Lecture 2

Bluespec System Verilog (BSV):
A language for hardware design

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Oregon Programming Language Summer School (OPLSS)
Eugene, OR
July 16, 2018

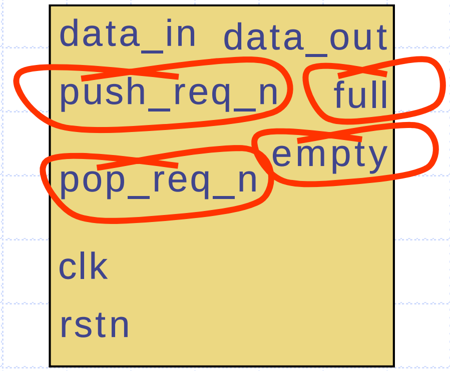
What is needed to bring hardware design to 21st Century

- Extreme IP reuse
  - Multiple instantiations of a block for different performance and application requirements
  - Packaging of IP so that the blocks can be assembled easily to build a large system (black box model)
- Ability to do modular refinement
- Whole system simulation to enable concurrent hardware-software development

"Intellectual Property"

IP reuse sounds wonderful until you try it ...

Example: Commercially available FIFO IP block



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

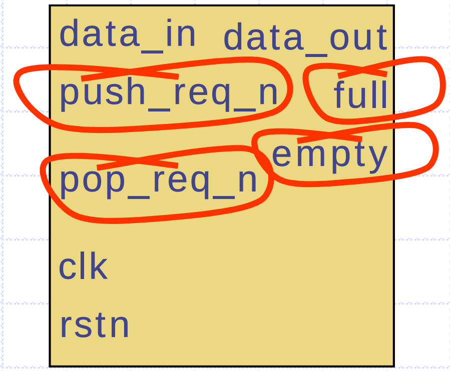
Thus, there is no conflict in a simultaneous push and pop when the FIFO is full. A simultaneous push and pop cannot occur when the FIFO is empty, since there is no pop data to prefetch. However, push data is captured in the FIFO.

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`.

These constraints are spread over many pages of the documentation...

IP reuse sounds wonderful until you try it ...

Example: Commercially available FIFO IP block



An error occurs if a push is attempted while the FIF... l.

Thus, there is no conflict in a simult... when the FIFO is full. A simultaneous push and pop c... FIFO is empty, since there is no pop data to prefetch. How... red in the FIFO.

A pop o... pop_req_n is asserted (LOW), as long as the FIFO is not em... eq_n causes the internal read pointer to be incremented on the nex... of clk. Thus, the RAM read data must be captured on the clk following the a... on of pop_req_n.

No machine verification of such informal constraints is feasible

These constraints are spread over many pages of the documentation...

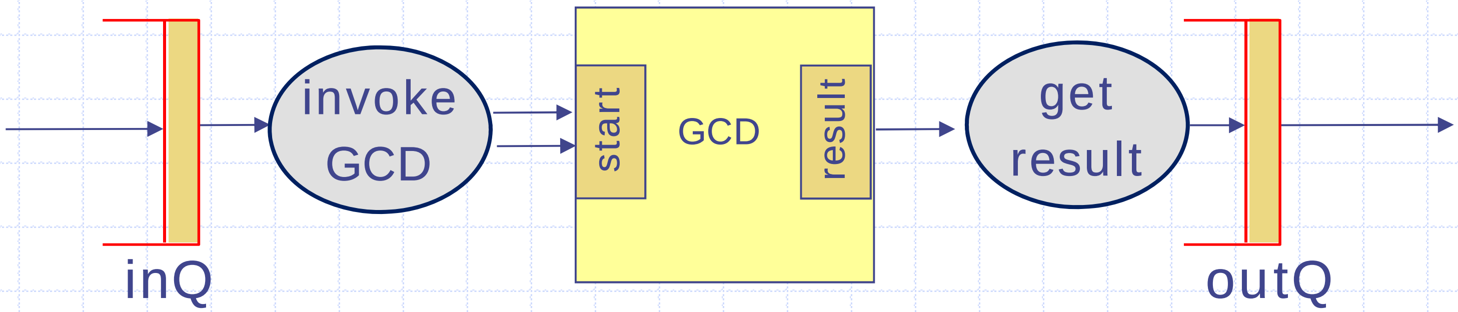
Bluespec can change all this

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

An example ...

A system that calls the GCD module repeatedly




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

```
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
```

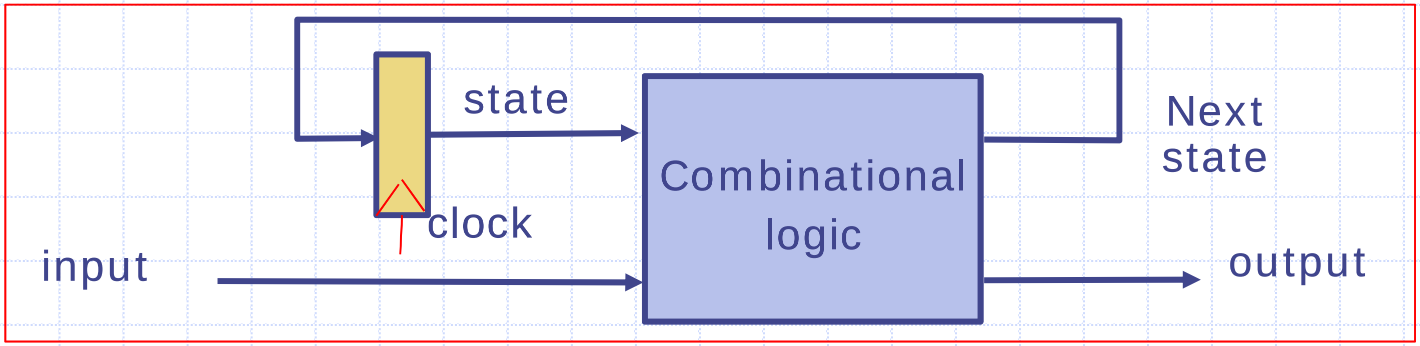
Plan

- Use the GCD example to illustrate
  - Guarded interfaces
  - Guarded atomic rules
  - Hardware generation
  - High-performance GCD

but first a tutorial on digital circuits

Finite State Machines (FSM) and Sequential Circuits

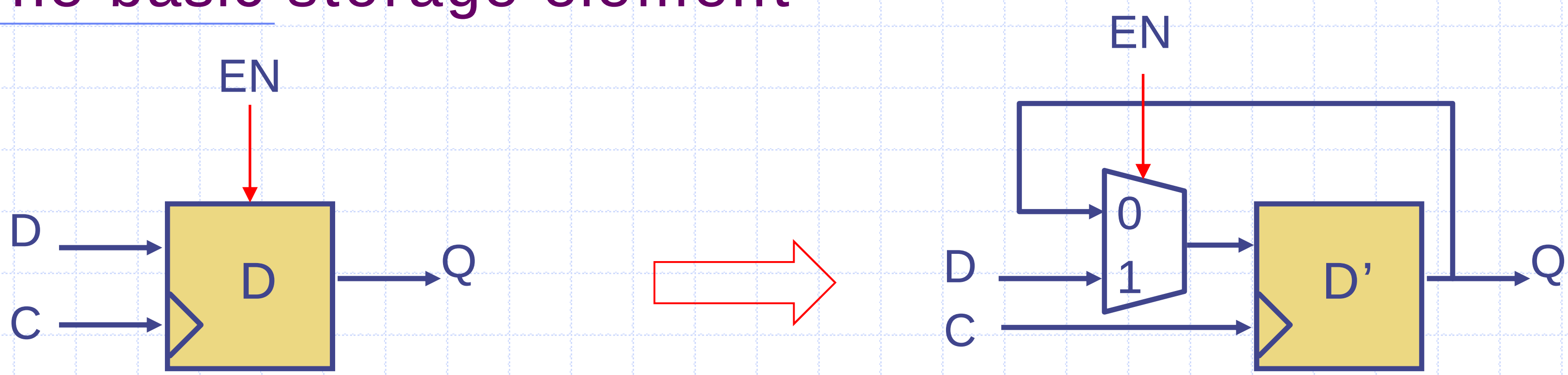
- FSMs are a mathematical object like the Boolean Algebra
  - A computer (in fact any digital hardware) is an FSM
- Synchronous Sequential Circuits is a method to implement FSMs in hardware



