule mapResp(True);
   count <= count + f(tempQ.first());
   tempQ.deq();
```

But!

Now possible to see count and i out-of-sync

We also have the following rule in the system:

```
rule checkRunningTotal(True);
  if (p(count))
       $display(i, count);
```

What refinement do we want to see?

- Pic: One rule cloud to two then three
  - Splitting is key.
  - Merging also.
  - Microsteps.

Asking Questions of BSV

- Grab compiler dump after static evaluation
  - TRS of bitvectors and Actions
- Convert rules into functions:
  - $\pi(s)$ :: State -> Bool
  - $\delta(s)$ :: State -> State
- Use this to form SAT queries about rule execution traces

i.e. Does A followed by B behave like B followed by A?

Example:

```
Reg#(int) x <- mkReg(0);
Reg#(int) y <- mkReg(0);

rule swap(x!=0 && x < y);
  x <= y-x; y <= x;
endrule
method req(nx,ny)
         when (x==0);
  x <= nx; y <= ny;
endmethod
method result
         when(x==0);
   return y;
endmethod
```

```
R1_swap_guard(s0) =
 reg$Rd(getX(s0))!=0 &&
 reg$Rd(getX(s0))<reg$Rd(getY(s0))
R1_swap_body(s0) = let
 xv=reg$Rd(getX(s))
 yv=reg$Rd(getY(s))
 s1=updX(s0,reg$Wr(yv-xv,getX(s0))
 s2=updY(s1,reg$Wr(xv,getY(s1)))
 in s2
meth_req_guard(s0) =
 reg$Rd(getX(s0)) == 0
meth_req_body(nx,ny,s0) = let
 s1=updX(s0,reg$Wr(yv-xv,getX(s0))
 s2=updY(s1,reg$Wr(xv,getY(s1)))
 in s2
```

Designing Bluespec

- Language aimed at rapid design
  - Emphasis on refinement
  - A lot of work
- Large designs:
  - H.264
  - AirBlue – WiFi baseband