Lecture 3

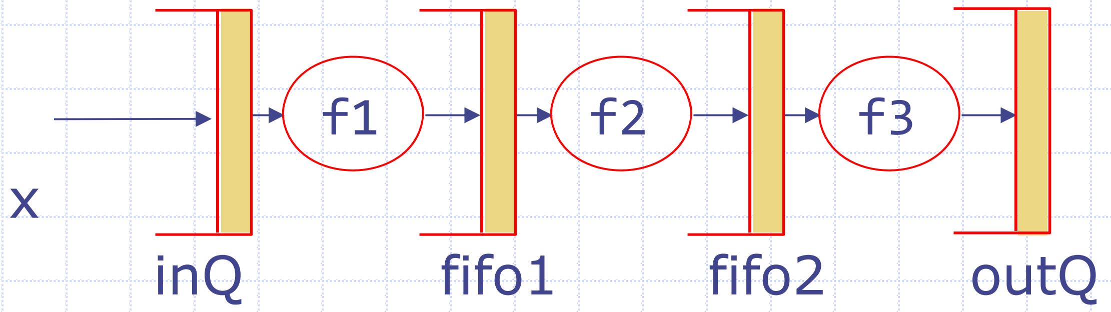
Bluespec System Verilog (BSV):
Concurrency and Semantics

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Oregon Programming Language Summer School (OPLSS)
Eugene, OR
July 18, 2018

Elastic pipeline

Use FIFOs instead of pipeline registers



Without concurrent execution it is hardly a pipelined system

```
rule stage1;
  fifo1.enq(f1(inQ.first));
  inQ.deq;    endrule
rule stage2;
  fifo2.enq(f2(fifo1.first));
  fifo1.deq; endrule
rule stage3;
  outQ.enq(f3(fifo2.first));
  fifo2.deq; endrule
```

- When can stage1 rule fire?
  - inQ has an element
  - fifo1 has space
- Can these 3 rules execute concurrently?

  Yes, but it must be possible to do enq and deq in a fifo simultaneously

Multirule Systems

- Most systems we have seen so far had multiple rules but only one rule was ready to execute at any given time (pair-wise mutually exclusive rules)
- Consider a system where multiple rules can be ready to execute at a given time
  - When can two such rules be executed together?
  - What does the synthesized hardware look like for concurrent execution of rules?

Meaning of Multi-rule Systems

Repeatedly:

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

Non-deterministic choice; User annotations can be used in rule selection

One-rule-at-a-time-semantics: Any legal behavior of a Bluespec program can be explained by observing the state updates obtained by applying only one rule at a time

However, for performance we execute multiple rules concurrently whenever possible

Concurrent execution of rules

- Two rules can execute concurrently, if concurrent execution would not cause a double-write error, *and*
- The final state can be obtained by executing rules one-at-a-time in some sequential order

Double-Write Error

```
rule one;
  y <= 3; x <= 5; x <= 7; endrule
```

Double write

```
rule two;
  y <= 3; if (b) x <= 7; else x <= 5; endrule
```

No double write

```
rule three;
  y <= 3; x <= 5; if (b) x <= 7; endrule
```

Possibility of a double write

- Parallel composition of actions, and consequently a rule containing it, is illegal if a double-write possibility exists
- The BSV compiler rejects a program if it there is any possibility of a double write in a rule or method

Can these rules execute concurrently?

(without violating the one-rule-at-a-time-semantics)

Example 1

```
rule ra;
   x <= x+1;
endrule
rule rb;
   y <= y+2;
endrule
```

Example 2

```
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= x+2;
endrule
```

Example 3

```
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= y+2;
endrule
```

Final value of (x,y) (initial values (0,0))

	Ex 1	Ex 2	Ex3
Concurrent Execution	(1,2)	(1,2)	(1,2)
		≠ ≠	≠
ra<rb	(1,2)	(1,3)	(1,2)
rb<ra	(1,2)	(3,2)	(3,2)
	No Conflict	Conflict	ra<rb

Conflict Matrix (CM)

BSV compiler generates the pairwise conflict information

Example 1

```
rule ra;
  x <= x+1;
endrule
rule rb;
  y <= y+2;
endrule
```

	ra	rb
ra	C	CF
rb	CF	C

Example 2

```
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
```

	ra	rb
ra	C	C
rb	C	C

Example 3

```
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= y+2;
endrule
```

	ra	rb
ra	C	<
rb	>	C

ra C rb : rules can't be executed concurrently
ra < rb : rules can be executed concurrently; the net effect is as if ra executed before rb
CF: rules can be performed concurrently; the net effect is the same with both rule orders

Conflict Matrix for an Interface

- Conflict Matrix (CM) defines which methods of a module can be called concurrently
- CM for a register:

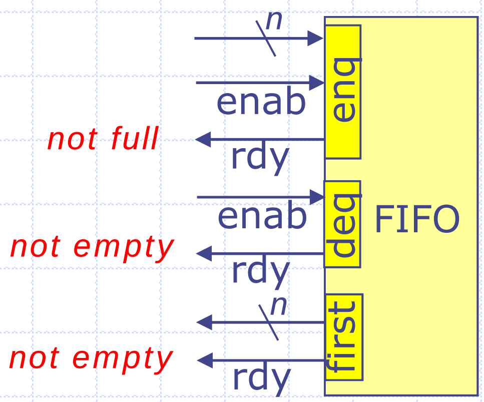
	reg.r	reg.w
reg.r	CF	<
reg.w	>	C

  - Two reads can be performed concurrently
  - Two concurrent writes conflict and are not permitted
  - A read and a write can be performed concurrently and it behaves as if the read happened before the write
- CM of a register is used systematically to derive the CM for the interface of a module and the CM for rules

A few examples...