- Large circuits need to be described as a *collection of cooperating FSMs*

D Flip-flop with Write Enable

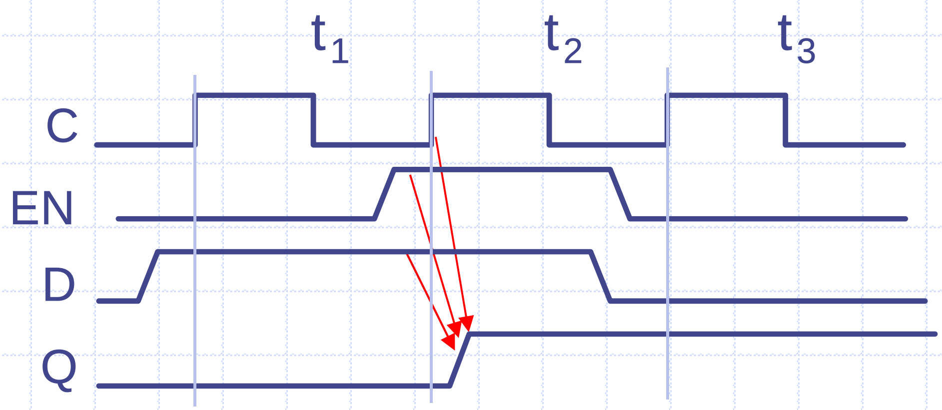
The basic storage element



EN	D	Q^t	Q^{t+1}
0	X	0	0
0	X	1	1
1	0	X	0
1	1	X	1

hold (EN = 0 rows)

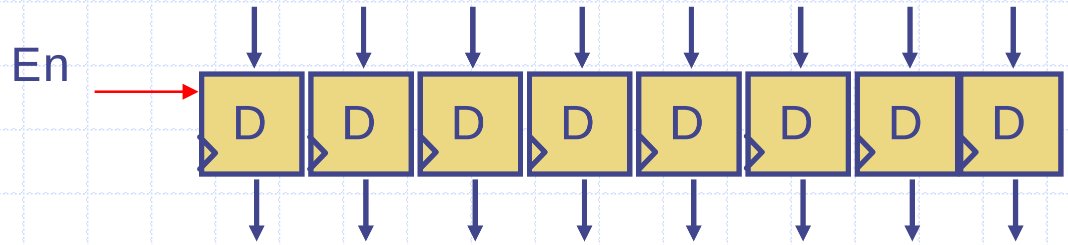
copy input (EN = 1 rows)



No need to show the clock explicitly

Data is captured only if EN is on

Registers



Register: A group of flip-flops with a common enable

Register file: A group of registers with a shared set of input and output ports

Clocked Sequential Circuits

- Any sequential circuit can be built using D flip-flops (with write-enable)
  - The state of the flip flop can change only when the write enable is on
  - The change of state can only be seen a clock later
- In a circuit with a single-clock domain all flip flops are connected to the same clock
  - To avoid clutter, the clock input is not shown
- Clock inputs are not needed in BSV descriptions unless we design multi-clock circuits

A module in BSV describes a sequential circuit

- A module has internal state
- The internal state can only be read and manipulated by the (interface) methods
- An *action method* 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)

Let us design a GCD module

GCD algorithm

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

15	6	
9	6	*subtract*
3	6	*subtract*
6	3	*swap*
3	3	*subtract*
0	3	*subtract*

answer: 3

```
def gcd(a, b):
    if a == 0: return b  # stop
    elif a >= b: return gcd(a-b,b) # subtract
    else: return gcd (b,a) # swap
```

GCD

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

  rule gcd;




  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


endmodule
```

start should be called only if the module is not busy

Assume b /= 0

GCD

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

  rule gcd;




  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
```

getResult can be called only when the result is ready is true

Assume b /= 0

GCD

```
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
```

gcd will execute repeatedly until x becomes 0

Assume b /= 0

Rule

A module may contain rules

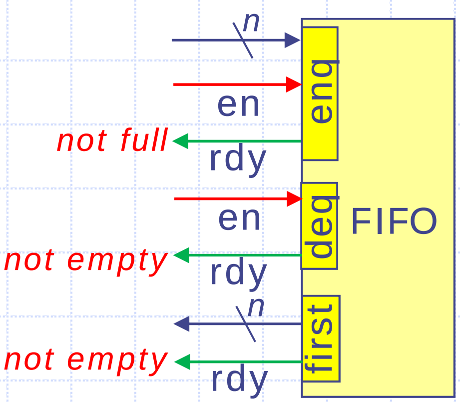
```
rule gcd;
   if (x >= y) begin x <= x – y; end //subtract
   else if (x != 0) begin x <= y; y <= x; end //swap
endrule
```

- A rule is a collection of actions, which invoke methods
- 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

Guarded interfaces

- User convenience: Include some checks (readyness, fullness, ...) in the method definition itself to avoid having 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

Rules with guards

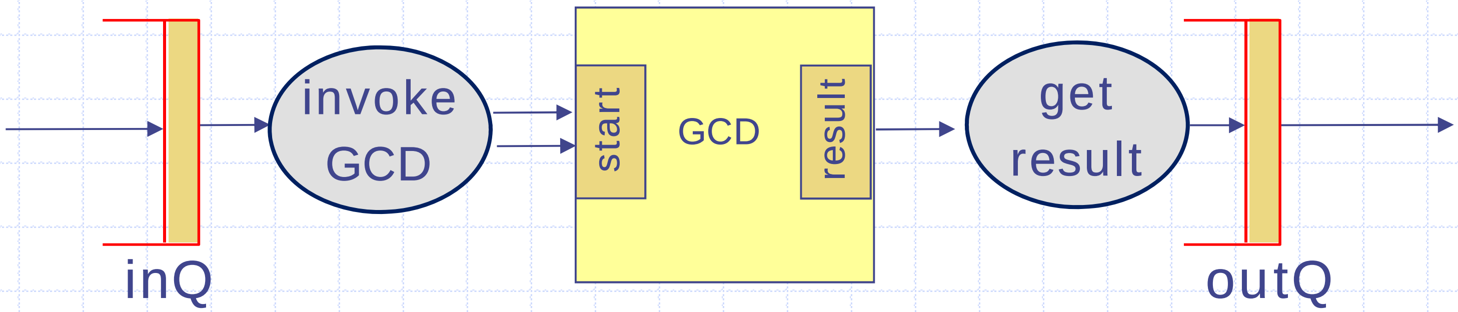
- Like a method, a rule can have an *explicit and implicit guard* (true guards can be omitted)

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

explicit guard

- A rule can execute only if all of it's explicit and implicit guards are true, i.e., if any guard is false the rule has no effect

