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

- inQ has an element
- fifo1 has space

Without concurrent execution it is hardly a pipelined system

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

```
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
```
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

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.

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

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.

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

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

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

Double write
No double write
Possibility of a double write
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

\[ \text{rule } ra; \]
\[ x \leq x+1; \]
\[ \text{endrule} \]
\[ \text{rule } rb; \]
\[ y \leq y+2; \]
\[ \text{endrule} \]

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

Ex 1: (1,2)  Ex 2: ≠ ≠  Ex 3: (1,2)

Concurrent Execution

\(ra < rb\)

No Conflict

\(rb < ra\)

\(ra < rb\)

Example 2

\[ \text{rule } ra; \]
\[ x \leq y+1; \]
\[ \text{endrule} \]
\[ \text{rule } rb; \]
\[ y \leq x+2; \]
\[ \text{endrule} \]

Example 3

\[ \text{rule } ra; \]
\[ x \leq y+1; \]
\[ \text{endrule} \]
\[ \text{rule } rb; \]
\[ y \leq y+2; \]
\[ \text{endrule} \]
**Conflict Matrix (CM)**

BSV compiler generates the pairwise conflict information

Example 1

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

Example 2

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

Example 3

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

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>CF</td>
</tr>
<tr>
<td>rb</td>
<td>CF</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>rb</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>&lt;</td>
</tr>
<tr>
<td>rb</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th></th>
<th>reg.r</th>
<th>reg.w</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg.r</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>reg.w</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

- 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 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?

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

Assume, if there is only one element in the FIFO, it resides in da
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;
    endmethod
    method Action deq if (va);
        va <= False;
    endmethod
    method t first if (va);
        return da;
    endmethod
endmodule

Two-Element FIFO

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

<table>
<thead>
<tr>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>CF</td>
<td>CF</td>
</tr>
<tr>
<td>CF</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>CF</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>
Many other FIFO designs are possible

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

<table>
<thead>
<tr>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>

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
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.

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

Similarly for each action method and actionValue method.
Preliminaries – need for muxes

We associate a control wire \( v_i \) with each value \( x_i \); \( x_i \) has a meaningful value only if its corresponding \( v_i \) 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., \( v_i \)'s must be pairwise mutually exclusive.

\( x, x_1, \) and \( x_2 \) are bit vectors and must have the same size.

BSV compiler ensures this.