One-Element FIFO

```
module mkFifo (Fifo#(1, t));
  Reg#(t)    d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(t x) if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method t first if (v);
    return d;
  endmethod
endmodule
```

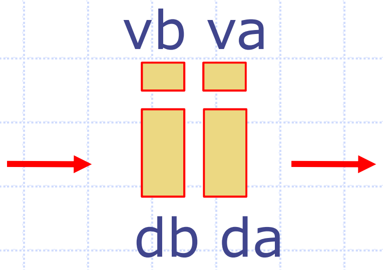


	enq	deq	first
enq	C	C	>
deq	C	C	>
first	<	<	CF

enq and deq are mutually exclusive and therefore can never execute concurrently

This FIFO is not useful for implementing pipelined system

How about a Two-Element FIFO?

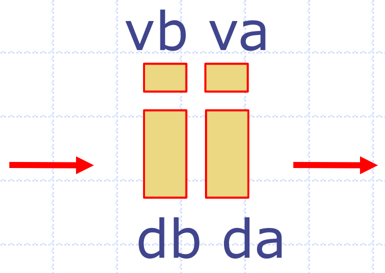


Assume, if there is only one element in the FIFO, it resides in da

- Initially, both va and vb are false
- First enq will store the data in da and mark va true
- An enq can be done as long as vb is false; a deq can be done as long as va is true

Two-Element FIFO

```
module mkCFFifo (Fifo#(2, t));
 //instantiate da, va, db, vb
  rule canonicalize if (vb && !va);
    da <= db;
    va <= True;
    vb <= False;
  endrule
  method Action enq(t x) if (!vb);
    begin db <= x; vb <= True; end
  endmethod
  method Action deq if (va);
    va <= False;
  endmethod
  method t first if (va);
    return da;
  endmethod
endmodule
```



Both enq and deq can execute concurrently

But neither enq or deq can execute again until the canonicalize rule fires!

	enq	deq	first
enq	C	CF	CF
deq	CF	C	>
first	CF	<	CF

Many other FIFO designs are possible

	enq	deq	first
enq	C	>	>
deq	<	C	>
first	<	<	CF

Pipelined FIFO

one can enq into a full FIFO if a deq is done simultaneously

	enq	deq	first
enq	C	<	<
deq	>	C	>
first	>	<	CF

Bypass FIFO

one can deq from an empty FIFO if a enq is done simultaneously

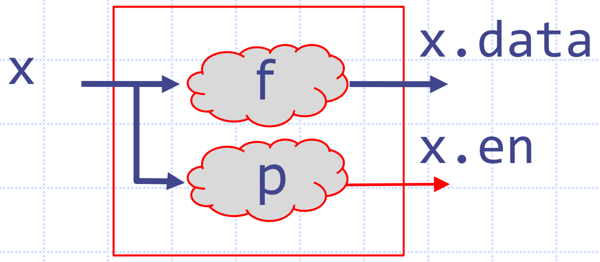
Design of such FIFOs requires the use of EHRs, registers with bypasses. Unfortunately, we don't have time to discuss them here

Hardware generation using *conflict* (CM) information

Preliminaries

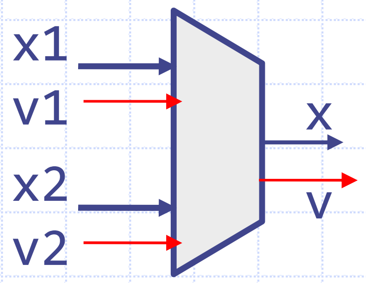
- Recall, BSV compiler generates a combinational circuit for each rule and method
- If rule or method sets a register x then it must generate both the data and the enable signal for the register, e.g.

```
rule foo(p(x)); x <= f(x); endrule
```



