BSV execution model and concurrent rule scheduling

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

February 25, 2013  http://csg.csail.mit.edu/6.375  L06-1

BSV Execution Model

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

Highly non-deterministic; User annotations can be used in rule selection

A legal behavior of a BSV program can be explained by observing the state updates obtained by applying only one rule at a time

One-rule-at-time semantics

Concurrent scheduling of rules

- The **one-rule-at-a-time** semantics plays the central role in defining functional correctness and verification but for meaningful hardware design it is necessary to execute multiple rules concurrently without violating the **one-rule-at-a-time** semantics.

- What do we mean by concurrent scheduling?
  - First - some hardware intuition
  - Second - semantics of rule execution
  - Third - semantics of concurrent scheduling
BSV Rule Execution

- A BSV program consists of state elements and rules, aka, Guarded Atomic Actions (GAA) that operate on the state elements.
- Application of a rule modifies some state elements of the system in a deterministic manner.

![Diagram of BSV Rule Execution]


---

some insight into Concurrent rule firing

- There are more intermediate states in the rule semantics (a state after each rule step).
- In the HW, states change only at clock edges.

![Diagram of Concurrent rule firing]

February 25, 2013  http://csg.csail.mit.edu/6.375  L06-6
Parallel execution reorders reads and writes

- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules.
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks.

Correctness

- Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule execution.
- Consequence: the HW can never reach a state unexpected in the rule semantics.
Compiling a Rule

```plaintext
rule r (f.first() > 0) {
    x <= x + 1;
    f.deq();
}
endrule
```

Combining State Updates: strawman

- \( \pi \)’s from the rules that update \( R \)
- \( \delta \)’s from the rules that update \( R \)

What if more than one rule is enabled?
Combining State Updates

The BSV compiler determines which rules among the rules whose guards are ready can be executed concurrently. It then divides the rules into disjoint sets such that the rules within each set are conflict free. Among conflicting sets of enabled rules it picks one set by some predetermined priority and this process is repeated until no rules are enabled.
A compiler test for concurrent rule firing  James Hoe, Ph.D., 2000

- Let RS(r) be the set of registers rule r may read
- Let WS(r) be the set of registers rule r may write

Rules ra and rb are conflict free (CF) if

\[ (RS(ra) \cap WS(rb) = \emptyset) \land (RS(rb) \cap WS(ra) = \emptyset) \land (WS(ra) \cap WS(rb) = \emptyset) \]

Rules ra and rb are sequentially composable (SC) (ra < rb) if

\[ (RS(rb) \cap WS(ra) = \emptyset) \land (WS(ra) \cap WS(rb) = \emptyset) \]

If Rules ra and rb conflict if they are not CF or SC

Scheduling and control logic

Compiler synthesizes a scheduler such that at any given time ϕ’s for only non-conflicting rules are true
Bluespec semantics

Bluespec: Two-Level Compilation

- Level 1 compilation
  - Bluespec (Objects, Types, Higher-order functions)
  - Rules and Actions (Term Rewriting System)
- Level 2 synthesis
  - Object code (Verilog/C)
- Lennart Augustsson
  @Sandburst 2000-2002
  - Type checking
  - Massive partial evaluation and static elaboration
  - Now we call this Guarded Atomic Actions
- James Hoe & Arvind
  @MIT 1997-2000
  - Rule conflict analysis
  - Rule scheduling
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:
- source
  - compile
  - run w/params

Hardware Toolflow:
- source
  - elaborate w/params
  - run w/params

The language after type checking and static elaboration

rule name e a
a ::= x <= e register assignment
| a ; a parallel actions
| if (e) a conditional action
| a when e guarded action
| m.g(e) action method call
| t = e ; a binding
e ::= c | t | x value read | op(e,e) method read | m.f(e) | t= e; e action call

February 25, 2013
http://csg.csail.mit.edu/6.375 L06-17
Guard Lifting rules

- All the guards can be “lifted” to the top of a rule
  - (a1 when p) ; a2 \rightarrow (a1 ; a2) when p
  - a1 ; (a2 when p) \rightarrow (a1 ; a2) when p
  - if (p when q) a \rightarrow (if (p) a) when q
  - if (p) (a when q) \rightarrow (if (p) a) when (q \mid \neg p)
  - (a when p1) when p2 \rightarrow a when (p1 \& p2)
  - x <= (e when p) \rightarrow (x <= e) when p
  - m.g_B(e when p) \rightarrow m.g_B(e) when p

Similarly for expressions ...

- Rule r (a when p) \rightarrow Rule r (if (p) a)

We will give a procedure to evaluate rules after guard lifting.

