Modules with Guarded Interfaces

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
A hardware module for computing GCD

Euclid’s algorithm for computing the Greatest Common Divisor (GCD):

\[
\begin{array}{c|c}
15 & 6 \\
9 & 6 \\
3 & 6 \\
6 & 3 \\
3 & 3 \\
0 & 3
\end{array}
\]

\[\text{answer: } 3\]

```python
def gcd(a, b):
    if a == 0: return b  # stop
    elif a >= b: return gcd(a-b, b)  # subtract
    else: return gcd(b, a)  # swap
```
interface GCD;
  method Action start (Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) getResult;
  method Bool busy;
  method Bool ready;
endinterface

GCD can be started if the module is not busy; Results can be read when ready
module mkGCD (GCD);
  Reg#(Bit#(32)) x <- mkReg(0);  Reg#(Bit#(32)) y <- mkReg(0);
  Reg#(Bool) busy_flag <- mkReg(False);
rule gcd;
  if (x >= y) begin x <= x - y; end //subtract
  else if (x != 0) begin x <= y; y <= x; end //swap
endrule
method Action start(Bit#(32) a, Bit#(32) b);
  x <= a; y <= b; busy_flag <= True;
endmethod
method ActionValue#(Bit#(32)) getResult;
  busy_flag <= False; return y;
endmethod
method Bool busy; return busy_flag;
endmethod
method Bool ready; return (busy_flag && (x==0));
endmethod
endmodule

GCD in BSV

Assume b /= 0

start should be called only if the module is not busy;
getResult should be called only when ready is true.
A rule is a collection of actions, which invoke methods

All actions in a rule execute in parallel

A rule can execute any time and when it executes all of its actions must execute atomically
Parallel Composition of Actions & Double-Writes

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, 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.
FIFO Module Interface

interface Fifo#(numeric type size, type t);
  method Bool notFull;
  method Bool notEmpty;
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface

- enq should be called only if notFull returns True;
- deq and first should be called only if notEmpty returns True
module mkFifo (Fifo#(1, t));
  Reg#(t) d <- mkRegU;
  Reg#(Bool) v <- mkReg(False);
method Bool notFull;
  return !v;
endmethod
method Bool notEmpty;
  return v;
endmethod
method Action enq(t x);
  v <= True; d <= x;
endmethod
method Action deq;
  v <= False;
endmethod
method t first;
  return d;
endmethod
endmodule

An Implementation:
One-Element FIFO

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

```
rule stream;
  if (inQ.notEmpty && outQ.notFull)
    begin
      outQ.enq(f(inQ.first));
      inQ.deq;
    end
endrule
```

Boolean “AND” operation
Streaming a module

```plaintext
rule invokeGCD;
    if(inQ.notEmpty && !gcd.busy)
    begin let x = tpl_1(inQ.first);
          let y = tpl_2(inQ.first);
          gcd.start(x,y); inQ.deq;
    end
endrule

rule getResult;
    if(outQ.notFull && gcd.ready)
    begin let x <- gcd.result; outQ.enq(x); end
endrule
```

Action value method
Guarded interfaces

Make the life of the programmers easier: Include some checks (readyness, fullness, ...) in the method definition itself, so that the user does not have to test the applicability of the method from outside.

Guarded Interface:
- Every method has a guard (rdy wire)
- The value returned by a method is meaningful only if its guard is true
- Every action method has an enable signal (en wire) and it can be invoked (en can be set to true) only if its guard is true

```vhdl
interface Fifo#(numeric type size, type t);
    method Action enq(t x);
    method Action deq;
    method t first;
endinterface
```

notice, en and rdy wires are implicit
One-Element FIFO Implementation with guards

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

Notice, no semicolon turns the if into a guard
Rules with guards

Like a method, a rule can also have a guard.

A rule can execute only if its guard is true, i.e., if the guard is false the rule has no effect.

True guards can be omitted.

Syntax: In rules, “if” is optional before the guard!

```
rule foo if (p);
begin x1 <= e1; x2 <= e2 end
endrule
```
Streaming a function using a FIFO with guarded interfaces

$$\text{rule stream;}
\text{ if(inQ.notEmpty \&\& outQ.notFull) begin outQ.enq(f(inQ.first)); inQ.deq; end endrule}
$$

$$\text{rule stream (inQ.notEmpty \&\& outQ.notFull); outQ.enq(f(inQ.first)); inQ.deq; endrule}
$$

The implicit guards of the method calls are sufficient here
Interface without guards

```verilog
interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
    method Bool busy;
    method Bool ready;
endinterface
```

Interface with guards

```verilog
interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
    method Bool busy;
    method Bool ready;
endinterface
```
GCD with guarded interfaces
implementation

module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy <- mkReg(False);
rule gcd;
    if (x >= y) begin x <= x - y; end //subtract
    else if (x != 0) begin x <= y; y <= x; end //swap
endrule

method Action start(Bit#(32) a, Bit#(32) b) if (!busy);
    x <= a; y <= b; busy <= True;
endmethod

method ActionValue (Bit#(32)) getResult if (busy && (x==0));
    busy <= False; return y;
endmethod
endmodule
Using the GCD module with guarded interfaces

rule invokeGCD;
  if(inQ.notEmpty && !gcd.busy)
    begin let x = tpl_1(inQ.first);
      let y = tpl_2(inQ.first);
      gcd.start(x,y); inQ.deq;
    end
endrule

rule getResult;
  if(outQ.notFull && gcd.ready)
    begin let x <- gcd.getResult; outQ.enq(x);
    end
endrule
Using the GCD module with guarded interfaces

A rule can be executed only if the guards of all of its actions are true

rule invokeGCD;
  let x = tpl_1(inQ.first);
  let y = tpl_2(inQ.first);
  gcd.start(x,y);
  inQ.deq;
endrule

rule getResult;
  let x <- gcd.getResult;
  outQ.enq(x);
endrule
Switch using FIFOs with guarded interfaces

```
rule switch;
    if (inQ.first.color == Red) begin
        redQ.enq (inQ.first.value); inQ.deq;
    end else begin // color is Green
        greenQ.enq(inQ.first.value); inQ.deq;
    end
endrule
```

What is the implicit guard?

```
inQ.notEmpty ? ((inQ.first.color == Red) ? redQ.notFull : greenQ.notFull) : False
```
Switch:
Split into two rules

```plaintext
rule switchRed if (inQ.first.color == Red);
    redQ.enq(inQ.first.value); inQ.deq;
endrule;

rule switchGreen if (inQ.first.color == Green);
    greenQ.enq(inQ.first.value); inQ.deq;
endrule;
```

Only one of the rules can be active in a given state.
Guards vs Ifs

method Action enq(t x) if (!v);
  v <= True; d <= x;
endmethod

versus

method Action enq(t x);
  if (!v) begin v <= True; d <= x; end
endmethod

guard is !v; enq can be applied only if v is false

guard is True, i.e., the method is always applicable.
if v is true then x would get lost; bad
Observation

- These sample programs are not very complex and yet it would have been tedious to express these programs in a state table or as a circuit directly.
- Parallel Execution of rules is not allowed if it can cause a double write error.

More on parallel execution of rules after the Spring Break.