- similarly for each action method and actionValue method

Preliminaries – need for muxes



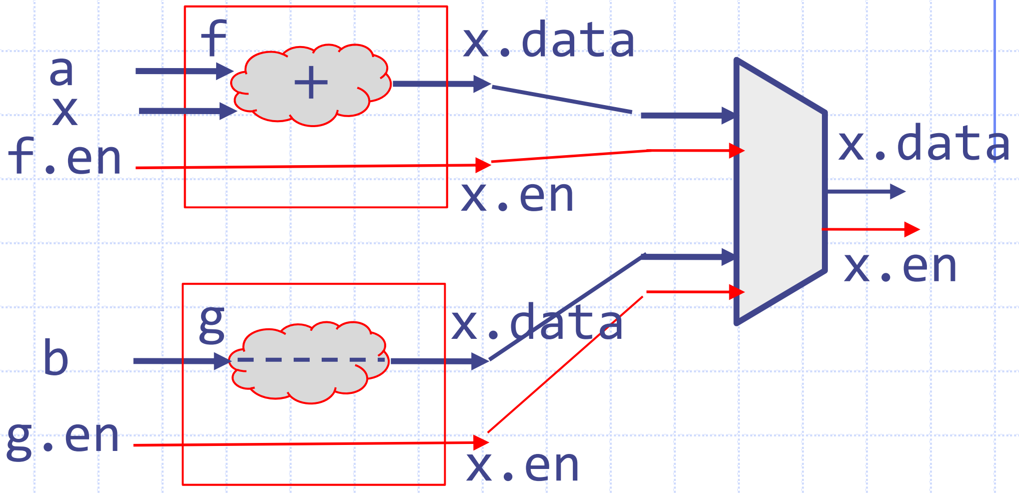
x = (v1 & x1) | (v2 & x2)
v = v1 | v2

- We associate a control wire `vi` with each value `xi`; `xi` has a meaningful value only if its corresponding `vi` is true
- When we merge two or more values, at most one v*i* should be true at any given time (*one-hot-encoding*), i.e., `vi`'s must be pairwise mutually exclusive
- x, x1, and x2 are bit vectors and must have the same size

BSV compiler ensures this

Need for conflict information

```
module mkEx (...);
   Reg#(t) x <- mkReg(0);
   method Action f(t a);
      x <= x+a;
   endmethod
   method Action g(t b);
      x <= b;
   endmethod
endmodule
```



- Note at most one of f.en and g.en should be true; otherwise this circuit is not meaningful
- *How does the compiler ensure that?*
- CM to rescue: CM for `mkEx` will show that methods f and g conflict and should never be called at the same time

Using CM

```
module mkEx (...);
  Reg#(t) x <- mkReg(0);
  method Action f(t a);
     x <= x+a;
  endmethod
  method Action g(t b);
     x <= b;
  endmethod
endmodule
```

The CM for `mkEx` will show that methods f and g conflict

Suppose `m <- mkEx();`

```
rule ra;... m.f(1); m.g(2);... endrule
```

ra is an illegal rule

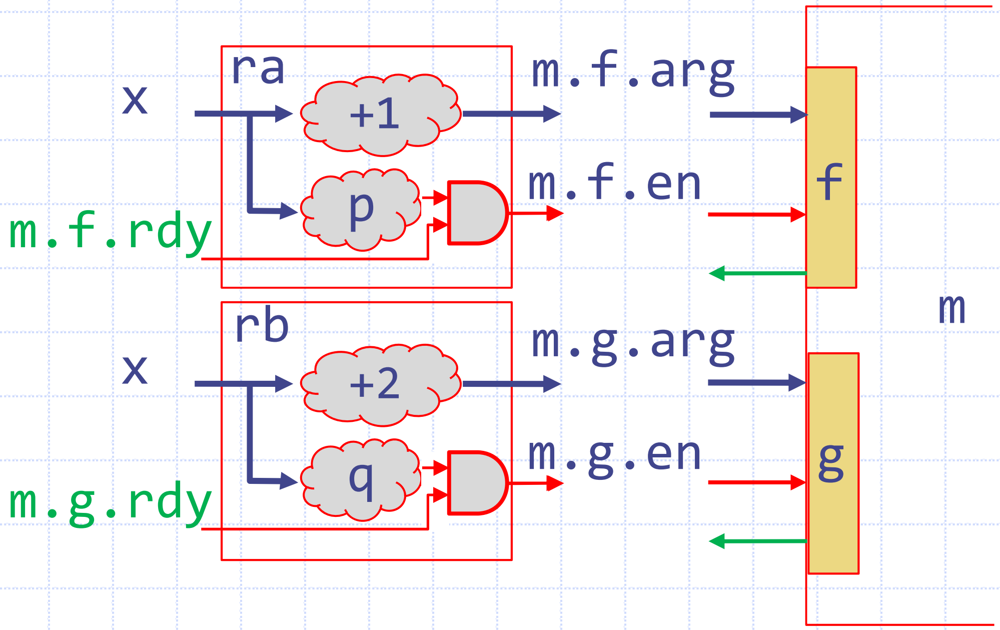
```
rule rb;... m.f(1); ... endrule
rule rc;... m.g(2); ... endrule
```

rb and rc should not be scheduled concurrently, and executed one by one

how?

