Hardware Synthesis from Bluespec

Arvind

Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
interface GCD;
    method Action start
        (Bit#(32) a, Bit#(32) b);
    method ActionValue(Bit#(32))
        getResult;
endinterface
Module as a sequential machine

- Inputs and outputs are defined by the type of the module, i.e., its interface definition

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

- Inputs and outputs are defined by the type of the module, i.e., its interface definition
  - A read method has no enable input wire

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

- Inputs and outputs are defined by the type of the module, i.e., its interface definition
  - A read method has no enable input wire
  - An Action method has no output data wires

```verilog
interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue(Bit#(32)) getResult;
endinterface
```
Inputs and outputs are defined by the type of the module, i.e., its interface definition:

- A read method has no enable input wire
- An Action method has no output data wires
- An ActionValue method has ready, enable, output data wires as well as optional input data wires
module mkGCD (GCD);

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

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

method ActionValue#(Bit#(32)) getResult if (busy && (x==0));

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 <- mkReg(False);

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

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

    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));
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 <- mkReg(False);

    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));
endmodule

Assume b != 0

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 <- mkReg(False);

    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

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 <- mkReg(False);

    rule gcd;
    
    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

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 <- 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

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

Assume b != 0
Another GCD implementation
twice the throughput

```
Another GCD implementation
twice the throughput
```
Another GCD implementation
twice the throughput

We can build a GCD module with the same interface but with twice the throughput by putting two gcd modules in parallel.
We can build a GCD module with the same interface but with twice the throughput by putting 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.
Another GCD implementation twice the throughput

- We can build a GCD module with the same interface but with twice the throughput by putting two gcd modules in parallel.
- turnI is used by the start method to direct the input to the gcd whose turn it is and then turnI is flipped.
- Similarly, turnO is used by getResult to get the output from the appropriate gcd, and then turnO is flipped.
interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface
module mkMultiGCD (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 mkMultiGCD (GCD);
    GCD gcd1 <- mkGCD();
    GCD gcd2 <- mkGCD();

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

High-throughput GCD code

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;
endmodule

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
Hardware Synthesis
High-level idea

- Every module represents a sequential machine
Hardware Synthesis
High-level idea

- Every module represents a sequential machine
- Register is a primitive module – its implementation is outside the language
Hardware Synthesis
High-level idea

- Every module represents a sequential machine
- Register is a primitive module – its implementation is outside the language
  - Register bit width is derived from its type
Hardware Synthesis
High-level idea

- Every module represents a sequential machine
- Register is a primitive module – its implementation is outside the language
  - Register bit width is derived from its type
- Each Register and module is instantiated explicitly
  - Input/Output wires of a module are derived from its interface, i.e., type
Hardware Synthesis

High-level idea

- Every module represents a sequential machine
- Register is a primitive module – its implementation is outside the language
  - Register bit width is derived from its type
- Each Register and module is instantiated explicitly
  - Input/Output wires of a module are derived from its interface, i.e., type
- Rules and methods define the combinational logic to connect registers and modules
Hardware Synthesis
High-level idea

- Every module represents a sequential machine
- Register is a primitive module – its implementation is outside the language
  - Register bit width is derived from its type
- Each Register and module is instantiated explicitly
  - Input/Output wires of a module are derived from its interface, i.e., type
- Rules and methods define the combinational logic to connect registers and modules

The resulting hardware is a collection of sequential machines, which itself behaves like a sequential machine
Register: The Primitive module

- Implementation is defined outside the language
- A register is created using `mkReg` or `mkRegU`
- The guards of `_write` and `_read` are always true and not generated
- Special syntax
  - `x <= e` instead of `x._write(e)`
  - `x` instead of `x._read` in expressions

```plaintext
interface Reg#(type t);
  method Action _write(t x);
  method t _read;
endinterface
```
Module to Sequential Machine synthesis

October 2, 2018
Module to Sequential Machine synthesis

- Module M
- Inputs: en, rdy
- Outputs: en, rdy

Diagram:
- reg
- reg
- reg
- reg

October 2, 2018
Module to Sequential Machine synthesis

Declared in the module

October 2, 2018
Module to Sequential Machine synthesis

Clock → reg → reg → reg → Module M → reg

en → rdy

October 2, 2018
Module to Sequential Machine synthesis

Clock → reg → reg → reg → reg → Combinational logic

Module M with inputs en and rdy and outputs en and rdy
Module to Sequential Machine synthesis

Combinational logic

Clock

reg

reg

reg

Module M

en

rdy

en

rdy
Module to Sequential Machine synthesis

October 2, 2018
Module to Sequential Machine synthesis
Module to Sequential Machine synthesis

