Constructive Computer Architecture:  
Bluespec execution model and concurrency semantics

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

Contributors to the course material

- Arvind, Rishiyur S. Nikhil, Joel Emer, Muralidaran Vijayaraghavan
- Staff and students in 6.375 (Spring 2013), 6.S195 (Fall 2012), 6.S078 (Spring 2012)
  - Asif Khan, Richard Ruhler, Sang Woo Jun, Abhinav Agarwal, Myron King, Kermin Fleming, Ming Liu, Li-Shiuan Peh
- External
  - Prof Amey Karkare & students at IIT Kanpur
  - Prof Jihong Kim & students at Seoul Nation University
  - Prof Derek Chiou, University of Texas at Austin
  - Prof Yoav Etsion & students at Technion
Finite State Machines (Sequential Ckts)

<table>
<thead>
<tr>
<th>Present state Q1 Q2</th>
<th>Next State, Output X=0</th>
<th>Next State, Output X=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11,0</td>
<td>01,0</td>
</tr>
<tr>
<td>01</td>
<td>11,0</td>
<td>00,0</td>
</tr>
<tr>
<td>10</td>
<td>10,0</td>
<td>11,1</td>
</tr>
<tr>
<td>11</td>
<td>10,0</td>
<td>10,1</td>
</tr>
</tbody>
</table>

Typical description:
State Transition Table or Diagram
Easily translated into circuits


A computer (if fact all digital hardware) is an FSM
Neither State tables nor diagrams is suitable for describing very large digital designs
- large circuits must be described in a modular fashion
  -- as a collection of cooperating FSMs
Bluespec is a modern programming language to describe cooperating FSMs
- This lecture is about understanding the semantics of Bluespec
In this lecture we will use pseudo syntax, and assume that type checking has been performed (programs are type correct)

KBS0: A simple language for describing Sequential cKts -1

A program consists of a collection of registers \((x, y, \ldots)\) and rules

- Registers hold the state from one clock cycle to the next
- A rule specifies how the state is to be modified each clock cycle
- All registers are read at the beginning of the clock cycle and updated at the end of the clock cycle
KBS0: A simple language for describing Sequential ckts - 2

A rule is simply an action \(<a>\) described below. Expression \(<e>\) is a way of describing combinational ckts

\(<a> ::= x := <e> \quad \text{register assignment}
| <a> ; <a> \quad \text{parallel actions}
| \text{if } (<e>) <a> \quad \text{conditional action}
| \text{let } t = <e> \text{ in } <a> \quad \text{binding}

\(<e> ::= c \quad \text{constants}
| t \quad \text{value of a binding}
| x.r \quad \text{register read}
| \text{op}(<e>,<e>) \quad \text{operators like And, Or, Not, +, ...}
| \text{let } t = <e> \text{ in } <e> \quad \text{binding}

We will assume that the names in the bindings (t ...) can be defined only once (single assignment restriction)

Evaluating expressions and actions

- The state of the system \(s\) is defined as the value of all its registers
- An expression is evaluated by computing its value on the current state
- An action defines the next value of some of the state elements based on the current value of the state
- A rule is evaluated by evaluating the corresponding action and simultaneously updating all the affected state elements
Bluespec Execution Model

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

Need a evaluator to define how a rule transforms the state

KBS0 Evaluator

- We will write the evaluator as a software program using case-by-case analysis of syntax

\[
evalE :: \text{Bindings, State, e} \rightarrow \text{Value}\\
evalA :: \text{Bindings, State, a} \rightarrow \text{Bindings, StateUpdates}
\]

Bindings is a set of (variable name, value) pairs

State is a set of (register name, value) pairs.