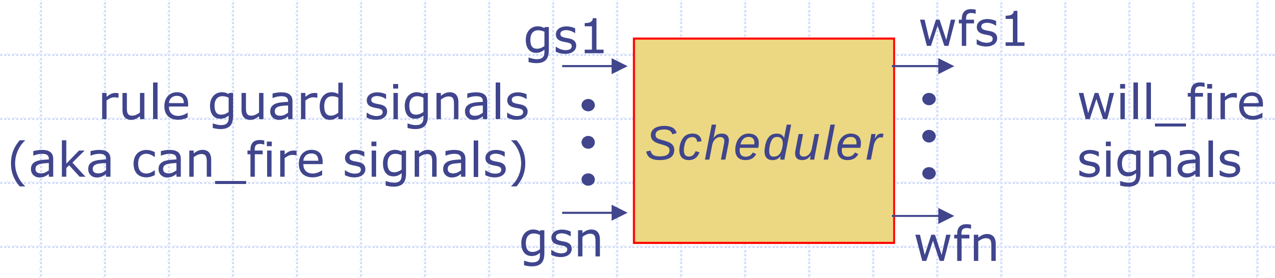
Concurrent rule execution

```
rule ra(p(x));
   m.f(x+1);
endrule
rule rb(q(x));
   m.g(x+2)
endrule
```



- This circuit will execute rules ra and rb concurrently
- This circuit is correct only if rules ra and rb do not conflict ($\Rightarrow$ methods f and g of m do not conflict)
- Suppose rules ra and rb do conflict!

Need for a rule scheduler

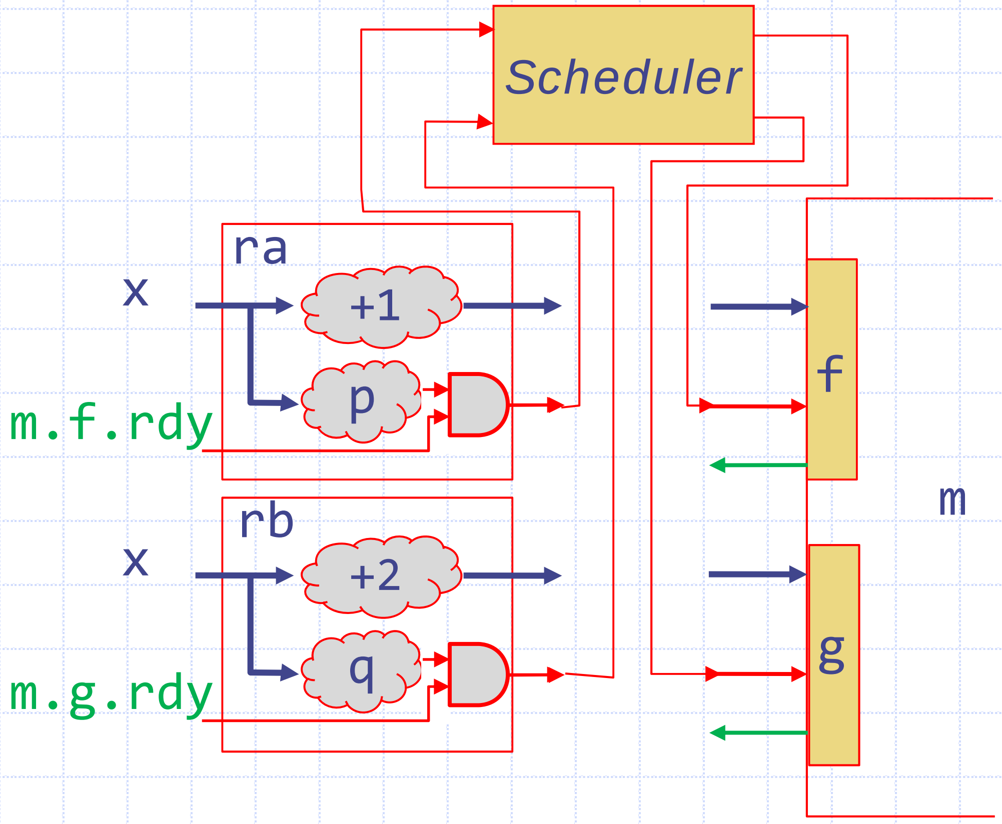


- Guards (gs1 … gsn) of many rules may be true simultaneously, and some of them may conflict
- BSV compiler constructs a combinational scheduler circuit with the following property:

> for all *i* and *j*, if wfs*i* and wfs*j* are true then the corresponding gs*i* and gsj must be true and rules *i* and *j* must not conflict with each other

Circuit with a scheduler

```
rule ra (p(x));
   m.f(x+1);
endrule
rule rb (q(x));
   m.g(x+2)
endrule
```

