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
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
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 "Intellectual Property"
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
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)
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

data_in  data_out
push_req_n  full
pop_req_n  empty
clk
rstn
Difficulty in IP reuse

Example: Commercially available FIFO module

First-In-first-Out

data_in data_out
push_req_n full
pop_req_n empty
clk
rstn
Difficulty in IP reuse

Example: Commercially available FIFO module

An error occurs if a push is attempted while the FIFO is full.
Difficulty in IP reuse

Example: Commercially available FIFO module

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.
Difficulty in IP reuse

Example: Commercially available FIFO module

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.
Difficulty in IP reuse

Example: Commercially available FIFO module

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 \( \text{pop}\_\text{req}\_n \) is asserted (LOW), as long as the FIFO is not empty. Asserting \( \text{pop}\_\text{req}\_n \) causes the internal read pointer to be incremented on the next rising edge of \( \text{clk} \). Thus, the RAM read data must be captured on the \( \text{clk} \) following the assertion of \( \text{pop}\_\text{req}\_n \).

These constraints are spread over many pages of the documentation...
Difficulty in IP reuse

Example: Commercially available FIFO module

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 \texttt{pop\_req\_n} is asserted (LOW), as long as the FIFO is not empty. Asserting \texttt{pop\_req\_n} causes the internal read pointer to be incremented on the next rising edge of \texttt{clk}. Thus, the RAM read data must be captured on the \texttt{clk} following the assertion of \texttt{pop\_req\_n}.

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

Example: Commercially available FIFO module

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

interface Counter;
  method Action inc;
  method Bit#(2) read;
endinterface
Bluespec: Sequential Circuit as a module with Interface

- A module has internal state

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

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

```
Modulo-4 counter
```

- inc
- read

2
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

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

interface Counter;
  method Action inc;
  method Bit#(2) read;
endinterface
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 \textit{atomic} -- either all the specified state elements are modified or none of them are modified (no partially modified state is visible)

\textbf{A module in Bluespec is like a class definition in Java or C++}
interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface
Modulo-4 counter in Bluespec

module mkCounter(Counter);

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

endmodule
Modulo-4 counter in Bluespec

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

module mkCounter(Counter);
  method Action inc;
  method Bit#(2) read;
endmodule

Implementation
```
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;

        method Bit#(2) read;

endmodule
```

Implementation
Modulo-4 counter in Bluespec

```verilog
module mkCounter(Counter);
  Reg#(Bit#(2)) cnt <- mkReg(0);
  method Action inc;
  method Bit#(2) read;
endmodule
```

State specification

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

Implementation
Modulo-4 counter in Bluespec

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

module mkCounter(Counter);
  Reg#(Bit#(2)) cnt <- mkReg(0);
  method Action inc;
    #1 inc;
  endmethod

  method Bit#(2) read;
    #1 read;
  endmethod
endmodule
```

Implementation
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;

  method Bit#(2) read;
endmodule

Implementation
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;

        method Bit#(2) read;
            return cnt;
        endmethod
    endmethod
endmodule
```

Implementation
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

Implementation
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

Implementation

Action to specify how the value of the cnt is to be set
Modulo-4 counter in Bluespec

```
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
```

Implementation
Modulo-4 counter in Bluespec

```
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
```

Implementation

\[
q_0^{t+1} = \overline{q_0^t}
\]
\[
q_1^{t+1} = q_1^t \oplus q_0^t
\]
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
```

Implementation

\[
\begin{align*}
q_0^{t+1} &= \overline{q_0^t} \\
q_1^{t+1} &= q_1^t \oplus q_0^t
\end{align*}
\]
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
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
Modulo-4 counter

The generated circuit

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
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

```verilog
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
```

![FIFO Diagram](image-url)
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
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
Another Example: FIFO
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))

method Bool notFull;

method Bool notEmpty;

method Action enq(t x);

method Action deq;

method t first;

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
An Implementation: One-Element FIFO

module mkFifo (Fifo #(1, t))
    Reg#(t)    d <- mkRegU;
    Reg#(Bool) v <- mkReg(False);
method Bool notFull;

method Bool notEmpty;

method Action enq(t x);

method Action deq;

method t first;
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
module mkFifo (Fifo#(1, t))
    Reg#(t) d <- mkRegU;
    Reg#(Bool) v <- mkReg(False);
method Bool notFull;

method Bool notEmpty;

method Action enq(t x);

method Action deq;

method t first;
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
module mkFifo (Fifo#(1, t))
    Reg#(t)  d  <- mkRegU;
    Reg#(Bool)  v  <- mkReg(False);
    method Bool notFull;
        return !v;
    endmethod
    method Bool notEmpty;
method Action enq(t x);
method Action deq;
method t first;
endmodule
An Implementation: One-Element FIFO

```verilog
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);
method Action deq;
method t first;
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
```
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;

method t first;

endmodule
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;
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
An Implementation: One-Element FIFO

```verilog
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
```

```verilog
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
```
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

An Implementation: One-Element FIFO

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
Streaming a function

rule stream;
  if (inQnotEmpty && outQnotFull)
    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 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)
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)
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
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
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

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

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
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
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
    v  <= False;
endmethod
method t first
    return d;
endmethod
endmodule

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
    v <= False;
endmethod
method t first
    return d;
endmethod
endmodule

Notice, no semicolon turns the if into a guard
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
    return d;
endmethod
endmodule
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
Streaming a function using a FIFO with guarded interfaces

```plaintext
rule stream;
  if(inQ.notEmpty && outQ.notFull)
    begin outQ.enq(f(inQ.first)); inQ.deq; end
endrule
```
Streaming a function using a FIFO with guarded interfaces

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

Interface with guards