Rule evaluation

<table>
<thead>
<tr>
<th>rule name e a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ::= x &lt;= e</td>
</tr>
<tr>
<td>\mid a ; a</td>
</tr>
<tr>
<td>\mid if (e) a</td>
</tr>
<tr>
<td>\mid m.g(e)</td>
</tr>
<tr>
<td>\mid t = e ; a</td>
</tr>
</tbody>
</table>

| e ::= c | t | x | op(e,e) | m.f(e) | t = e; e |

evalA :: (Bindings, State, a) \rightarrow (Bindings, StateUpdates)

evalE :: (Bindings, State, e) \rightarrow Value
**Action evaluator**

*no method calls*

\[
\text{evalA} :: (\text{Bindings}, \text{State}, a) \rightarrow (\text{Bindings}, \text{StateUpdates})
\]

\[
\text{evalA}(bs, s, [[x <= e]]) = (bs, (x, \text{evalE}(bs, s, e)))
\]

\[
\text{evalA}(bs, s, [[a1 ; a2]]) = \begin{align*}
&\quad \text{let } (bs', u1) = \text{evalA}(bs, s, a1) \\
&\qquad (bs'', u2) = \text{evalA}(bs', s, a2) \\
&\quad \text{in } (bs'', u1 + u2)
\end{align*}
\]

\[
\text{evalA}(bs, s, [[\text{if } (e) a]]) = \begin{cases} 
\text{if } \text{evalE}(bs, s, e) \text{ then } \text{evalA}(bs, s, a) \\
\text{else } (bs, \{\})
\end{cases}
\]

\[
\text{evalA}(bs, s, [[t = e; a]]) = \begin{align*}
&\quad \text{let } v = \text{evalE}(bs, s, e) \\
&\quad \text{in } \text{evalA}(bs + (t,v), s, a)
\end{align*}
\]

Initially \(bs\) is empty and state contains old register values.

---

**Expression evaluator**

*no method calls*

\[
\text{evalE} :: (\text{Bindings}, \text{State}, \text{exp}) \rightarrow \text{Value}
\]

\[
\begin{align*}
\text{evalE}(bs, s, [[c]]) &= c & \text{if } t \text{ does not exist in } bs \\
\text{evalE}(bs, s, [[t]]) &= \text{lookup}(bs, t) & \text{if } t \text{ does exist in } bs \\
\text{evalE}(bs, s, [[x]]) &= s.x & \text{the rule is illegal}
\end{align*}
\]

\[
\text{evalE}(bs, s, [[\text{op}(e1,e2)]])) = \text{op}(\text{evalE}(bs, s, e1), \text{evalE}(bs, s, e2))
\]

Method calls can be evaluated by substituting the body of the method call, i.e., \(m.g(e)\) is \(a[e/x]\) where the definition of \(m.g\) is \(g(x) = a\).

To apply a rule, we first evaluate its guard and then if the guard is true we compute the state updates and then simultaneously update all the state variables.
Legal BSV rules

- A legal BSV rule does not contain multiple assignments to the same state element or combinational cycles.

Examples:

```verbatim
rule ra if (z>10);
  x <= x+1; endrule
rule rb;
  x <= x+1; if (p) x <= 7 endrule
rule rc;
  x <= y+1; y <= x+2 endrule
rule rd;
  t1 = f(t2); t2 = g(t1); x <= t1; endrule
```

In general, the legality of a rule can be determined only at run time.

Concurrent scheduling:

Semantic view

- Suppose rule ra a and rule rb b are legal rules and a and b are free of guards. ra and rb are concurrently schedulable, iff,
  1. rule rab (a;b) is legal
  2. for all s, (a;b)(s) = a(b(s)) or b(a(s))

Theorm 1: If rules ra and rb are conflict free (CF) then \( \forall s, (a;b)(s) = a(b(s)) = b(a(s)) \)

Theorm 2: If rules ra and rb are sequentially composable (SC) (ra<rb) then \( \forall s, (a;b)(s) = b(a(s)) \)
Example 1

rule ra if (z>10);
        x <= x+1;
endrule

rule rb if (z>20);
        y <= y+2;
endrule

rule ra_rb;
        if (z>10) x <= x+1;
        if (z>20) y <= y+2;
endrule

Example 2

rule ra if (z>10);
        x <= y+1;
endrule

rule rb if (z>20);
        y <= x+2;
endrule

rule ra_rb;
        if (z>10) x <= y+1;
        if (z>20) y <= x+2;
endrule
Example 3

rule ra if (z>10);
    x <= y+1;
endrule

rule rb if (z>20);
    y <= y+2;
endrule

{\(x_0, y_0, 30\)} \Rightarrow ra \{y_0+1, y_0, 30\}

{\(x_0, y_0, 30\)} \Rightarrow rb \{x_0, y_0+2, 30\}

{\(x_0, y_0, 30\)} \Rightarrow ra_rb \{y_0+1, y_0+2, 30\}

Example 4

rule ra;
    x <= y+1; u <= u+2;
endrule

rule rb;
    y <= y+2; v <= u+1;
endrule

rule ra_rb;
    x <= y+1; u <= u+2;
    y <= y+2; v <= u+1;
endrule

February 25, 2013
http://csg.csail.mit.edu/6.375
L06-27