s.x gives the value of register x in the current state

Syntax is represented as \[[...]]\]
**KBS0: Expression evaluator**

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

- \[ \text{evalE}(bs, s, [[c]]) = c \]  
  lookup t; if t does not exist in bs then the rule is illegal
- \[ \text{evalE}(bs, s, [[t]]) = bs[t] \]  
  not exist in bs then the rule is illegal
- \[ \text{evalE}(bs, s, [[x.r]]) = s[x] \]
- \[ \text{evalE}(bs, s, [[\text{op}(e1,e2)]]) = \text{op}(\text{evalE}(bs, s, [[e1]]), \text{evalE}(bs, s, [[e2]])) \]
- \[ \text{evalE}(bs, s, [[(\text{let} t = e \text{ in } e1)]])) = \begin{cases} v = \text{evalE}(bs, s, [[e]]); & \text{if evalE}(bs, s, [[e]]) \text{ then evalA}(bs, s, [[a]]) \\ \text{return} \text{evalE}(bs+(t,v), s, [[e1]]) \end{cases} \]

Initial bindings \( bs \) is empty initially

---

**KBS0: Action evaluator**

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

- \[ \text{evalA}(bs, s, [[x.w(e)]])) = (x, \text{evalE}(bs, s, [[e]])) \]
- \[ \text{evalA}(bs, s, [[a1 ; a2]]) = \begin{cases} u1 = \text{evalA}(bs, s, [[a1]]); & \text{merges two sets of updates; the rule is illegal if there are multiple updates for the same register} \\ u2 = \text{evalA}(bs', s, [[a2]]) \\ \text{return} u1 + u2 \end{cases} \]
- \[ \text{evalA}(bs, s, [[\text{if} (e) a]]) = \begin{cases} & \text{if evalE}(bs, s, [[e]]) \text{ then evalA}(bs, s, [[a]]) \\ & \text{else} \{} \end{cases} \]
- \[ \text{evalA}(bs, s, [[(\text{let} t = e \text{ in } a)]])) = \begin{cases} v = \text{evalE}(bs, s, [[e]]) \\ \text{return} \text{evalA}(bs+(t,v), s, [[a]]) \end{cases} \]

Initially \( bs \) is empty and \( s \) contains old register values
Rule evaluator

To apply a rule, we compute the state updates using EvalA and then simultaneously update all the state variables that need to be updated.

Evaluation in the presence of modules

It is easy to extend the evaluator we have shown to include non-primitive method calls:

- An action method, just like a register write, can be called at most once from a rule.
- The only additional complication is that a value method with parameters can also be called at most once from an action.
- If these conditions are violated then it is an illegal rule/action/expression.
Evaluation in the presence of guards

- In the presence of guards the expression evaluator has to return a special value – NR (for “not ready”). This ultimately affects whether an action can affect the state or not.
- Instead of complicating the evaluator we will give a procedure to lift when’s to the top of a rule. At the top level a guard behaves just like an “if”.

Guard Elimination
Guards vs If’s

- A guard on one action of a parallel group of actions affects every action within the group
  \[(a1 \text{ when } p1); a2 \implies (a1; a2) \text{ when } p1\]

- A condition of a Conditional action only affects the actions within the scope of the conditional action
  \[(\text{if } (p1) a1); a2\]
  \[p1 \text{ has no effect on } a2 \ldots\]

- Mixing ifs and whens
  \[(\text{if } (p) (a1 \text{ when } q)); a2\]
  \[= ((\text{if } (p) a1); a2) \text{ when } ((p \& q) \lor \neg p)\]
  \[= ((\text{if } (p) a1); a2) \text{ when } (q \lor \neg p)\]

Method calls have implicit guards

- Every method call, except the primitive method calls, i.e., \(x.r, x.w\), has an implicit guard associated with it
  - \(m.\text{enq}(x)\), the guard indicated whether one can enqueue into fifo \(m\) or not

- Make the guards explicit in every method call by naming the guard and separating it from the unguarded body of the method call, i.e., syntactically replace \(m.g(e)\) by
  \[m.g_B(e) \text{ when } m.g_G\]
  - Notice \(m.g_G\) has no parameter because the guard value should not depend upon the input
Make implicit guards explicit

\[
\langle a \rangle ::= \text{x.w(} <e> \text{)} \\
\quad \mid \langle a \rangle ; \langle a \rangle \\
\quad \mid \text{if } (\langle e \rangle) \langle a \rangle \\
\quad \quad \mid \text{m.g(} <e> \text{)} \\
\quad \mid \text{let } t = <e> \text{ in } \langle a \rangle \\
\quad \mid \langle a \rangle \text{ when } <e>
\]