```
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 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
```

Interface with guards

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

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

module mkGCD (GCD);

method Action start(Bit#(32) a, Bit#(32) b);

method ActionValue#(Bit#(32)) getResult;

endmodule

interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);

    method Action start(Bit#(32) a, Bit#(32) b) ;

    method ActionValue#(Bit#(32)) getResult ;

endmodule

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

```
module mkGCD (GCD);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  Reg#(Bool) busy_flag <- mkReg(False);

  method Action start(Bit#(32) a, Bit#(32) b);

  method ActionValue#(Bit#(32)) getResult;

endmodule
```

```
interface GCD;
  method Action start (Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) getResult;
endinterface
```
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);

method Action start(Bit#(32) a, Bit#(32) b);
    x <= a; y <= b; busy_flag <= True;
endmethod
method ActionValue#(Bit#(32)) getResult;
endmodule

interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);

method Action start(Bit#(32) a, Bit#(32) b);
    x <= a; y <= b; busy_flag <= True;
endmethod
method ActionValue#(Bit#(32)) getResult ;
endmodule

interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);

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

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

module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);

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

    method ActionValue#(Bit#(32)) getResult ;

endmodule

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

Assume b != 0
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);

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

method ActionValue#(Bit#(32)) getResult ;
    busy_flag <= False; return y;
endmethod
endmodule

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

Assume b != 0
GCD with Guards

module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);
endmodule

Assume b != 0

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

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

method ActionValue#(Bit#(32)) getResult \* if (busy && (x==0));
    busy_flag <= False; return y;
endmethod
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy_flag <- mkReg(False);

rule gcd;
endrule

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

method ActionValue#(Bit#(32)) getResult if (busy && (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

Assume b != 0
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 //subtruct
    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_flag <= True;
endmethod
method ActionValue #(Bit #(32)) getResult if (busy && (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

Assume b != 0
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) overload;
    x <= a; y <= b; busy_flag <= True;
endmethod

method ActionValue #(Bit#(32)) getResult overload;
    busy_flag <= False; return y;
endmethod
endmodule

Assume b != 0
Rule gcd will execute repeatedly until x becomes 0
A module may contain rules

```
rule gcd;
    if (x >= y) begin x <= x - y; end //subtract
else if (x != 0) begin x <= y; y <= x; end //swap
endrule
```
A rule is a collection of actions, which invoke methods.
A module may contain rules

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

- A rule is a collection of actions, which invoke methods
- All actions in a rule execute in parallel
Rule

A module may contain rules

```plaintext
rule gcd;
  if (x >= y) begin x <= x - y; end //subtract
  else if (x != 0) begin x <= y; y <= x; end //swap
endrule
```

What is meaning of this?

- A rule is a collection of actions, which invoke methods
- All actions in a rule execute in parallel
A module may contain rules

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

A rule is a collection of actions, which invoke methods

All actions in a rule execute in parallel
Rule

A module may contain rules

```plaintext
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
```

What is meaning of this? Swap!

- A rule is a collection of actions, which invoke methods
- All actions in a rule execute in parallel
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
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^t; 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

atomicity

What is meaning of this? Swap!
Rules with guards

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

```plaintext
rule foo if (p);
   begin x1 <= e1; x2 <= e2; end
endrule
```
Rules with guards

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

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

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

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

Syntax: In rules, "if" is optional before the guard!
Rules with guards

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

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

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

Syntax: In rules, "if" is optional before the guard!
Rules with guards

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

```plaintext
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
Rules with guards

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

\[
\text{rule foo if } (p); \quad \text{guard} \\
\text{begin } x1 \leftarrow e1; \ x2 \leftarrow e2; \ \text{end} \\
\text{endrule}
\]

Syntax: In rules, “if” is optional before the 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
- An alternative way to write the gcd rule:
Rules with guards

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

```plaintext
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
- An alternative way to write the gcd rule:

```plaintext
rule gcdSubtract if (x >= y);
    x <= x - y;
endrule
```
Rules with guards

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

```plaintext
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:

```plaintext
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
Parallel Composition of Actions & Double-Writes

```plaintext
rule one;
y <= 3; x <= 5; x <= 7; endrule
```

Double write
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
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
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 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 of Actions & Double-Writes

- **rule one:**
  
  \[ y \leq 3; \ x \leq 5; \ x \leq 7; \ \text{endrule} \]

  **Double write**

- **rule two:**
  
  \[ y \leq 3; \ \text{if (b)} \ x \leq 7; \ \text{else} \ x \leq 5; \ \text{endrule} \]

  **No double write**

- **rule three:**
  
  \[ y \leq 3; \ x \leq 5; \ \text{if (b)} \ x \leq 7; \ \text{endrule} \]

  **Possibility of a double write**

- Parallel composition, and consequently a rule containing it, is illegal if a double-write possibility exists
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
Streaming a module

Rules are the glue that connects modules
Streaming a module

Rules are the glue that connects modules

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

Rules are the glue that connects modules

```plaintext
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
```
Streaming a module

Rules are the glue that connects modules

```
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
```

Action value method
Streaming a module

Rules are the glue that connects modules

```
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
```
Streaming a module

Rules are the glue that connects modules

**Rule invokeGCD**

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

**Rule getResult**

```plaintext
rule getResult ;
  let x <- gcd.getResult;
  outQ.enq(x);
endrule
```
Streaming a module

Rules are the glue that connects modules

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

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

Implicit guards?

September 27, 2018
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
```
Streaming a module

Rules are the glue that connects modules

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

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

Implicit guards?

inQ is not empty
gcd is not busy
Bluespec: A new way of expressing behavior using Guarded Atomic Actions

- A module, like an object in OO languages, has a well-defined interface
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”
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
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 \textit{guarded}; it can be applied only if it is “ready”
- The modules are glued together (composed) using \textit{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?

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

versus
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