Streaming the GCD




```
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

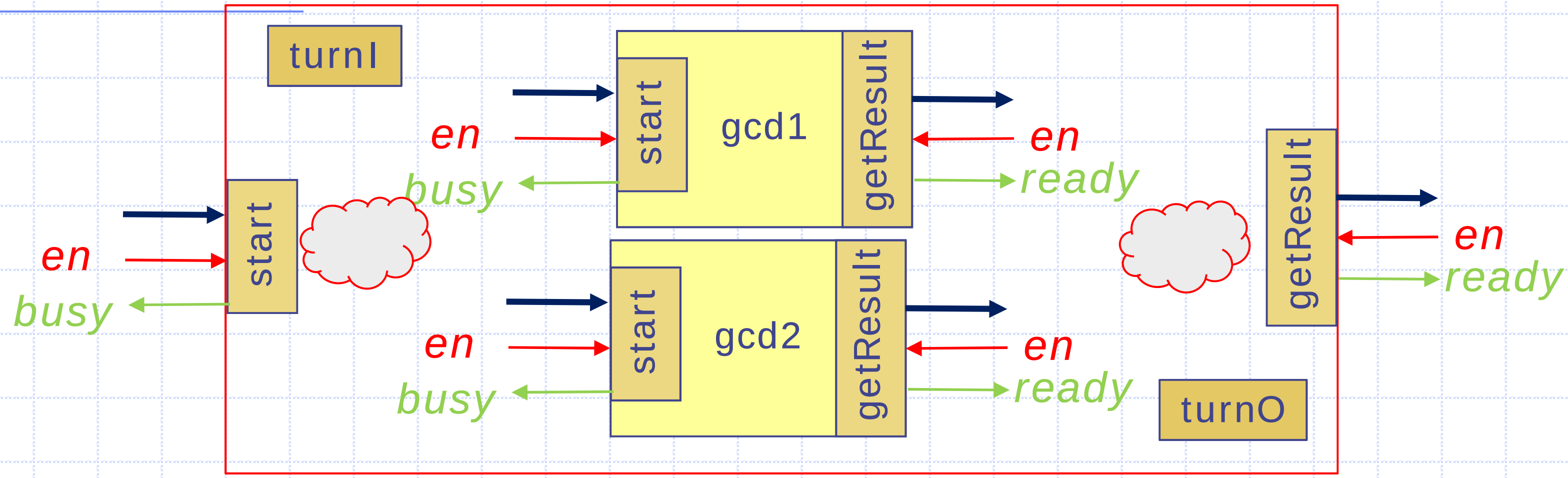
explicit guard?

implicit guards?

Latency-Insensitive interface

- Notice, GCD interface is latency-insensitive; no assertion can be made about how many cycles later the result would be ready
- The interface also does not tell us if GCD is pipelined or not
  - Our implementation is not pipelined
- The interface also does not tell us if the results come out in-order
  - If the results can come out of order, the user should tag the inputs and outputs
- This latency-insensitivity allows us to refine the GCD module as we see fit.

GCD with 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
- A variable turnI can be used by `start` to direct the input to the gcd whose turn it is. Then flip it
- Similalry getResult can use turnO to pull the result from the appropriate gcd

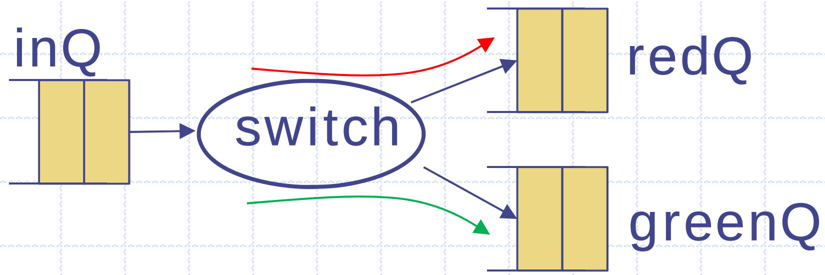
High throughput GCD code

```
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
```

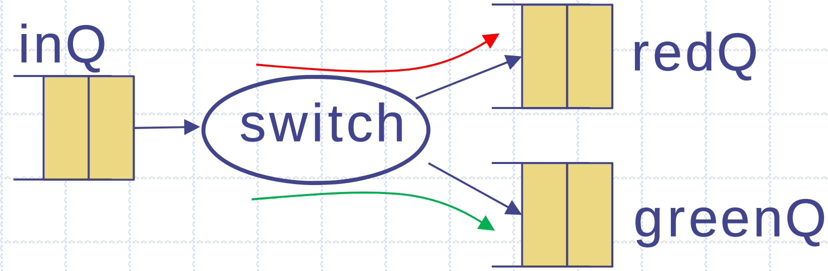
Switch using FIFOs with guarded interfaces




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

What is the implicit guard?

Switch using FIFOs with guarded interfaces



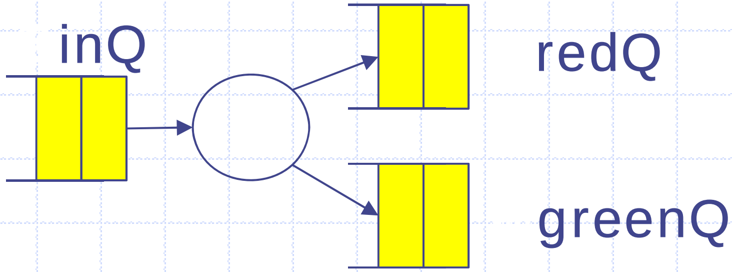

```
rule switch;
  if (inQ.first.color == Red) begin
    redQ.enq(inQ.first.value); inQ.deq;
  end else begin // color is Green
    greenQ.enq(inQ.first.value); inQ.deq;
  end
endrule
```

What is the implicit guard?

```
inQ.notEmpty ?
    ((inQ.first.color == Red) ?
            redQ.notFull : greenQ.notFull)
    : False
```

Guards are convenient!

Mutually Exclusive rules



Switch can be split into two mutually exclusive rules

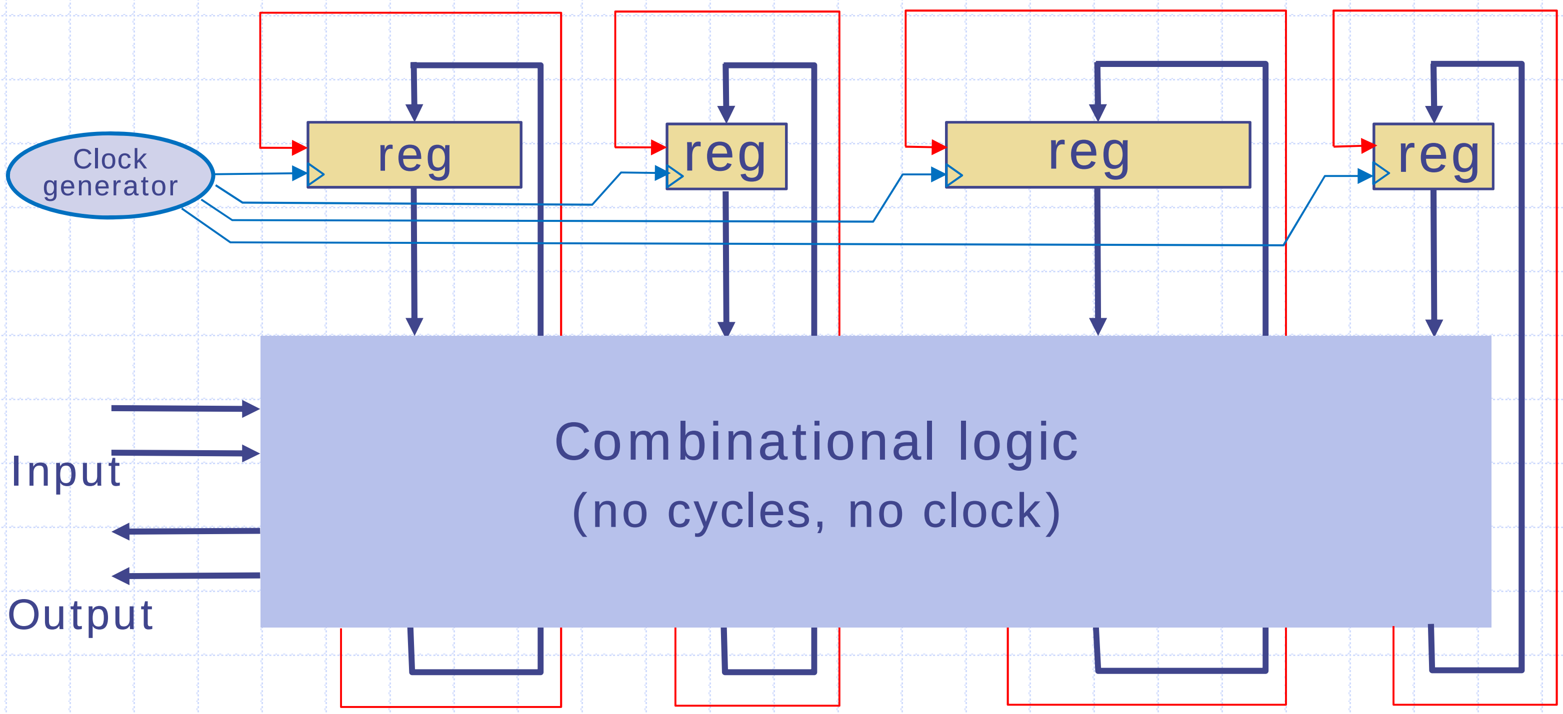
```
rule switchRed if (inQ.first.color == Red);
  redQ.enq(inQ.first.value); inQ.deq;
endrule;

