Sequential Circuits: Modules with Guarded Interfaces

Arvind
Computer Science & Artificial Intelligence Lab
M.I.T.
Types of Methods

- **Value method**: Don’t update the state of the module, only observe the internal state
  - Example: mod4counter.read, gcd.busy, gcd.ready

- **Action method**: Only updates the state of the module, doesn’t return any value
  - Example: mod4counter.inc, gcd.start
  - The circuit for an Action method contains an *enable* wire, which must be true for the call to take effect

- **ActionValue#(t)**: Updates the state of the module and returns a value of type t.
  - Example: gcd.getResult
  - The circuit for an ActionValue method contains an *enable* wire, which must be true for the call to take effect

All methods can have input arguments e.g. gcd.start(x,y)
Method calls

- Value method
  - `let` `counterValue = mod4counter.read;
  - `Bool isGcdBusy = gcd.busy;

- Action method
  - `mod4counter.inc;
  - `gcd.start(13,27);

- ActionValue#(t)
  - `let` `resultGcd <- gcd.getResult;
  - Notice the use of ‘<-’ instead of ‘=’
  - Suppose we wrote
    - `let` `badResultGCD = gcd.getResult;
    - then the type of `badResultGCD` would be `ActionValue#(t)`
      instead of `t`.
  - ‘=’ just names the value on the right hand side while
    ‘<-’ indicates a side effect in addition to a return value
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
    = busy_flag;

method Bool ready
    = (x==0);
endmodule

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

A module may contain rules

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

- A rule has a name (e.g., gcd)
- 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!
Parallel Composition of Actions & Double-Writes

rule one;
  y <= 3; x <= 5; x <= 7; endrule  \hspace{1cm} \text{Double write}

rule two;
  y <= 3; if (b) x <= 7; else x <= 5; endrule \hspace{1cm} \text{No double write}

rule three;
  y <= 3; x <= 5; if (b) x <= 7; endrule \hspace{1cm} \text{Possibility of a double write}

- Parallel composition, and consequently a rule containing it, is illegal if a double-write possibility exists
- The Bluespec compiler \textit{rejects} a program if there is any possibility of a double write in a rule or method
A fifo is an important data structure which is used extensively both in hardware and software to connect *things* together.

- A producer enqueues values into the fifo.
- A consumer dequeues values from the fifo.
- Dequeued values come out in the same order in which they were enqueued (i.e. First In, First Out).
- In hardware, fifo have fixed size which is often as small as 1, and therefore the producer blocks when enqueueing into a full fifo and the consumer blocks when dequeueing from an empty fifo.
FIFO in hardware
First-In-First-Out queue

```plaintext
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 v notFull;
        return !v;
    endmethod
    method Bool v 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 explicitly 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

```verilog
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
```

Guard expression is what is connected to the rdy wire of a method

Syntax: lack of 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
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) <<< if (!busy_flag);
    x <= a; y <= b; busy_flag <= True;
endmethod

method ActionValue#(Bit#(32)) getResult <<< if (busy_flag&&x==0);
    busy_flag <= False; return y;
endmethod

endmodule

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

- 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. Equivalently, the absence of a guard means the guard is always true

- 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
  ```

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

- Suppose we have a queue of pairs of numbers and we want to compute their GCDs and put the results in an output queue.
- We can build such a system by creating the following modules:
  ```
  Fifo#(1, Vector#(2, t)) inQ <- mkFifo;  
  Fifo#(1, t) outQ <- mkFifo;  
  GCD gcd <- mkGCD;  
  ```
- To glue these modules together we define two rules:
  - `invokeGCD` to push data from `inQ` into `gcd`
  - `getResult` to fetch result from `gcd` and put it into `outQ`
Streaming a module: code

```
rule invokeGCD
  if(inQ.first.rdy && inQ.deq.rdy && gcd.start.rdy);
  let x = inQ.first[0];
  let y = inQ.first[1];
  gcd.start(x,y);
  inQ.deq;
endrule

rule getResult
  if(gcd.getResult.rdy && outQ.enq.rdy);
  let x <- gcd.getResult;
  outQ.enq(x);
endrule
```
Power of Abstraction: 
Another GCD implementation

- A GCD module with the same interface but with twice the throughput; uses two gcd modules in parallel
- $\text{turnI}$ is used by the start method to direct the input to the gcd whose turn it is and then $\text{turnI}$ is flipped
- Similarly, $\text{turnO}$ is used by $\text{getResult}$ to get the output from the appropriate gcd, and then $\text{turnO}$ is flipped

```verilog
interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface
```
module mkMultiGCD (GCD);
    GCD gcd1 <- mkGCD();
    GCD gcd2 <- mkGCD();
    Reg#(Bool) turnI <- mkReg(False);
    Reg#(Bool) turnO <- mkReg(False);
    method Action start(Bit#(32) a, Bit#(32) b);
        if (turnI) gcd1.start(a,b); else gcd2.start(a,b);
        turnI <= !turnI;
    endmethod
    method ActionValue (Bit#(32)) getResult;
        Bit#(32) y;
        if (turnO) y <- gcd1.getResult
        else y <- gcd2.getResult;
        turnO <= !turnO
        return y;
    endmethod
endmodule

interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface
Summary

- Modules with guarded interfaces is a new way of expressing sequential circuits
- 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 compiler ensures that a method is enabled only when it is ready
- The modules are glued together (composed) using *atomic actions*, which call methods
- An atomic action can execute only if all the called methods can be executed simultaneously

next lecture - Hardware synthesis
Take-home problem

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

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

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