Lifting implicit guards

rule foo if (True);
\quad (\text{if } (p) \text{ fifo.enq(8)}); \text{x.w(7)}

rule foo if (fifo.enq}_G \mid \neg p); 
\quad \text{if } (p) \text{ fifo.enq}_B(8); \text{x.w(7)}

All implicit guards are made explicit, and lifted and conjoined to the rule guard
Guard Lifting Axioms
without Let-blocks

- All the guards can be “lifted” to the top of a rule
  - \((a_1 \text{ when } p) ; a_2 \) \(\Rightarrow (a_1 ; a_2) \text{ when } p\)
  - \(a_1 ; (a_2 \text{ when } p) \) \(\Rightarrow (a_1 ; a_2) \text{ when } p\)
  - \(\text{if } (p \text{ when } q) a \) \(\Rightarrow \text{if } (p) a \text{ when } q\)
  - \(\text{if } (p) (a \text{ when } q) \) \(\Rightarrow \text{if } (p) a \text{ when } (q \mid \lnot p)\)
  - \((a \text{ when } p_1) \text{ when } p_2 \) \(\Rightarrow a \text{ when } (p_1 \& p_2)\)
  - \(m.g_B(e \text{ when } p) \) \(\Rightarrow m.g_B(e) \text{ when } p\)

similarly for expressions ...
  - Rule \(r (a \text{ when } p) \) \(\Rightarrow \text{Rule } r (\text{if } (p) a)\)

We will call this guard lifting transformation WIF, for when-to-if

A complete guard lifting procedure also requires rules for let-blocks
Let-blocks: Variable names and guards

- let t = e in f(t)
- Since e can have a guard, a variable name, t, can also have an implicit guard
- Essentially every expression has two parts: unguarded and guarded and consequently t has two parts $t_B$ and $t_G$
- Each use of the variable name has to be replaced by $(t_B \text{ when } t_G)$

Lift procedure

- $LWE :: (\text{Bindings, Exp}) \rightarrow (\text{Bindings, Exp}_B, \text{Exp}_G)$
- $LW :: (\text{Bindings, Exp}) \rightarrow (\text{Bindings, Action}_B, \text{Exp}_G)$
- Returned exp, actions and bindings are all free of when’s

- Bindings is a collection of (t,e) pairs where e is restricted to be
  
  \begin{align*}
  &c \mid x.r \mid t \mid \text{op}(t,t) \mid m.h(t) \mid \{\text{body: } t, \text{guard: } t\}
  \end{align*}

- The bindings of the type (t, {body:tx, guard:ty}) are not needed after When Lifting because all such t’s would have been eliminated from the returned expressions
Bindings

The bindings that LW and LWE return are simply a collection of \((t, e)\) pairs where \(e\) is restricted to be
\[
\begin{align*}
&c | x.r | x.r0 | x.r1 | t | \text{op}(t, t) | m.h(t) \\
&\{\text{body: } t, \text{guard: } t\}
\end{align*}
\]

The bindings of the type \((t, \{\text{body: } tx, \text{guard: } ty\})\) are not needed after When Lifting because all such \(t\)'s would have been eliminated from the returned expressions.

LWE: procedure for lifting when's in expressions

\[
\text{LWE} :: (\text{Bindings}, \text{Exp}) \rightarrow (\text{Bindings}, \text{Exp}_B, \text{Exp}_G)
\]

\[
\begin{align*}
\text{LWE}(bs, [[c]]) &= (bs, c, T) & \text{LWE}(bs, [[x.r]]) &= (bs, x.r, T) \\
\text{LWE}(bs, [[x.r0]]) &= (bs, x.r0, T) & \text{LWE}(bs, [[x.r1]]) &= (bs, x.r1, T) \\
\text{LWE}(bs, [[t]]) &= (bs, bs[t].\text{body}, bs[t].\text{guard})
\end{align*}
\]