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

Only one of the rules can be active in a given state

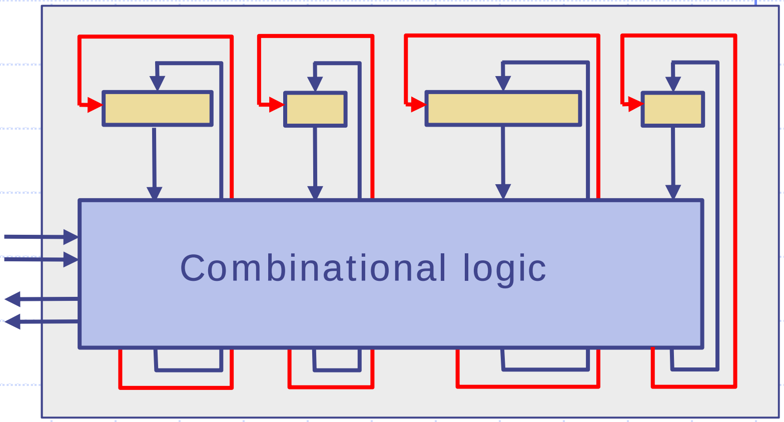
Hardware Synthesis from BSV

Synchronous Sequential Machines



BSV to Sequential Circuits

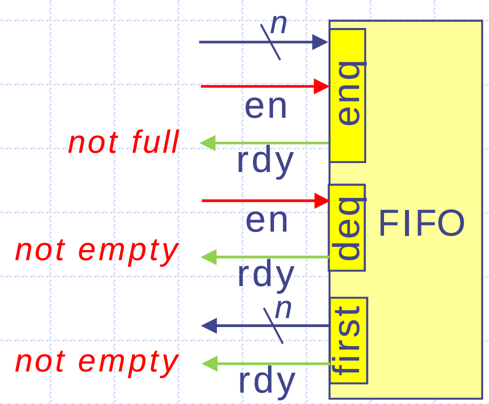
- Each Register and its width is declared explicitly
- All registers are driven by a common clock which is implicit; your program has no control over it



- Combinational logic is derived from the rules and methods you write
- Your program defines the input value and the enable for each register
- Each rule, action method, and action value method generates an enable signal for each register it sets directly or indirectly

One-Element FIFO Implementation with guards

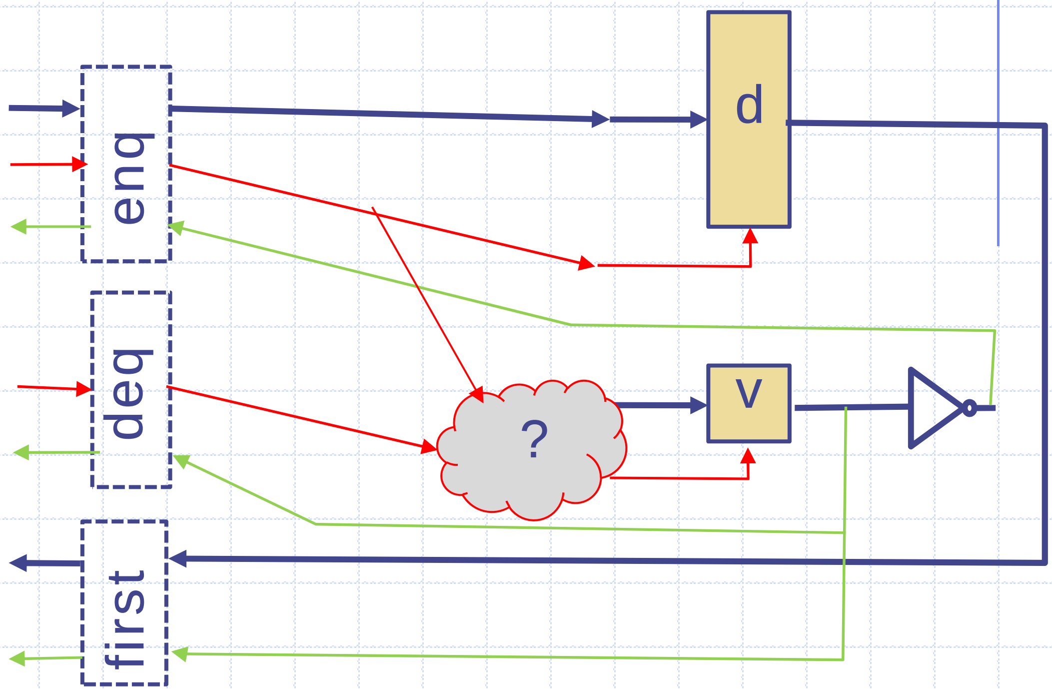
```
module mkFifo (Fifo#(1, t));
  Reg#(t)    d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(t x)  if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq  if (v);
    v <= False;
  endmethod
  method t first if (v);
    return d;
  endmethod
endmodule
```




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

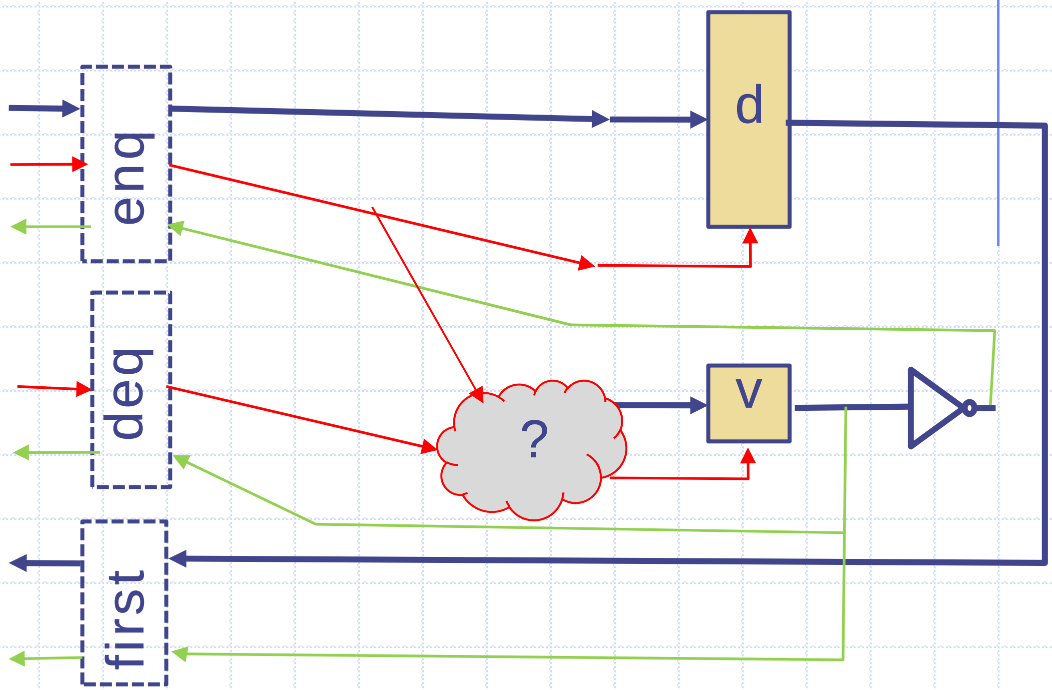
FIFO Circuit

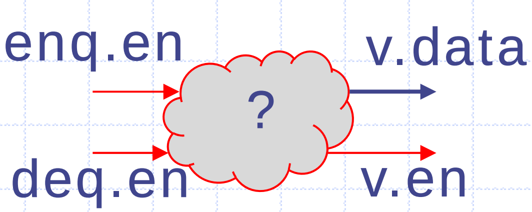
```
module mkFifo (Fifo#(1, t));
  Reg#(t)    d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(t x) if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method t first if (v);
    return d;
  endmethod
endmodule
```