Defined by the rules and methods of the module

Combinational logic
Module to Sequential Machine synthesis

Combinational logic

Clock

Input

Output

reg

reg

reg

reg

en

rdy

Module M

f

g
Module to Sequential Machine synthesis

Defined by the Module Interface declaration

October 2, 2018
Module to Sequential Machine synthesis

Input

Output

Combinational logic

Clock

reg

reg

reg

reg

Module M

en
rdy

en
rdy
Synchronous Sequential Machines

- In this subject we will study only single-clock circuits, where all the registers are connected to the same clock.
Synchronous Sequential Machines

- In this subject we will study only single-clock circuits, where all the registers are connected to the same clock.
- Bluespec design has no control over this clock and consequently it never shows up in Bluespec codes.
  - We will not show the clock in our circuit diagrams.
Synchronous Sequential Machines

- In this subject we will study only single-clock circuits, where all the registers are connected to the same clock.
- Bluespec design has no control over this clock and consequently it never shows up in Bluespec codes.
  - We will not show the clock in our circuit diagrams
- Bluespec design does specify the enable signal for each register implicitly and the Bluespec compiler generates the enable signal and the associated data for each register.
Rules and methods define the combinational logic

```vhdl
module mkEx1 (...);
    Reg#(t) x <= mkRegU;
method Action f(t a);
    x <= e;
endmethod
endmodule
```
Rules and methods define the combinational logic

module mkEx1 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e;
    endmethod
endmodule
Rules and methods define the combinational logic

```verilog
module mkEx1 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e;
    endmethod
endmodule
```
Rules and methods define the combinational logic

module mkEx1 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e;
    endmethod
endmodule

October 2, 2018
Rules and methods define the combinational logic

module mkEx1 (...);
    Reg#(t) x <= mkRegU;
    method Action f(t a);
        x <= e;
    endmethod
endmodule

module mkEx2 (...);
    Reg#(t) x <= mkRegU;
    method Action f(t a);
        if (b) x <= e;
    endmethod
endmodule
Rules and methods define the combinational logic

module mkEx1 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    x <= e;
  endmethod
endmodule

module mkEx2 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    if (b) x <= e;
  endmethod
endmodule
Rules and methods define the combinational logic

module mkEx1 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e;
    endmethod
endmodule

module mkEx2 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        if (b) x <= e;
    endmethod
endmodule

October 2, 2018
Rules and methods define the combinational logic

module mkEx1 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    x <= e;
  endmethod
endmodule

module mkEx2 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    if (b) x <= e;
  endmethod
endmodule

module mkEx3 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    if (b) x <= e1;
    else x <= e2;
  endmethod
endmodule
Rules and methods define the combinational logic

```
module mkEx1 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e;
    endmethod
endmodule

module mkEx2 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        if (b) x <= e;
    endmethod
endmodule

module mkEx3 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        if (b) x <= e1;
        else x <= e2;
    endmethod
endmodule
```
Rules and methods define the combinational logic

module mkEx1 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e;
    endmethod
endmodule

module mkEx2 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        if (b) x <= e;
    endmethod
endmodule

module mkEx3 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        if (b) x <= e1;
        else x <= e2;
    endmethod
endmodule
Dealing with multiple sources for register assignments

module mkEx4 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e1;
    endmethod
    method Action g(t a);
        x <= e2;
    endmethod
endmodule
Dealing with multiple sources for register assignments

module mkEx4 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    x <= e1;
  endmethod
  method Action g(t a);
    x <= e2;
  endmethod
endmodule
Dealing with multiple sources for register assignments

module mkEx4 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    x <= e1;
  endmethod
  method Action g(t a);
    x <= e2;
  endmethod
endmodule
Dealing with multiple sources for register assignments

module mkEx4 (...);
    Reg#(t) x <= mkRegU;
    method Action f(t a);
        x <= e1;
    endmethod
    method Action g(t a);
        x <= e2;
    endmethod
endmodule

f.data -> x.data
f.en -> x.en

g.data -> x.data
g.en -> x.en
Dealing with multiple sources for register assignments

module mkEx4 (...);
    Reg#(t) x <= mkRegU;
    method Action f(t a);
        x <= e1;
    endmethod
    method Action g(t a);
        x <= e2;
    endmethod
endmodule
Dealing with multiple sources for register assignments

Both f and g can’t assign to x at the same time

module mkEx4 (...);
  Reg#(t) x <= mkRegU;
  method Action f(t a);
    x <= e1;
  endmethod
  method Action g(t a);
    x <= e2;
  endmethod
endmodule
Dealing with multiple sources for register assignments