\[
\begin{align*}
\text{LWE}(bs, [[\text{Op}(e1, e2)]]) &= \{bs1, t1_B, t1_G = \text{LWE}(bs, [[e1]]); \\
&bs2, t2_B, t2_G = \text{LWE}(bs1, [[e2]]); \\
&\text{return } bs2, \text{Op}(t1_B, t2_B), (t1_G & t2_G)\}
\end{align*}
\]

\[
\begin{align*}
\text{LWE}(bs, [[\text{m.h}(e)]]) &= \{bs1, t_B, t_G = \text{LWE}(bs, [[e]]); \\
&\text{return } bs1, \text{m.h}(t_B), (t_G & \text{m.h}(t_G))\}
\end{align*}
\]

\[
\begin{align*}
\text{LWE}(bs, [[e1 \text{ when } e2]]) &= \{bs1, t1_B, t1_G = \text{LWE}(bs, [[e1]]); \\
&bs2, t2_B, t2_G = \text{LWE}(bs1, [[e2]]); \\
&bs3 = bs2 + (tx, t2_B & t2_G) \\
&\text{return } bs3, t1_B, (tx & t1_G)\}
\end{align*}
\]

\[
\begin{align*}
\text{LWE}(bs, [[\text{let } t=e1 \text{ in } e2]]) &= \{bs1, t_B, t_G = \text{LWE}(bs, [[e1]]); \\
&bs2 = bs1 + (tx, t_B) + (ty, t_G) \\
&\text{return } \text{LWE}(bs2, [[e2]])\}
\end{align*}
\]
**LW: procedure for lifting when’s in actions**

\[ \text{LW :: (Bindings, Exp) -> (Bindings, Action}_B, \text{ Exp}_G) \]

\[
\begin{align*}
\text{LW (bs, [[x.w(e)]])} &= \{bs_1, t_B, t_G = \text{LWE}(bs, [[e]]); \\
&\quad \text{return } bs_1, x.w(t_B), t_G\} \\
\text{LW (bs, [[m.g(e)]])} &= \{bs_1, t_B, t_G = \text{LWE}(bs, [[e]]); \\
&\quad \text{return } bs_1, m.g_B(t_B), (t_G & m.g_G)\} \\
\text{LW (bs, [[a1;a2]])} &= \{bs_1, a_1_B, g_1 = \text{LW}(bs, [[a1]]); \\
&\quad bs_2, a_2_B, g_2 = \text{LW}(bs_1, [[a2]]); \\
&\quad \text{return } bs_2, (a_1_B; a_2_B), (g_1 & g_2)\} \\
\text{LW (bs, [[if (e) a]])} &= \{bs_1, t_B, t_G = \text{LWE}(bs, [[e]]); \\
&\quad bs_2, a_B, g = \text{LW}(bs_1, [[a]]); \\
&\quad bs_3 = bs_2 + (t_B, t_G) + (t_G, t_B); \\
&\quad \text{return } bs_3, a_B, (g \cup tx) \& ty\} \\
\text{LW (bs, [[a when e]])} &= \{bs_1, t_B, t_G = \text{LWE}(bs, [[e]]); \\
&\quad bs_2, a_B, g = \text{LW}(bs_1, [[a]]); \\
&\quad \text{return } bs_2 + (t_B, t_G) + (t_G, t_B); \\
\text{LW (bs, [[let t=e in a]])} &= \{bs_1, t_B, t_G = \text{LWE}(bs, [[e]]); \\
&\quad bs_2 = bs_1 + (t_B, t_G) + (t_G, t_B)\}
\end{align*}
\]

\[ \text{tx, ty are new variable} \quad \text{return } \text{LW}(bs_2, [[a]]) \]

---

**WIF: when-to-if transformation**

- **Given rule** ra a,
  
  WIF(ra) returns
  
  rule ra (let bs in (if (g) a_B))

  assuming LW({}), a) returns (bs, a_B, g)

- **Notice,**
  - WIF(ra) has no when’s
  - WIF(a1;a2) ≠ (WIF(a1);WIF(a2))