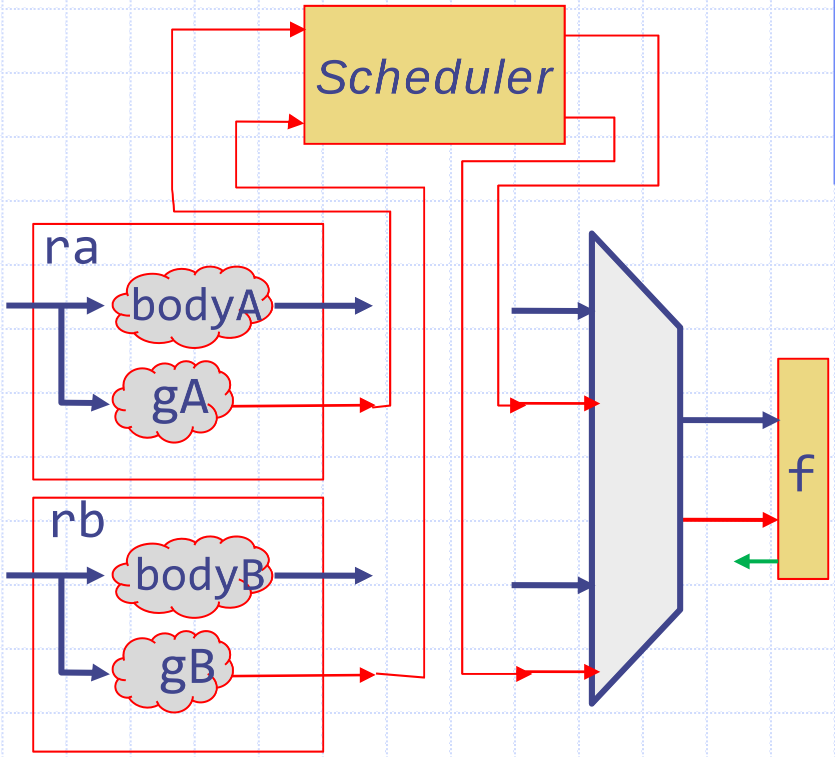


- The scheduler is generated based on the CM of ra and rb, which in turn depends upon the CM of m
- Generally a scheduler has small number of gates
- A correct but low performance scheduler may schedule only one rule at a time

A more complete picture

need for muxes

- Multiple rules may invoke the same method, so we need to put a mux in front of the interface
- Again, if the scheduler is implemented correctly, it is guaranteed that only one of the inputs to the mux will be true (one-hot encoding)



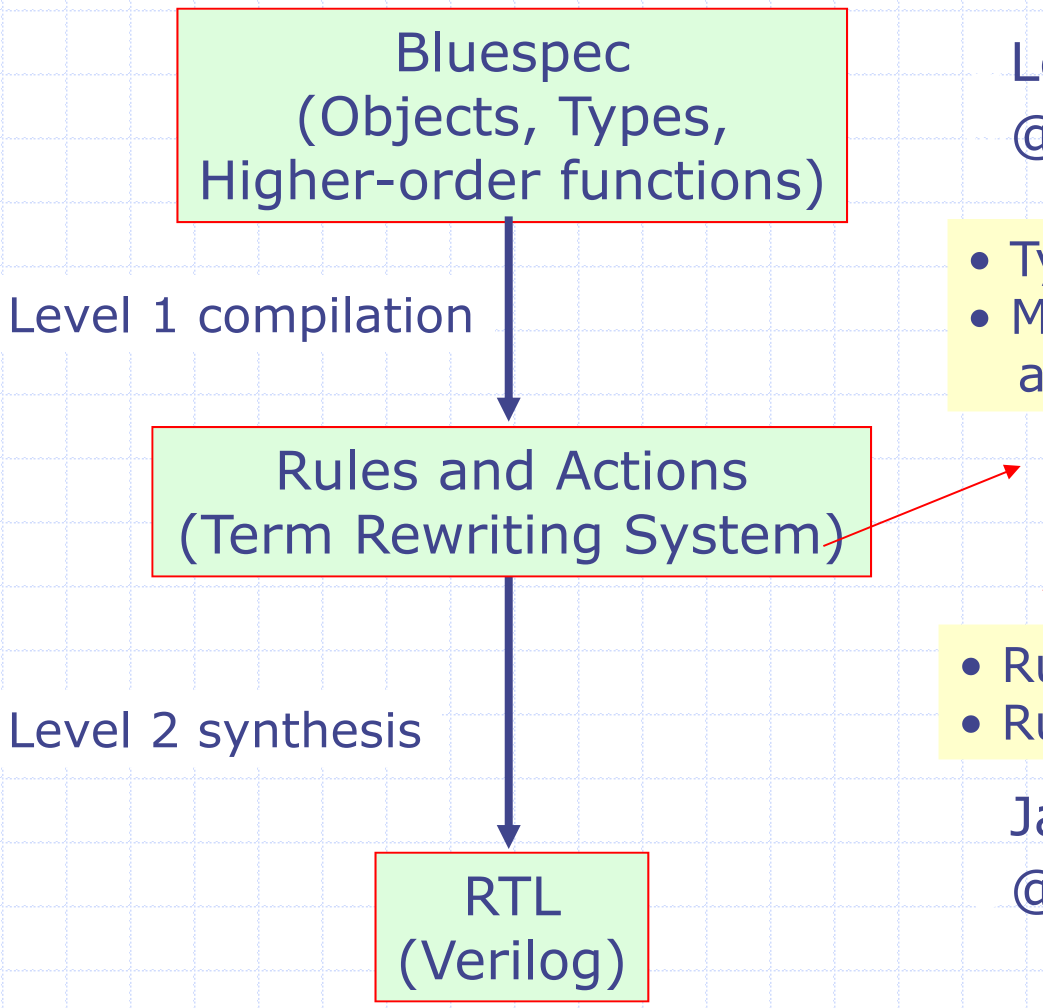
Takeaway

- One-rule-at-a-time semantics are very important to understand what behaviors a system can show
- Efficient hardware for multi-rule system requires that many rules execute in parallel without violating the one-rule-at-time semantics
- BSV compiler builds a scheduler circuit to execute as many rules as possible concurrently
- For high-performance designs we have to worry about the CM characteristics of our modules