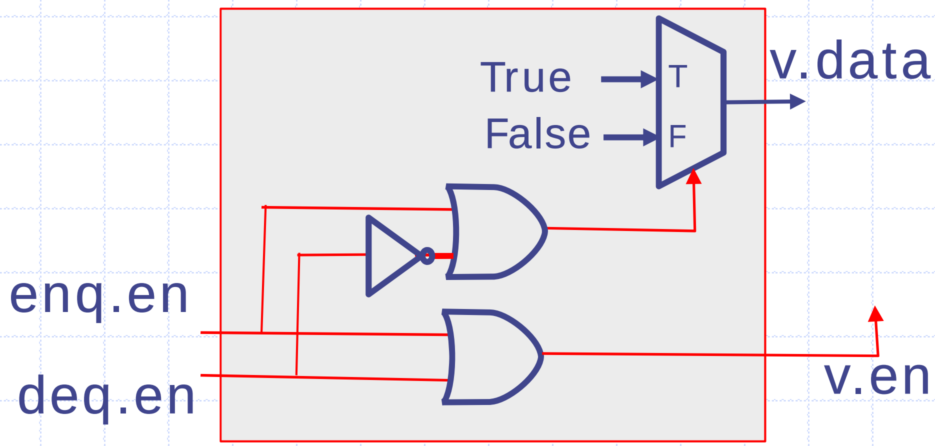


FIFO Circuit

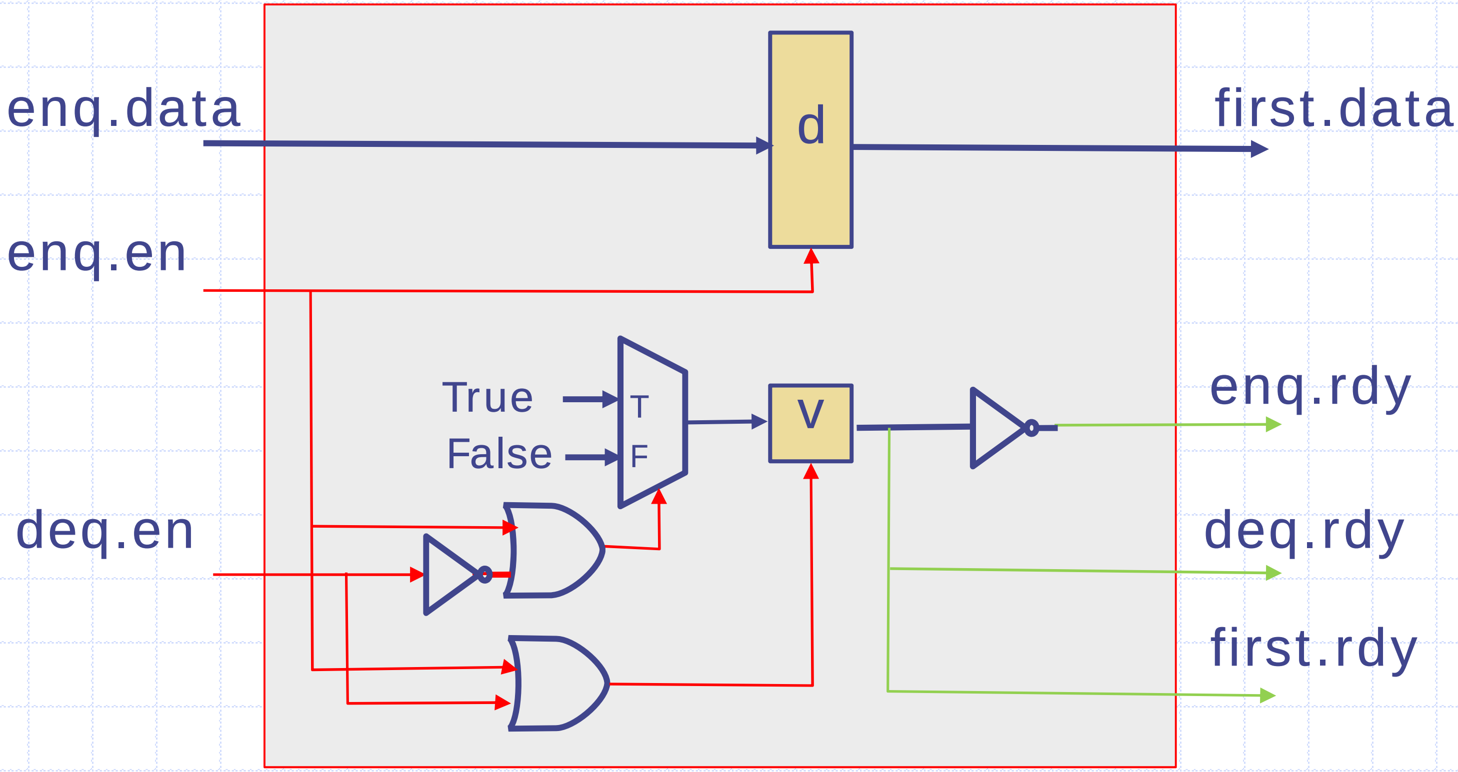
```
module mkFifo (Fifo#(1, t));
  Reg#(t)    d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(t x) if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq if (v);
    v <= False;
  endmethod
  method t first if (v);
    return d;
  endmethod
endmodule
```







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

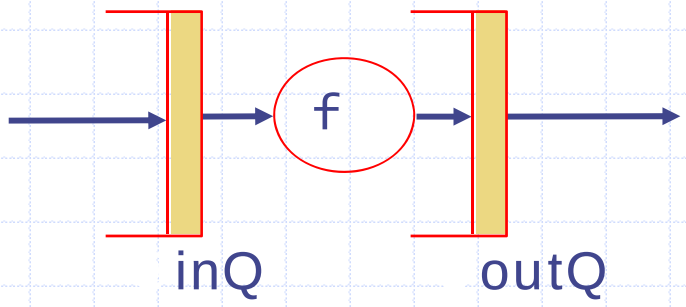
	inputs			state		next state		outputs			
	enq. en	enq. data	deq. en	d^t	v^t	d^{t+1}	v^{t+1}	enq. rdy	deq. rdy	first. rdy	first. data
	0	x	0	0	0	0	0	1	0	0	0(?)
	0	x	0	0	1	0	1	0	1	1	0
	0	x	0	1	0	1	0	1	0	0	1(?)
	0	x	0	1	1	1	1	0	1	1	1
	1	0	0	x	0	0	1	1	0	0	?
	1	1	0	x	0	1	1	1	0	0	?
Illegal inputs	1				1			0			
			1		0				0		
	1		1		0			1			
	1		1		1				1		

Constraints on the use of methods of a FIFO

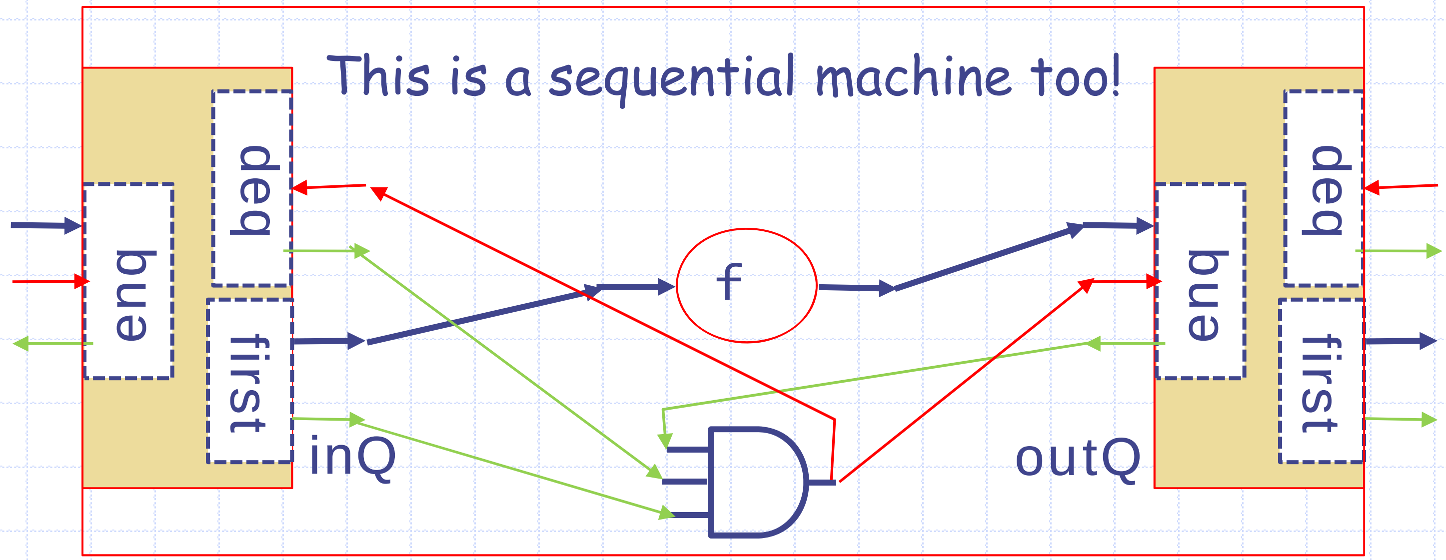
- The BSV compiler makes sure that the enq.en is not set to True unless enq.rdy is True
- Similarly, for deq.en and deq.rdy
- Your code is such that enq.rdy and deq.rdy also cannot be True simultaneously. Thus, the input for the v register is always well defined

