Sequential Circuits as Modules with Interfaces

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.
Shortcomings of the current hardware design practice

- Complex hardware is viewed as a bunch of modules whose input/output wires are connected in some manner.

- This view is technically correct but not helpful in the design process. It does not support adequately:
  - IP reuse
  - Multiple instantiations of a module for different performance and application requirements
  - Packaging of IP so that modules can be assembled easily to build a large system (black-box use model)
  - Modular refinement, i.e., ability to refine a module without understanding the whole design
Difficulty in IP reuse

Example: Commercially available FIFO module
First-In-first-Out

An error occurs if a push is attempted while the FIFO is full.

Thus, there is no conflict in a simultaneous push and pop when the FIFO is full. A simultaneous push and pop cannot occur when the FIFO is empty, since there is no pop data to prefetch. However, push data is captured in the FIFO.

A pop operation occurs when pop_req_n is asserted (LOW), as long as the FIFO is not empty. Asserting pop_req_n causes the internal read pointer to be incremented on the next rising edge of clk. Thus, the RAM read data must be captured on the clk following the assertion of pop_req_n.

This description
1. mixes functionality and implementation
2. defines constraints on use in an ad hoc manner

a better interface definition is needed
Bluespec: Sequential Circuit as a module with Interface

- A module has internal state
- The internal state can only be read and manipulated by the (interface) methods
- An action specifies which state elements are to be modified
- Actions are *atomic* -- either all the specified state elements are modified or none of them are modified (no partially modified state is visible)

A module in Bluespec is like a class definition in Java or C++
Modulo-4 counter in Bluespec

interface Counter;
  method Action inc;
  method Bit#(2) read;
endinterface

module mkCounter(Counter);
  Reg#(Bit#(2)) cnt <- mkReg(0);
  method Action inc;
    cnt <= {cnt[1]^cnt[0],~cnt[0]};
  endmethod
  method Bit#(2) read;
    return cnt;
  endmethod
endmodule

State specification

Initial value

Action to specify how the value of the cnt is to be set

q0^{t+1} = \sim q0^t
q1^{t+1} = q1^t \oplus q0^t

Implementation
Modulo-4 counter

The generated circuit

```verilog
module moduloCounter(Counter);
    Reg#(Bit#(2)) cnt <- mkReg(0);
    method Action inc;
        cnt <= {cnt[1]^cnt[0], ~cnt[0]};
    endmethod
    method Bit#(2) read;
        return cnt;
    endmethod
endmodule
```
Another Example: FIFO
First-In-First-Out queue

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

Interface of a module defines its type
module mkFifo (Fifo#(1, t)) provisos (Bits#(t, tSz));
    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

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
Streaming a function

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

Boolean “AND” operation
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

```plaintext
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) if (!v);
    v <= True; d <= x;
endmethod
method Action deq if (v);
    v <= False;
endmethod
method t first if (v);
    return d;
endmethod
endmodule

Notice, no semicolon turns the if into a guard
Streaming a function using a FIFO with guarded interfaces

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

The implicit guards of the method calls are sufficient because a rule can execute only if the guards of all of its method calls are true.
GCD with and without guards

Interface without guards

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

- start should be called only if the module is not busy;
- getResult should be called only when ready is true

Interface with guards
GCD with Guards

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
endmodule

GCD with Guards

Assume b != 0
Rule gcd will execute repeatedly until x becomes 0
Guard?
Rule

A module may contain rules

```
rule gcd;
    if (x >= y) begin x <= x - y; end //subtract
    else if (x != 0) begin x^{t+1} <= y; y^{t+1} <= x; end //swap
endrule
```

- 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

What is meaning of this? Swap!
Rules with guards

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

```
rule foo if (p):
    guard
begin x1 <= e1; x2 <= e2; end
endrule
```

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

- A rule can execute only if it’s guard is true, i.e., if the guard is false the rule has no effect
- True guards can be omitted
- An alternative way to write the gcd rule:

```
rule gcdSubtract if (x >= y);
    x <= x - y;
endrule
rule gcdSwap if !(x >= y) && (x != 0);
    x <= y; y <= x;
endrule
```
Parallel Composition of Actions & Double-Writes

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

- Parallel composition, and consequently a rule containing it, is illegal if a double-write possibility exists
- The Bluespec compiler rejects a program if there is any possibility of a double write in a rule or method
Streaming a module

Rules are the glue that connects modules

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

```
rule getResult ; if(gcd.getResult.rdy
  let x <- gcd.getResult;   && outQ.enq.rdy)
  outQ.enq(x);
endrule
```

Implicit guards?

Action value method
Bluespec: A new way of expressing behavior using Guarded Atomic Actions

- A module, like an object in OO languages, has a well-defined interface
- However, unlike software OO languages, the interface methods are **guarded**; it can be applied only if it is “ready”
- The modules are glued together (composed) using *atomic actions*, which call the methods
- An atomic action can execute only if all the called methods can be executed simultaneously
Take-home problem

What is the difference in the behavior of these two implementations of enq? Are they both correct?

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

versus

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