Constructive Computer Architecture:

Bluespec execution model and concurrency semantics

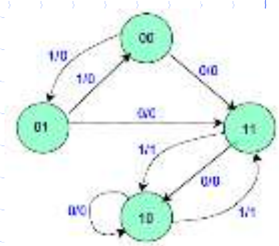
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

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

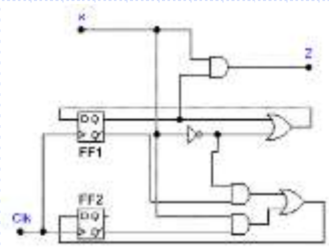
Finite State Machines (Sequential Ckts)

Present state Q1 Q2	Next State, Output X=0	Next State, Output X=1
00	11,0	01,0
01	11,0	00,0
10	10,0	11,1
11	10,0	10,1



Typical description:
State Transition Table or Diagram

Easily translated into circuits



http://www.ee.usyd.edu.au/tutorials/digital_tutorial/part3/t-diag.htm

Finite State Machines (Sequential Ckts)

- 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, ...) 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

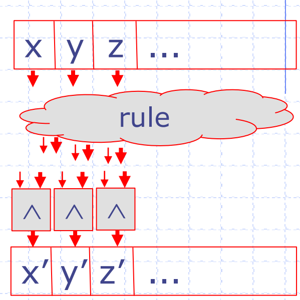
```
<a> ::= x:= <e>                 register assignment
     | <a> ; <a>                parallel actions
     | if (<e>) <a>             conditional action
     | let t = <e> in <a>       binding

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

Highly non-deterministic; 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

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 :: (Bindings, State, e) -> Value
evalA :: (Bindings, State, a)    -> (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

evalE :: (Bindings, State, exp) -> Value

```
evalE (bs, s, [[c]])  = c
evalE (bs, s, [[t]])  = bs[t]
evalE (bs, s, [[x.r]])  = s[x]
evalE (bs, s, [[op(e1,e2)]]) =
        op(evalE(bs, s, [[e1]]), evalE(bs, s, [[e2]]))
evalE (bs, s, [[(let t = e in e1)]]) =
        {  v = evalE(bs, s, [[e]]);
          return evalE(bs+(t,v), s, [[e1]])}
```

lookup t; if t does not exist in bs then the rule is illegal

add a new binding to bs. The operation is illegal if t is already present in bs

Bindings bs is empty initially

KBS0: Action evaluator

evalA :: (Bindings, State, a) -> StateUpdates

```
evalA (bs, s, [[x.w(e)]]) = (x, evalE(bs, s, [[e]]))
evalA (bs, s, [[a1 ; a2]]) =
   { u1 = evalA(bs, s, [[a1]]);
     u2 = evalA(bs', s, [[a2]])
    return u1 + u2 }
evalA (bs, s, [[if (e) a]]) =
   if evalE(bs, s, [[e]]) then evalA(bs, s, [[a]])
                        else  {}
evalA (bs, s, [[(let t = e in a)]]) =
   {  v = evalE(bs, s, [[e]])
   return evalA(bs+(t,v), s, [[a]]) }
```

merges two sets of updates; the rule is illegal if there are multiple updates for the same register

extends the bindings by including one for t

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
  - It 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 when p1); a2

  ==> (a1; a2) when p1
- A condition of a Conditional action only affects the actions within the scope of the conditional action

  (if (p1) a1); a2

  p1 has no effect on a2 ...
- Mixing ifs and whens

  (if (p) (a1 when q)) ; a2

  ≡ ((if (p) a1); a2) when ((p && q) | !p)

  ≡ ((if (p) a1); a2) when (q | !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.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

  $m.g_B(e)$ 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

<a> ::= x.w(<e>)
  | <a> ; <a>
  | if (<e>) <a>
  ~~| m.g(<e>)~~ m.g_B(<e>) when m.g_G
  | let t = <e> in <a>
  | <a> when <e>

<a> ::= <a> ; <a>
  | if (<e>) <a>
  | m.g(<e>) methods without guards
  | let t = <e> in <a>
  | <a> when <e>

The new kernel language

Lifting implicit guards

rule foo if (True);
  (if (p) fifo.enq(8)); x.w(7)

rule foo if (fifo.enq_G | !p);
  if (p) fifo.enq_B(8); 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
  - (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 | !p)
  - (a when p1) when p2 $\Rightarrow$ a when (p1 & p2)
  - 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 call this guard lifting transformation WIF, for when-to-if

A complete guard lifting procedure also requires rules for let-blocks

Optional: A complete procedure for guard lifting

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 when t_G)

Lift procedure

LWE :: (Bindings, Exp) -> (Bindings, Exp_B, Exp_G)
LW :: (Bindings, Exp) -> (Bindings, $Action_B$, 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

  c | x.r | t | op(t,t) | m.h(t) | {body: t, guard: t}

- 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
  c | x.r | x.r0| x.r1 | t | op(t,t) | m.h(t)
  | {body: t, guard: t}
- 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

LWE: procedure for lifting when's in expressions

LWE :: (Bindings, Exp) -> (Bindings, Exp_B, Exp_G)