Both f and g can’t assign to x at the same time

Compiler has to ensure in such a setting that both f and g can’t be enabled together
A new mux-like structure to deal with multiple sources

\[ x = (v1 \& x1) \mid (v2 \& x2) \]

\[ v = v1 \mid v2 \]
A new mux-like structure to deal with multiple sources

\[ x = (v_1 \& x_1) \mid (v_2 \& x_2) \]
\[ v = v_1 \mid v_2 \]

- \( x_i \) has a meaningful value only if its corresponding \( v_i \) is true
A new mux-like structure to deal with multiple sources

\[ x = (v_1 \land x_1) \lor (v_2 \land x_2) \]
\[ v = v_1 \lor v_2 \]

- \( x_i \) has a meaningful value only if its corresponding \( v_i \) is true
- Compiler has to ensure that at most one \( v_i \) is true at any given time; the circuit will behave unpredictably if multiple input signals are valid
A new mux-like structure to deal with multiple sources

$x_i$ has a meaningful value only if its corresponding $v_i$ is true

Compiler has to ensure that at most one $v_i$ is true at any given time; the circuit will behave unpredictably if multiple input signals are valid

$x = (v_1 \& x_1) \mid (v_2 \& x_2)$

$v = v_1 \mid v_2$
A new mux-like structure to deal with multiple sources

- $x_i$ has a meaningful value only if its corresponding $v_i$ is true
- Compiler has to ensure that at most one $v_i$ is true at any given time; the circuit will behave unpredictably if multiple input signals are valid

$x = (v_1 \& x_1) \mid (v_2 \& x_2)$
$v = v_1 \mid v_2$
A new mux-like structure to deal with multiple sources

- $x_i$ has a meaningful value only if its corresponding $v_i$ is true.
- Compiler has to ensure that at most one $v_i$ is true at any given time; the circuit will behave unpredictably if multiple input signals are valid.

$x = (v_1 \& x_1) \mid (v_2 \& x_2)$

$v = v_1 \mid v_2$
Synthesis of multiple sources for register assignments

module mkEx4 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e1;
    endmethod
    method Action g(t a);
        x <= e2;
    endmethod
endmodule
module mkEx4 (...);
    Reg#(t) x <= mkRegU;
    method Action f(t a);
        x <= e1;
    endmethod
    method Action g(t a);
        x <= e2;
    endmethod
endmodule
Synthesis of multiple sources for register assignments

module mkEx4 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    x <= e1;
  endmethod
  method Action g(t a);
    x <= e2;
  endmethod
endmodule
Synthesis of multiple sources for register assignments

- Compiler has to ensure that both f and g can’t be enabled together

```verilog
module mkEx4 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
    x <= e1;
  endmethod
  method Action g(t a);
    x <= e2;
  endmethod
endmodule
```
Example

One-Element FIFO Implementation

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
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
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

- I/O: Interface
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

I/O: Interface
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

- I/O: Interface
- Instantiate state
FIFO Circuit

```verilog
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
```

- I/O: Interface
- Instantiate state
FIFO Circuit

```verilog
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
```

- I/O: Interface
- Instantiate state
- Insert muxes
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

- I/O: Interface
- Instantiate state
- Insert muxes
### FIFO Circuit

```verilog
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
```

- **I/O: Interface**
- Instantiate state
- Insert muxes
- Compile enq

---

October 2, 2018
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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
FIFO Circuit

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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
FIFO Circuit

```verilog
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
```

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
FIFO Circuit

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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
FIFO Circuit

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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
- Compile deq
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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
- Compile deq
FIFO Circuit

```
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
```

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
- Compile deq
FIFO Circuit

```verilog
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
```

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
- Compile deq
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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
- Compile deq
- Compile first
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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
- Compile deq
- Compile first
FIFO Circuit

```
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
```

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
- Compile deq
- Compile first
Redrawing the FIFO Circuit

enq.data → first.data

enq.en

deq.en

v

True → enq.rdy
False → deq.rdy

first.rdy
Redrawing the FIFO Circuit

- enq.data
- enq.en
- deq.en
- True
- False
- first.data
- enq.rdy
- deq.rdy
- first.rdy
Redrawing the FIFO Circuit

A module is a sequential circuit with input and output wires corresponding to its interface methods.
## Next state transition

### Partial Truth Table

<table>
<thead>
<tr>
<th>enq. en</th>
<th>enq. data</th>
<th>deq. en</th>
<th>$d^t$</th>
<th>$v^t$</th>
<th>$d^{t+1}$</th>
<th>$v^{t+1}$</th>
<th>enq. rdy</th>
<th>deq. rdy</th>
<th>first. rdy</th>
<th>first. data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>d</td>
<td>1</td>
<td>d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>0</td>
<td>x</td>
<td>0</td>
<td>d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>1</td>
<td>d</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