Bluespec Semantics

Behaviors that can be generated by executing rules one at a time

Bluespec: Two-Level Compilation



Lennart Augustsson
@Sandburst 2000-2002

- Type checking
- Massive partial evaluation and static elaboration

Now we call this Guarded Atomic Actions

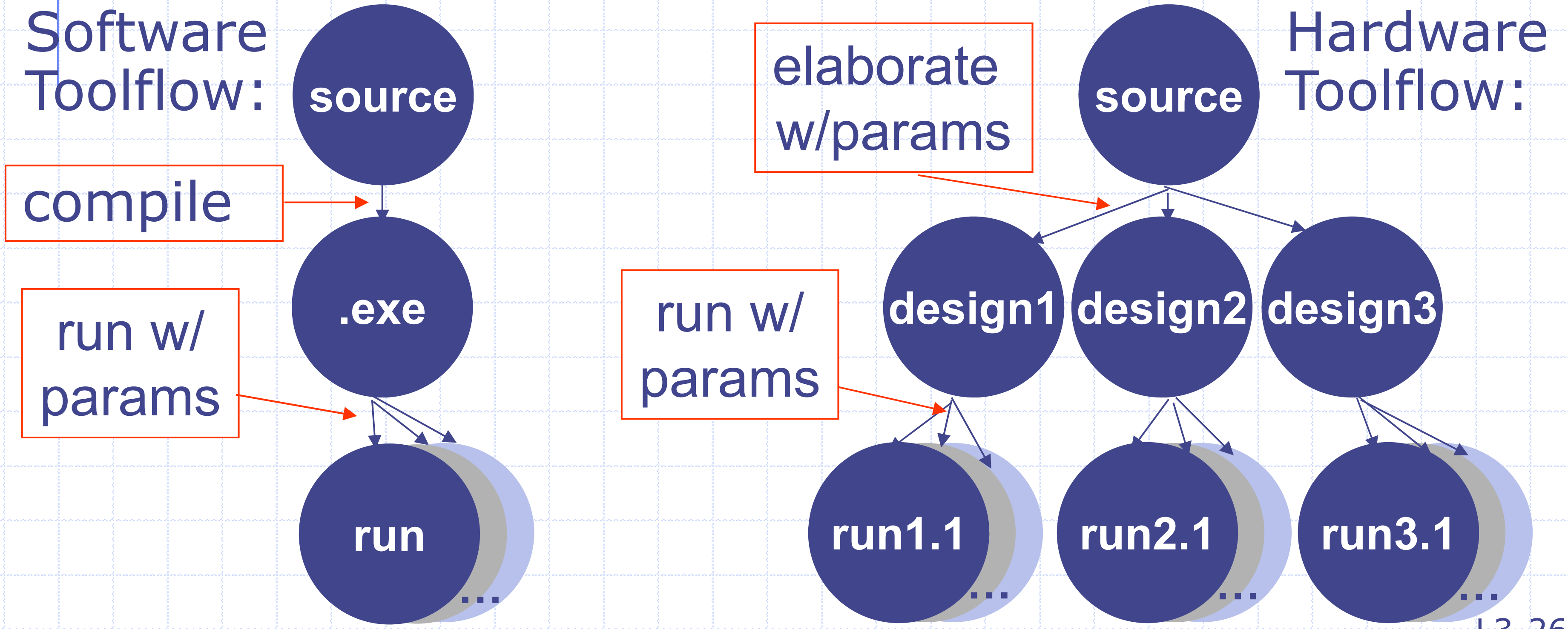
- Rule conflict analysis
- Rule scheduling

James Hoe & Arvind
@MIT 1997-2000

Static Elaboration

At compile time

- Inline function calls and unroll loops
- Instantiate modules with specific parameters
- Resolve polymorphism/overloading, perform most data structure operations



GAA Execution model

Repeatedly:

- Select a rule to execute ← Non-deterministic choice
- Compute the state updates
- Make the state updates

BSV Kernel syntax (monadic style)

No recursion: methods of only other modules can be called from a module

Expression

e ::= c | x | op(e)

Action

Syntax	
a ::= let x = r in a	← Register read
| r := e ; a	← Register assignment
| let x = f (e) in a	← Method call
| let x = e in a	← Let binding
| if e then a ; a	← Conditional action
| assert e ; a	← Guarded action
| return e	← Needed to extract the result of an action

Module

m ::= ⟨⟨r , c⟩*, ⟨s, a⟩*, ⟨f , λx. a⟩*⟩ | m + m

Registers with initial values

Rules

Methods

Non-modular operational semantics

Arvind, Nirav Dave, Michael Pellauer

2007

Semantics of executing an action

$$\boxed{O \vdash a \Rightarrow (U,v)}$$

- O is the set of values of *all the registers in all the modules* before action *a* executes
- U is the set of register updates implied by the execution of *a* (initially U is empty)
- *v* is the value returned as a consequence of executing *a*