```
LWE (bs, [[c]]) = (bs, c, T) ;          LWE (bs, [[x.r]]) = (bs, x.r, T)
LWE (bs, [[x.r0]]) = (bs, x.r0, T);     LWE (bs, [[x.r1]]) = (bs, x.r1, T)
LWE (bs, [[t]]) = (bs, bs[t].body, bs[t].guard)
LWE (bs, [[Op(e1,e2)]]) = {bs1, t1_B , t1_G = LWE(bs, [[e1]]);
                           bs2, t2_B , t2_G = LWE(bs1, [[e2]]);
                           return bs2, Op(t1_B, t2_B), (t1_G&t2_G)}
LWE(bs, [[m.h(e)]])     = {bs1, t_B , t_G  = LWE(bs, [[e]]);
                           return bs1, m.h_B(t_B), (t_G&m.h_G)}
LWE (bs, [[e1 when e2]]) = {bs1, t1_B , t1_G  = LWE(bs, [[e1]]);
                            bs2, t2_B , t2_G  = LWE(bs1, [[e2]]);
                            bs3 = bs2+(tx, t2_B&t2_G)
                            return bs3, t1_B, (tx&t1_G)}
LWE(bs, [[let t=e1 in e2]]) = {bs1, t_B , t_G = LWE(bs, [[e1]]);
                               bs2 = bs1+(tx,t_B)+(ty,t_G)
tx, ty are                          +(t,{body:tx,guard:ty})
new variable                   return LWE(bs2, [[e2]]}
```

LW: procedure for lifting when's in actions

LW :: (Bindings, Exp) -> (Bindings, Action_B, Exp_G)

LW (bs, [[x.w(e)]]) = {bs1, t_B , t_G = LWE(bs, [[e]]);
    return bs1, x.w(t_B), t_G}
LW (bs, [[m.g(e)]]) = {bs1, t_B , t_G = LWE(bs, [[e]]);
    return bs1, m.g_B(t_B), (t_G&m.g_G)}
LW (bs, [[a1;a2]]) = {bs1, $a1_B$, g1 = LW(bs, [[a1]]);
    bs2, $a2_B$, g2 = LW(bs1, [[a2]]);
    return bs2, ($a1_B$; $a2_B$), (g1&g2)}
LW (bs, [[if (e) a]]) = {bs1, t_B , t_G = LWE(bs, [[e]]);
    bs2, a_B , g = LW(bs1, [[a]]);
    bs3 = bs2+(tx,t_B)+(ty,t_G)
    return bs3, a_B, (g | !tx) & ty)}
LW (bs, [[a when e]]) = {bs1, t_B , t_G = LWE(bs, [[e]]);
    bs2, a_B , g = LW(bs1, [[a]]);
    return bs2+(tx, t_B&t_G), a_B, (tx&g)}
LW(bs, [[let t=e in a]]) = {bs1, t_B , t_G = LWE(bs, [[e]]);
    bs2 = bs1+(tx,t_B)+(ty,t_G)
    +(t,{body:tx,guard:ty})
    return LW(bs2, [[a]]}

tx, ty are new variable

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))