more on this topic in the next lecture

Streaming a function: Circuit



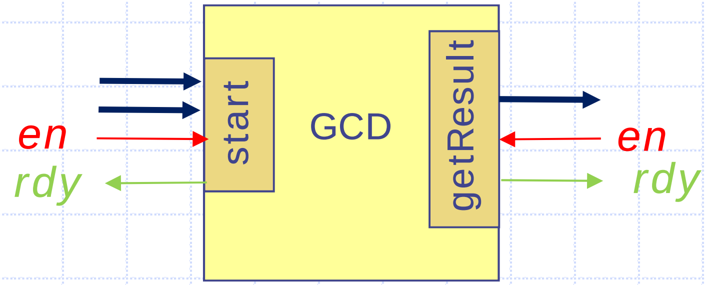

```
rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
```



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

Module as a sequential circuit

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



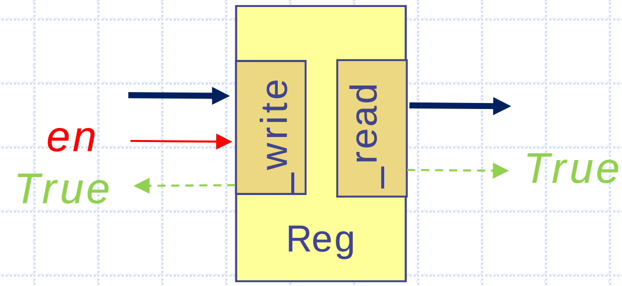
- In general:
  - A *read method* has no enable input wire
  - An *Action method* has no output data wires
  - An *ActionValue method* has both ready and enable wires as well as both input and output data wires

We can determine all the input and output wires of a module from its interface definition

Register as a primitive module

- A register is a primitive module in BSV and its implementation is defined outside the language

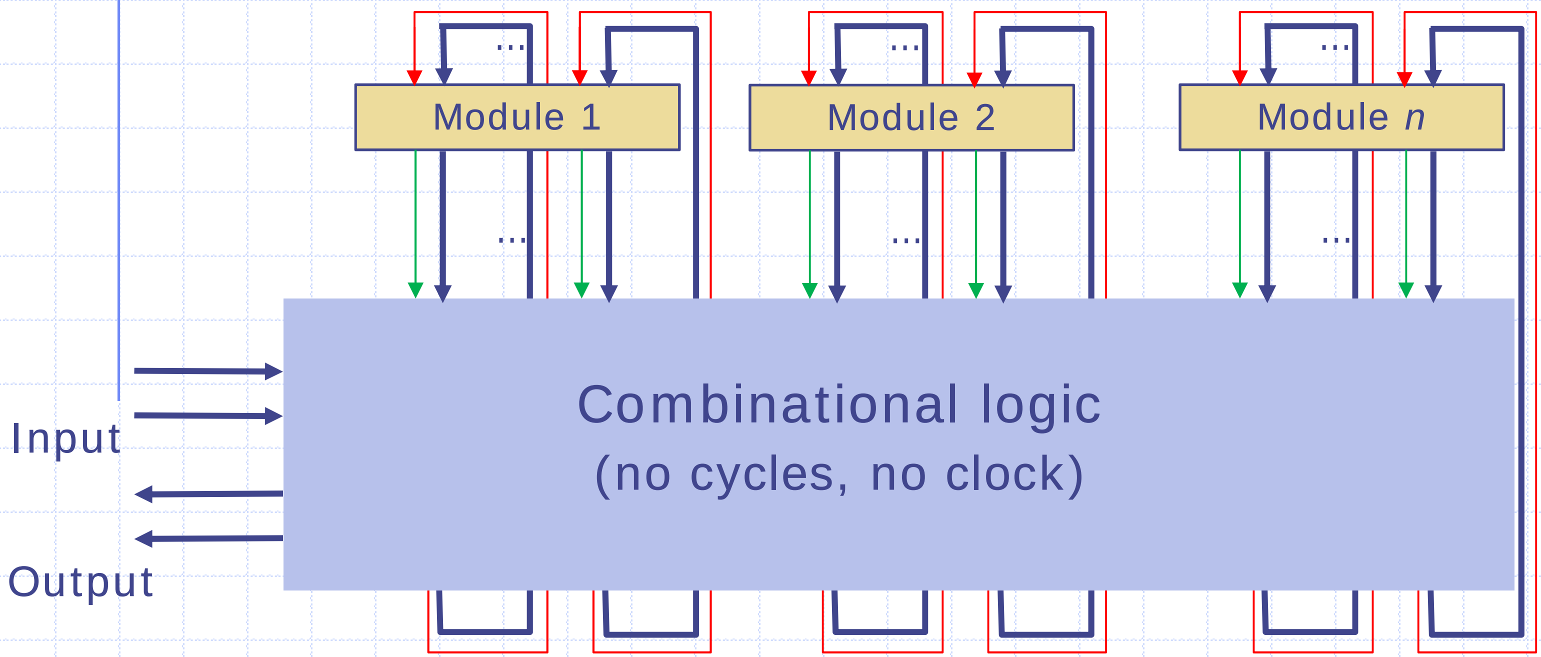
```
interface Reg#(type t);
   method Action _write(t x);
   method t _read;
endinterface
```



- Special syntax: we write
  - x <= e instead of x._write(e)
  - x instead on x._read in expressions
- The guards of _write and _read are always true
  - The guard wires are not generated for registers