\[
x = (v_1 \& x_1) | (v_2 \& x_2)
\]
\[
v = v_1 \mid v_2
\]
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  
rule rb;... m.f(1); ... endrule
rule rc;... m.g(2); ... endrule
```

ra is an illegal rule.

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

how?
Concurrent rule execution

This circuit will execute rules ra and rb concurrently.
This circuit is correct only if rules ra and rb do not conflict (⇒ 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 $gs_j$ must be true and rules $i$ and $j$ must not conflict with each other.
Circuit with a scheduler

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

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

Bluespec (Objects, Types, Higher-order functions)

Level 1 compilation

Rules and Actions (Term Rewriting System)

Level 2 synthesis

RTL (Verilog)

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

Software Toolflow:

compile

run w/ params

Hardware Toolflow:

source

elaborate w/params

design1

design2

design3

run w/ params

run1.1

run2.1

run3.1
GAA Execution model

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

Non-deterministic choice
BSV Kernel syntax
(monadic style)

Expression
\[ e ::= c \mid x \mid \text{op}(e) \]

Action
\[ a ::= \text{let } x = r \text{ in } a \]
\[ \mid r ::= e ; a \]
\[ \mid \text{let } x = f(e) \text{ in } a \]
\[ \mid \text{let } x = e \text{ in } a \]
\[ \mid \text{if } e \text{ then } a ; a \]
\[ \mid \text{assert } e ; a \]
\[ \mid \text{return } e \]

Module
\[ m ::= \langle (r,c)^*, (s,a)^*, (f,\lambda x.a)^* \rangle \mid m + m \]

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

- Register read
- Register assignment
- Method call
- Let binding
- Conditional action
- Guarded action
- Needed to extract the result of an action

Registers with initial values
Rules
Methods
Non-modular operational semantics

Arvind, Nirav Dave, Michael Pellauer
2007
Semantics of executing an action

\[ O \vdash a \Rightarrow (U,v) \]

- \( O \) is the set of values of \textit{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**

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

**reg-update**

\[ [[e]] = v_r \quad O \vdash a \Rightarrow (U,v) \]
\[ O \vdash (r := e; \ a) \Rightarrow (U \oplus \{(r, v_r)\}, v) \]

**let-action**

\[ [[e]] = v_x \quad O \vdash [v_x/x]a \Rightarrow (U,v) \]
\[ O \vdash (x = e; \ a) \Rightarrow (U, v) \]

**method call**

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

\[ \oplus \text{ represents a disjoint union;} \]
\[ \text{Otherwise it is a double-write error} \]
Action Semantics \textit{continued}

\begin{align*}
\text{If-True} & \quad [[e]] = \text{True} \quad O \vdash a_T \Rightarrow (U_T, -) \quad O \vdash a \Rightarrow (U, v) \\
& \quad O \vdash (\text{if } e \text{ then } a_T, a) \Rightarrow (U_T \oplus U, v) \\
\text{If-False} & \quad [[e]] = \text{False} \quad O \vdash a \Rightarrow (U, v) \\
& \quad O \vdash (\text{if } e \text{ then } a_T, a) \Rightarrow (U, v) \\
\text{assert} & \quad [[e]] = \text{True} \quad O \vdash a \Rightarrow (U, v) \\
& \quad O \vdash (\text{assert } e; a) \Rightarrow (U, v) \\
\text{return} & \quad [[e]] = v \\
& \quad O \vdash (\text{return } e) \Rightarrow (\{\}, v)
\end{align*}

The system will get stuck if the assertion fails
State transition

rule \( <s, a> \in \text{rulesOf}(m) \) \implies O \models a \Rightarrow (U, -)

\( O \models (\text{rule } s) \rightarrow O[U] \)

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

Behavior: sequence of state changes

\( <s, a> \in \text{rulesOf}(m) \) \implies O_n \models (\text{rule } s) \rightarrow O_{n+1} \)

\( <O_0, ..., O_n>, m \models <O_0, ..., O_n, O_{n+1}> \)

where \( O_0 \) is the initial register values

\([[:m:]],[/]<m詹, the meaning of a module>, is the set of all behaviors, given the initial register values
Module Refinement

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

- 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 far are non-modular because the method call rule looks inside the module of the called method.

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**

\[ O, m \models [O(r)/x]a \Rightarrow (U,v), l \]

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

**reg-update**

\[ [[e]] = v_r \]

\[ O, m \models a \Rightarrow (U,v), l \]

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

**Let-action**

\[ [[e]] = v_x \]

\[ O, m \models [v_x/x]a \Rightarrow (U,v), l \]

\[ O, m \models (x = e; \ a) \Rightarrow (U, v), l \]

**method call**

\[ [[e]] = v_y \]

\[ O, m \models [v_x/x]a \Rightarrow (U,v), l \]

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

\textbf{f must not be a method of module m}

\[ \oplus \text{ represents a disjoint union of labels;} \]

\textbf{Otherwise it is a double-method call error}

\textbf{v}_x \textbf{ is a free variable to represent the value returned by a method call}
Modular Semantics: actions

**If-True**

\[ [[e]] = \text{True} \quad O, m \models a_T \Rightarrow (U_{T'}, -), l_T \]
\[
O, m \models a \Rightarrow (U, v), l
\]
\[ O, m \models (\text{if } e \text{ then } a_T, a) \Rightarrow (U_T \oplus U, v), l_T \oplus l \]

**If-False**

\[ [[e]] = \text{False} \quad O, m \models a \Rightarrow (U, v), l \]
\[ O, m \models (\text{if } e \text{ then } a_T, a) \Rightarrow (U, v), l \]

**assert**

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

**return**

\[ [[e]] = v \quad O, m \models (\text{return } e) \Rightarrow (\{\}, v), \{\} \]
There can be only one • in l, thus \( l_1 \cup l_2 \) is defined only if the following holds:

1. if • \( \in l_1 \) then • \( \notin l_2 \)
2. if • \( \in l_2 \) then • \( \notin l_1 \)
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

\[ O, m \vdash (\text{empty-method}) \Rightarrow \{\}, \{\} \]

method calls

\[ <f, \lambda x.a> \in \text{methodsOf}(m) \quad \text{f in not in call set } x \]

\[ O, m \vdash (x) \Rightarrow U_1, I_1 \quad O, m \vdash [v_x/x]a \Rightarrow (U_2,v), I_2 \]

\[ O, m \vdash (x \cup f(v_x)) \Rightarrow U_1 \oplus U_2, \quad I_1 \oplus \{<f, v_x,v>\} \oplus I_2 \]

Notice the underline
Discharging a method call

discharge

\[
\begin{align*}
O_1 \vdash m_1 \Rightarrow U_1, l_1 & \quad O_2 \vdash m_2 \Rightarrow U_2, l_2 & \quad \text{compatible}(l_1,l_2) \\
<O_1, O_2> \vdash m_1 + m_2 \Rightarrow <U_1, U_2>, & \quad (l_1 \oplus l_2)/(m_1,m_2)
\end{align*}
\]

where

\text{compatible}(l1,l2) means

1. if \(<f,x,y> \in l_1 \text{ and } f \in \text{methodsOf}(m_2)\) then \(<f,x,y> \in l_2\)
2. if \(<f,x,y> \in l_2 \text{ and } f \in \text{methodsOf}(m_1)\) then \(<f,x,y> \in l_1\)

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

\[
\begin{align*}
&\text{rule-state-transition} \\
&\langle O_1, \ldots, O_n \rangle \mid m_1 + \ldots + m_n \Rightarrow \langle U_1, \ldots, U_n \rangle, \bullet \\
&\langle O_1, \ldots, O_n \rangle \mid m_1 + \ldots + m_n \rightarrow \langle O_1[U_1], \ldots, O_n[U_n] \rangle
\end{align*}
\]

Behavior: sequence of state changes

\[
\langle s, a \rangle \in \text{rulesOf}(m) b \quad OV_k \mid (\text{rule } s) \rightarrow OV_{k+1}
\]

\[
\langle OV_0, \ldots, OV_k \rangle, m \mid \langle OV_0, \ldots, OV_k, OV_{k+1} \rangle
\]

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 $\Rightarrow$)

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

Modular refinement theorem

if $A' \leq A$ (A' refines A) then $(A'+B) \leq (A+B)$

we don’t have to look inside B to refine A!
Syntactic merger: $+_s$

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

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

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 $m_2$ called in $m_1$ and then delete those method definitions from $m_2$
2. inline methods of $m_1$ called in $m_2$ and then delete those method definitions from $m_1$
3. Methods of $m$ are the union of methods remaining in $m_1$ and $m_2$

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.
- There 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