Action Semantics

reg-read
$$\frac{O \vdash [O(r)/x]a \Rightarrow (U,v)}{O \vdash (\text{Let } x = r;\ a) \Rightarrow (U,v)}$$

reg-update
$$\frac{[[e]] = v_r \qquad O \vdash a \Rightarrow (U,v)}{O \vdash (r := e;\ a) \Rightarrow (U \oplus \{(r, v_r)\}, v)}$$

let-action
$$\frac{[[e]] = v_x \qquad O \vdash [v_x/x]a \Rightarrow (U,v)}{O \vdash (x = e;\ a) \Rightarrow (U, v)}$$

method call
$$\frac{[[e]] = v_y \qquad f = \lambda y.b \quad O \vdash [v_y/y]b \Rightarrow (U_f,v_x) \quad O \vdash [v_x /x]a \Rightarrow (U,v)}{O \vdash (x = f(e);\ a) \Rightarrow (U_f \oplus U, v)}$$

$\oplus$ represents a disjoint union;
Otherwise it is a double-write error

Action Semantics *continued*

If-True
$$\frac{[[e]] = \text{True} \quad O \vdash a_T \Rightarrow (U_T, -) \quad O \vdash a \Rightarrow (U,v)}{O \vdash (\text{if } e \text{ then } a_T, a) \Rightarrow (U_T \oplus U, v)}$$

If-False
$$\frac{[[e]] = \text{False} \quad O \vdash a \Rightarrow (U,v)}{O \vdash (\text{if } e \text{ then } a_T ,a) \Rightarrow (U,v)}$$

assert
$$\frac{[[e]] = \text{True} \quad O \vdash a \Rightarrow (U,v)}{O \vdash (\text{assert } e;\ a) \Rightarrow (U,v)}$$

The system will get stuck if the assertion fails

return
$$\frac{[[e]] = v}{O \vdash (\text{return } e) \Rightarrow (\{\},v)}$$

State transition

rule $$\frac{\langle s, a\rangle \in \text{rulesOf}(m) \qquad O \vdash a \Rightarrow (U,-)}{O \vdash (\text{rule } s) \rightarrow O[U]}$$

where O[U] is the set of register values O updated by U

Behavior: sequence of state changes

$$\frac{\langle s,a\rangle \in \text{rulesOf}(m) \qquad O_n \vdash (\text{rule } s) \rightarrow O_{n+1}}{\langle O_0, \ldots, O_n\rangle, m \vdash \langle O_0, \ldots, O_n, O_{n+1}\rangle}$$

where O_0 is the initial register values

[[m]], the *meaning of a module*, is the set of all behaviors, given the initial register values

Module Refinement

$m_1 \leq m_2$ (m_1 refines m_2) if $[[m_1]] \subseteq [[m_2]]$

- One may want to observe state changes only in a subset of registers for refinement purposes
- If two sequences contain the same final state given the same initial state, we treat them as congruent or equivalent
- A system is *deterministic* if all its behaviors for a given input are congruent

Modular semantics

Murali Vijayaraghavan,
Adam Chlipala
2016

Modular semantics

- The operational semantics we have given so are non modular because the method call rule looks inside the module of the called method

method call

$$\frac{[[e]] = v_y \quad f = \lambda y.b \quad O \vdash [v_y/y]b \Rightarrow (U_f, v_x) \qquad O \vdash [v_x/x]a \Rightarrow (U, v)}{O \vdash (x = f(e);\ a) \Rightarrow (U_f \oplus U,\ v)}$$

- For modular semantics we need to *assume* the result returned by the called method and record it in a label
- Later we reconcile the labels when two modules communicate

Modular semantics: Action

reg-read

$$\frac{O, m \vdash [O(r)/x]a \Rightarrow (U,v), l}{O, m \vdash (\text{Let } x = r;\ a) \Rightarrow (U,v), l}$$

reg-update

$$\frac{[[e]] = v_r \qquad O, m \vdash a \Rightarrow (U,v), l}{O, m \vdash (r := e;\ a) \Rightarrow (U \oplus \{(r, v_r)\}, v), l}$$

Let-action

$$\frac{[[e]] = v_x \qquad O, m \vdash [v_x/x]a \Rightarrow (U,v), l}{O, m \vdash (x = e;\ a) \Rightarrow (U, v), l}$$

method call

$$\frac{[[e]] = v_y \qquad O, m \vdash [v_x /x]a \Rightarrow (U,v), l}{O, m \vdash (x = f(e);\ a) \Rightarrow (U, v), \{<f,v_y,v_x>\} \oplus l}$$

f must not be a method of module m

v_x is a free variable to represent the value returned by a method call

$\oplus$ represents a disjoint union of labels; Otherwise it is a double-method call error

Modular Semantics: actions

continued

If-True
$$\frac{[[e]] = \text{True} \quad O, m \vdash a_T \Rightarrow (U_T, -), l_T \quad O, m \vdash a \Rightarrow (U,v), l}{O, m \vdash (\text{if } e \text{ then } a_T, a) \Rightarrow (U_T \oplus U, v), l_T \oplus l}$$