Hierarchical sequential circuits

sequential circuits containing modules



Each module represents a sequential machine

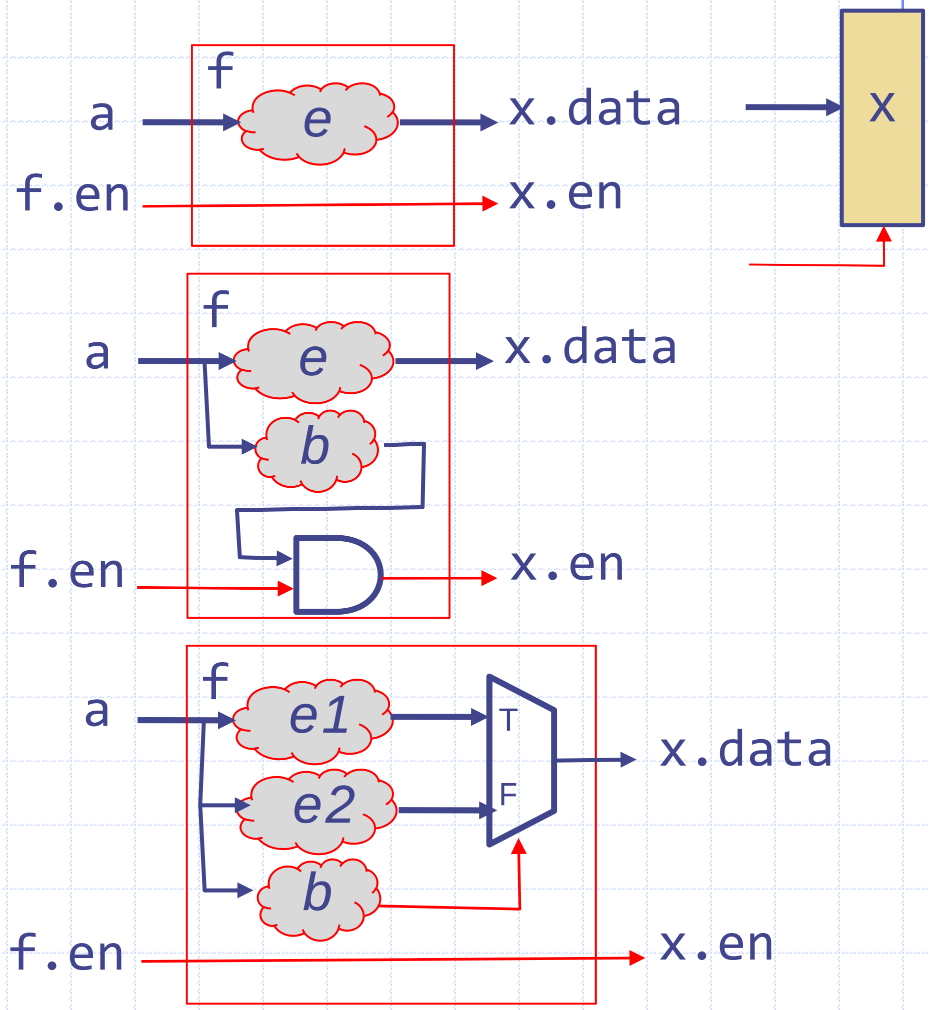
Register inputs and outputs are replaced by method inputs and outputs

Rules and methods *only* define combinational logic

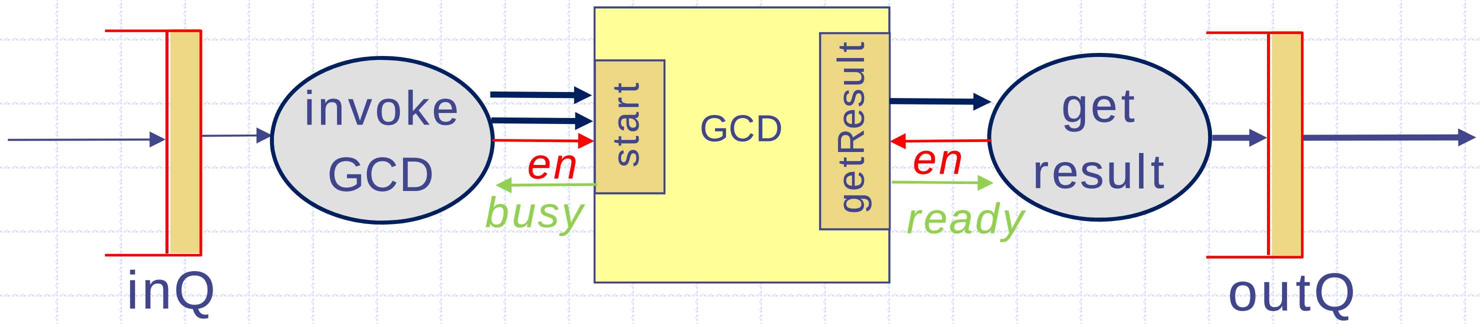
```
module mkEx1 (...);
  Reg#(t) x <- mkRegU;
  method Action f(t a);
     x <= e;
endmethod endmodule
```

```
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
```



Streaming the GCD module

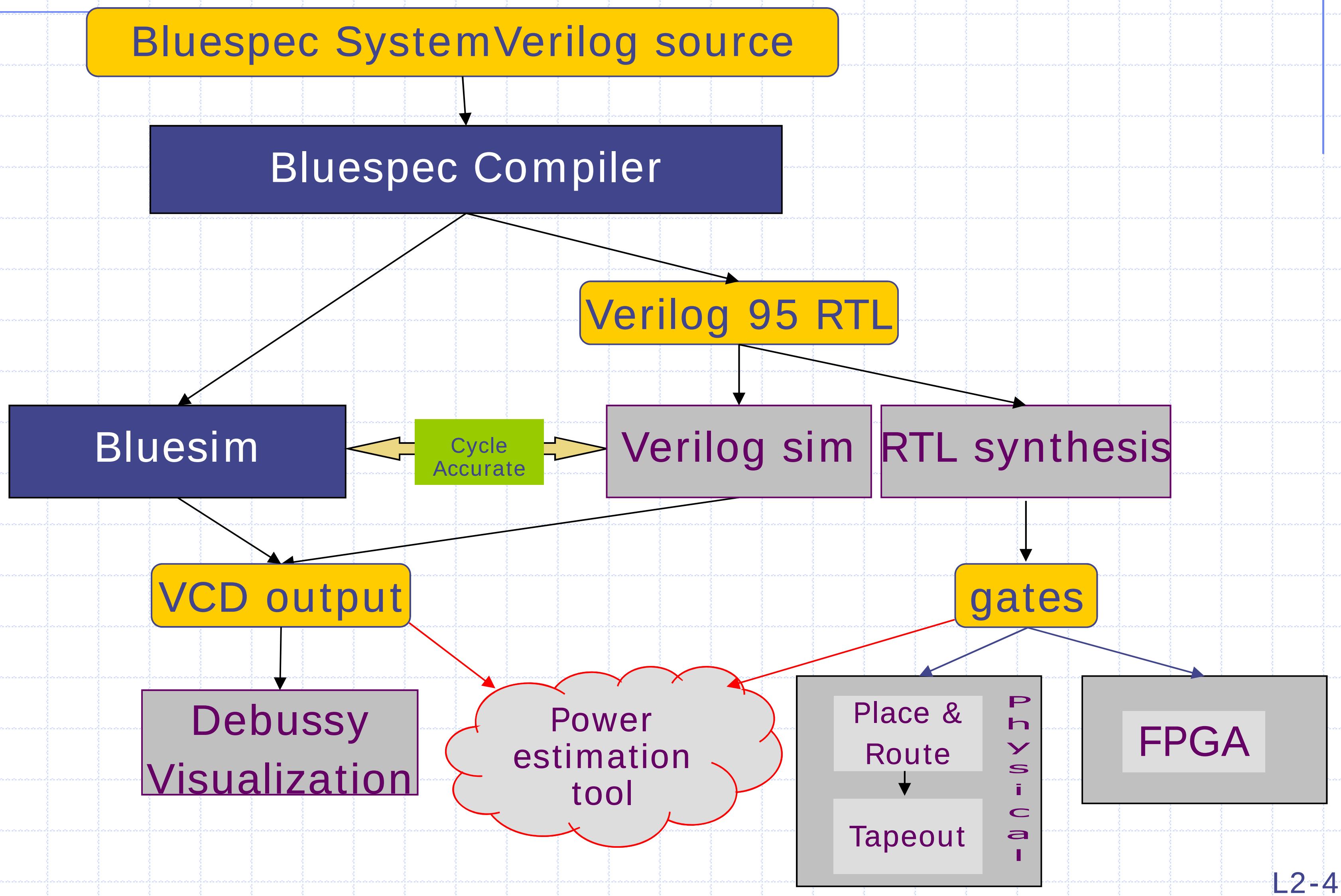



```
rule invokeGCD;
   let x = tpl_1(inQ.first);
   let y = tpl_2(inQ.first);
     gcd.start(x,y);
     inQ.deq;
endrule
```