- **inputs**: enq. en, enq. data, deq. en
- **state**: $d^t$, $v^t$
- **next state**: $d^{t+1}$, $v^{t+1}$
- **outputs**: enq. rdy, deq. rdy, first. rdy, first. data

---

October 2, 2018
## Next state transition

### Partial Truth Table

<table>
<thead>
<tr>
<th>Inputs</th>
<th>State</th>
<th>Next State</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$d^t$</td>
<td>$v^t$</td>
<td>$d^{t+1}$</td>
</tr>
<tr>
<td><strong>0</strong></td>
<td><strong>x</strong></td>
<td><strong>0</strong></td>
<td><strong>x</strong></td>
</tr>
<tr>
<td><strong>0</strong></td>
<td><strong>x</strong></td>
<td><strong>0</strong></td>
<td><strong>d</strong></td>
</tr>
<tr>
<td><strong>1</strong></td>
<td><strong>d</strong></td>
<td><strong>0</strong></td>
<td><strong>x</strong></td>
</tr>
<tr>
<td><strong>0</strong></td>
<td><strong>x</strong></td>
<td><strong>1</strong></td>
<td><strong>d</strong></td>
</tr>
</tbody>
</table>

### Illegal inputs

- 1
- 1
- 1
- 1
## Next state transition

Partial Truth Table

<table>
<thead>
<tr>
<th>inputs</th>
<th>state</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq.</td>
<td>enq.</td>
<td>deq.</td>
<td>d^t</td>
</tr>
<tr>
<td>en</td>
<td>data</td>
<td>enq.</td>
<td>enq.</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

**Illegal inputs**

| 1     | 0     | 1     | 0     | 0 |
| 1     | 0     | 1     | 1     | 1 |
| 1     | 1     | 1     | 1     | 1 | Tedious!
Constraints on the use of methods of a FIFO

- Bluespec compiler must ensure:
  - enq.en is not set to True unless enq.rdy is True
  - Similarly, deq.en is not set unless deq.rdy is True
Constraints on the use of methods of a FIFO

- Bluespec compiler must ensure:
  - enq.en is not set to True unless enq.rdy is True
  - Similarly, deq.en is not set unless deq.rdy is True

Let us illustrate this by putting the FIFO to use.
Streaming a function: Circuit

```plaintext
rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
```
Streaming a function: Circuit

Rule stream:
outQ.enq(f(inQ.first));
inQ.deq;
endrule
Streaming a function: Circuit

```
rule stream;
  outQ.enq(f(inQ.first));
inQ.deq;
endrule
```
Streaming a function: Circuit

rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
Stream a function: Circuit

```verilog
rule stream;  
  outQ.enq(f(inQ.first));  
  inQ.deq;  
endrule
```

- `inQ` and `outQ` are queues.
- `f` is a function.
- `enq`, `deq`, and `first` are operations on queues.
Streaming a function: Circuit

```
rule stream;
  outQ.enq(f(inQ.first));
inQ.deq;
endrule
```
Streaming a function: Circuit

```
rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
```
Streaming a function: Circuit

```plaintext
rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
```
Streaming a function: Circuit

```
rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
```
Streaming a function: Circuit

```
rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
```
Streaming a function: Circuit

rule stream;
    outQ.enq(f(inQ.first));
    inQ.deq;
endrule
Streaming a function: Circuit

```
rule stream;
  outQ.enq(f(inQ.first));
inQ.deq;
endrule
```

This is a sequential machine too!
Streaming a function: Circuit

This is a sequential machine too!

Notice that enq.en cannot be True unless enq.rdy is true; deq.en cannot be True unless deq.rdy is true

rule stream;
  outQ.enq(f(inQ.first));
inQ.deq;
endrule

October 2, 2018

MIT 6.004 Fall 2018
Redrawing the sequential circuit

Input wires:
- enq
- inQ
- first
- deq

Output wires:
- enq
- outQ
- first
- deq

f

October 2, 2018
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine

Register inputs and outputs are replaced by method inputs and outputs
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine

Combinational logic
(no cycles, no clock)

Register inputs and outputs are replaced by method inputs and outputs
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine

Combinational logic
(no cycles, no clock)

Register inputs and outputs are replaced by method inputs and outputs
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine

Combinational logic
(no cycles, no clock)

Register inputs and outputs are replaced by method inputs and outputs
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine

Combinational logic
(no cycles, no clock)

Register inputs and outputs are replaced by method inputs and outputs
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine

Register inputs and outputs are replaced by method inputs and outputs
Take-home problem: Draw the circuit for this GCD

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

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

Assume b != 0