If-False
$$\frac{[[e]] = \text{False} \quad O, m \vdash a \Rightarrow (U,v), l}{O, m \vdash (\text{if } e \text{ then } a_T, a) \Rightarrow (U,v), l}$$

assert
$$\frac{[[e]] = \text{True} \quad O, m \vdash a \Rightarrow (U,v), l}{O, m \vdash (\text{assert } e;\ a) \Rightarrow (U,v), l}$$

return
$$\frac{[[e]] = v}{O, m \vdash (\text{return } e) \Rightarrow (\{\},v), \{\}}$$

empty label

Special Initial label: •

Rule

$$\frac{\langle s, a\rangle \in \text{rulesOf}(m) \qquad O, m \vdash a \Rightarrow (U,-), l}{O, m \vdash (\text{rule } s) \Rightarrow U, \ \bullet \oplus l}$$

There can be only one • in l, thus $l_1 \cup l_2$ is defined only if the following holds:

1. if $\bullet \in l_1$ then $\bullet \notin l_2$
2. if $\bullet \in l_2$ then $\bullet \notin l_1$

Incoming method calls

A rule in a module can call several methods of another module concurrently; thus, we need to give semantics for concurrent method calls of a module

empty-method

$$\frac{}{O, m \vdash (\text{empty-method}) \Rightarrow \{\}, \{\}}$$

method calls

$$\frac{\langle f, \lambda x.a\rangle \in \text{methodsOf}(m) \qquad f \text{ in not in call set } x \qquad O, m \vdash (x) \Rightarrow U_1, l_1 \qquad O, m \vdash [v_x/x]a \Rightarrow (U_2, v), l_2}{O, m \vdash (x \cup f(v_x)) \Rightarrow U_1 \oplus U_2,\quad l_1 \oplus \{\langle \underline{f}, v_x, v\rangle\} \oplus l_2}$$

Notice the underline

Discharging a method call

discharge

$$\frac{O_1 \vdash m_1 \Rightarrow U_1,\ l_1 \qquad O_2 \vdash m_2 \Rightarrow U_2,\ l_2 \qquad \text{compatible}(l_1,l_2)}{\langle O_1, O_2\rangle \vdash m_1+m_2 \Rightarrow \langle U_1,U_2\rangle,\ (l_1 \oplus l_2)/(m_1,m_2)}$$

where

compatible(l1,l2) means

1. if $\langle f,x,y\rangle \in l_1$ and $f \in$ methodsOf(m_2) then $\langle \underline{f},x,y\rangle \in l_2$
2. if $\langle f,x,y\rangle \in l_2$ and $f \in$ methodsOf(m_1) then $\langle \underline{f},x,y\rangle \in l_1$

$(l_1 \oplus l_2)/(m_1,m_2)$ means $(l_1 \oplus l_2)$ where all the matching labels from m_1 and m_2 have been deleted

State transition

rule-state-transition

$$\frac{\langle O_1,\ldots,O_n\rangle \vdash m_1+\ldots+m_n \Rightarrow \langle U_1,\ldots,U_n\rangle, \bullet}{\langle O_1,\ldots,O_n\rangle \vdash m_1+\ldots+m_n \rightarrow \langle O_1[U_1], \ldots, O_n[U_n]\rangle}$$

Behavior: sequence of state changes

$$\frac{\langle s,a\rangle \in \text{rulesOf}(m)b \quad OV_k \vdash (\text{rule } s) \rightarrow OV_{k+1}}{\langle OV_0, \ldots , OV_k\rangle, m \vdash \langle OV_0, \ldots , OV_k, OV_{k+1}\rangle}$$

where $OV = \langle O_1,\ldots,O_n\rangle$ is the vector of the register values in all the modules and OV_0 is the vector of initial values

Labelled transitions

- [[m]], the *meaning of a module,* is the set of labels a module can produce by applying its rules given the initial register values (closure of $\Rightarrow$)

$$\boxed{m_1 \leq m_2 \;\; (m_1 \text{ refines } m_2) \text{ if } [[m_1]] \subseteq [[m_2]]}$$

- Modular refinement theorem

$$\boxed{\text{if } A' \leq A \; (A' \text{ refines } A) \text{ then } (A'+B) \leq (A+B)}$$

we don't have to look inside B to refine A!

Syntactic merger: $+_s$

Let $m_1 = \langle\langle r1 , c1\rangle^*, \langle s1, a1\rangle^*, \langle f1 , \lambda x.\ a\rangle^*\rangle$

$m_2 = \langle\langle r2 , c2\rangle^*, \langle s2, a2\rangle^*, \langle f2 , \lambda x.\ a\rangle^*\rangle$

where the identifiers in the two modules are pairwise disjoint, then

$m_1 +_s m_2$ produces a new module m by merging modules m_1 and m_2 such that

1. inline methods of m2 called in m1 and then delete those method definitions from m2
2. inline methods of m1 called in m2 and then delete those method definitions from m1
3. Methods of m are the union of methods remaining in m1 and m2

Theorem: $[[m_1 +_s m_2]] = [[m_1 + m_2]]$

Summary

- BSV is being used by us and many other companies to design extremely sophisticated hardware
- The is no discernable impact on the quality of hardware being produced
- Adam Chlipala and collaborators have built Kami, a system for writing mechanically checked proofs for BSV programs
- We are teaching our introductory logic design and computer architecture class using BSV

It is the nature of a man as he grows older, ..., to protest against change, particularly change for the better.

Travels with Charlie
John Steinbeck