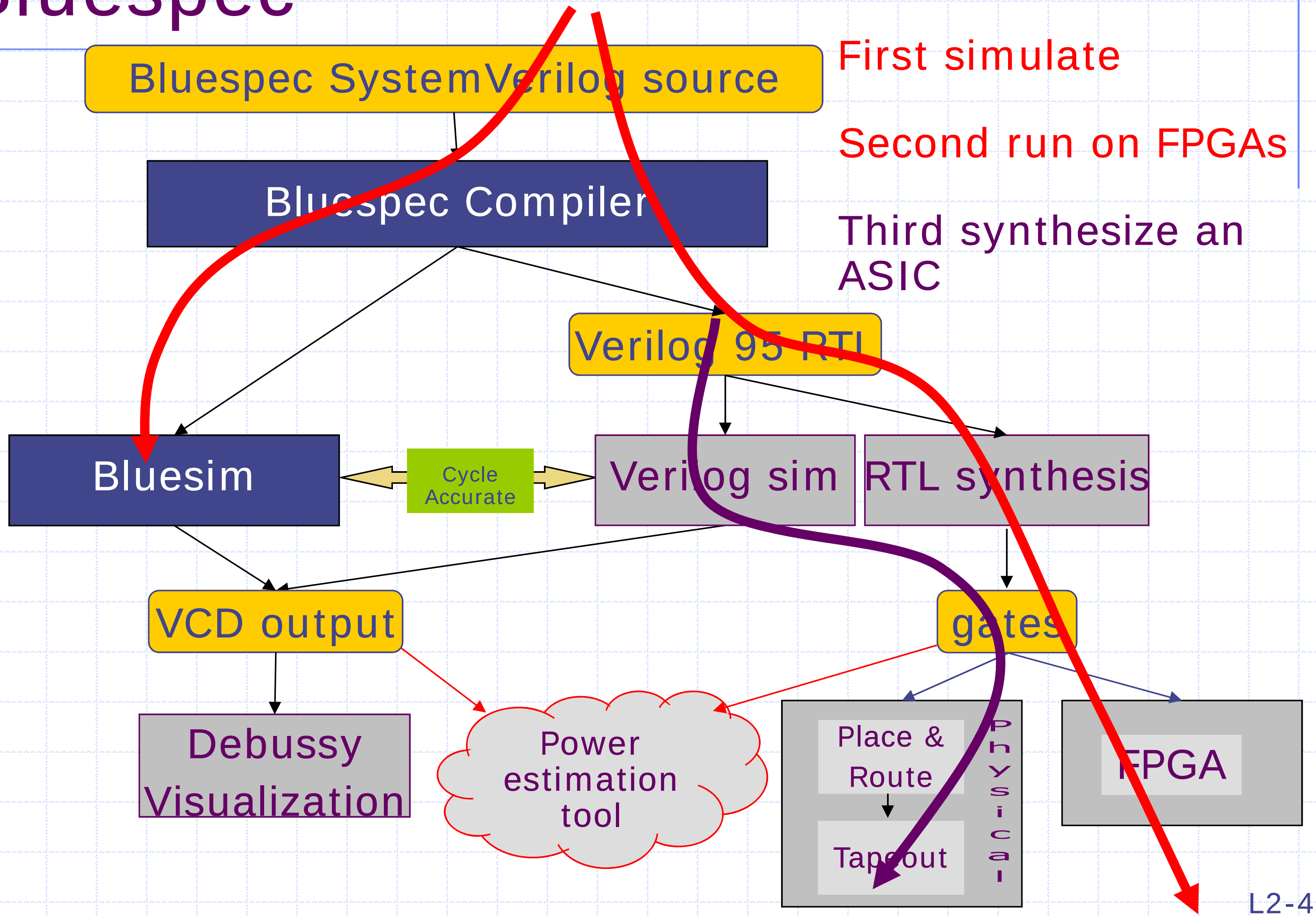
```
rule getResult;
   let x <- gcd.getResult;
   outQ.enq(x);
endrule
```

Draw a hardware circuit for these rules

High-level Synthesis from Bluespec



High-level Synthesis from Bluespec

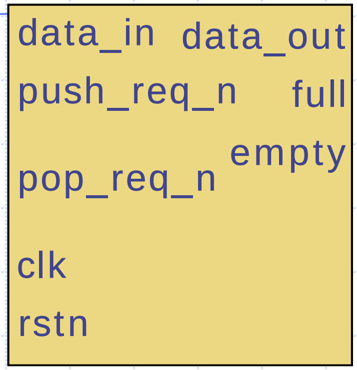


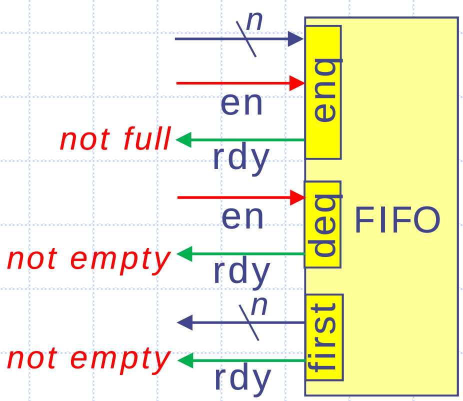
First simulate

Second run on FPGAs

Third synthesize an ASIC

Takeaway





- What makes the FIFO in BSV more useful is its precise interface definition and properties
- Modular refinement requires latency-insensitive designs, which are naturally supported by
  - Guarded interfaces
  - Guarded atomic actions, which provide the glue to connect modules, and which support synchrony of actions across modules

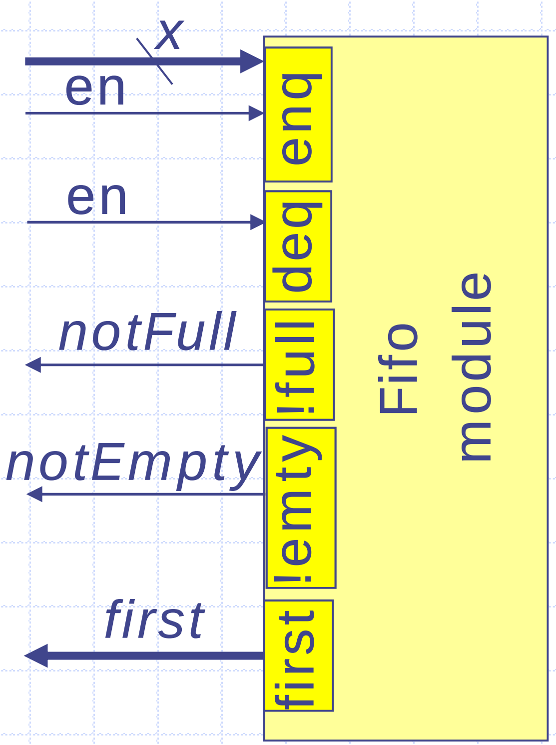
next lecture - parallel execution of rules and BSV semantics

Extras

FIFO Interface without guards

```
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
```

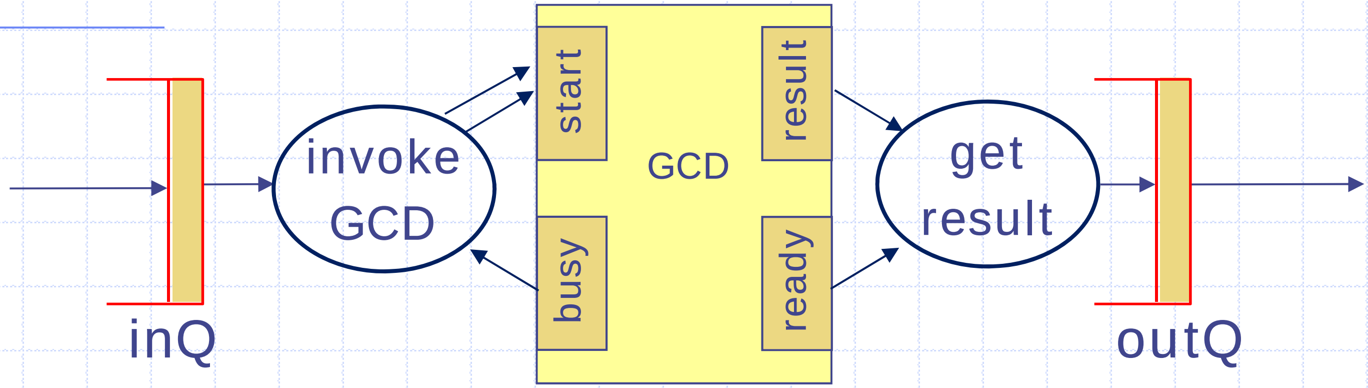
Type variable



- enq should be called only if notFull returns True;

- deq and first should be called only if notEmpty returns True

Streaming GCD (without guards)




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

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

Action value method

Guards vs Ifs

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

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

versus

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

guard is True, i.e., the method is always applicable.

if v is true then x would get lost; bad