Sequential Circuits: Modules with Guarded Interfaces

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

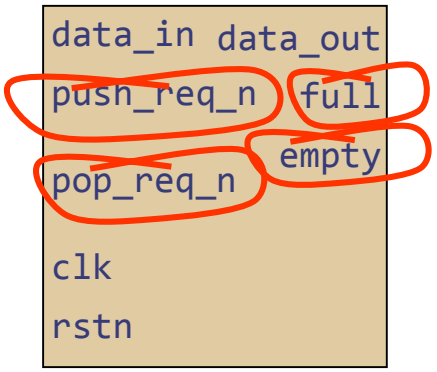
What is needed to make hardware design easier

- 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.

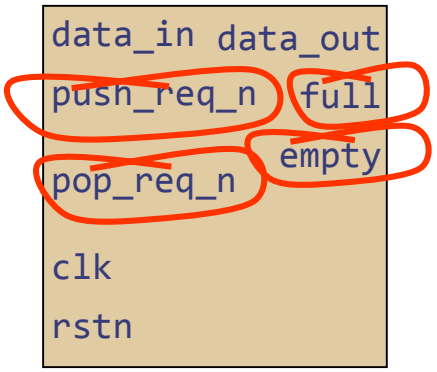
Thus, there is no conflict in a simultaneous push and pop when the FIFO is full. A simultaneous push and pop cannot occur when the FIFO is empty, since there is no pop data to prefetch. However, push data is captured in the FIFO.

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 FI[illegible]l.

Thus, there is no conflict in a simu[illegible]hen the FIFO is full. A simultaneous push and pop[illegible]IFO is empty, since there is no pop data to prefetch. How[illegible]ed in the FIFO.

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

No machine verification of such informal constraints is feasible

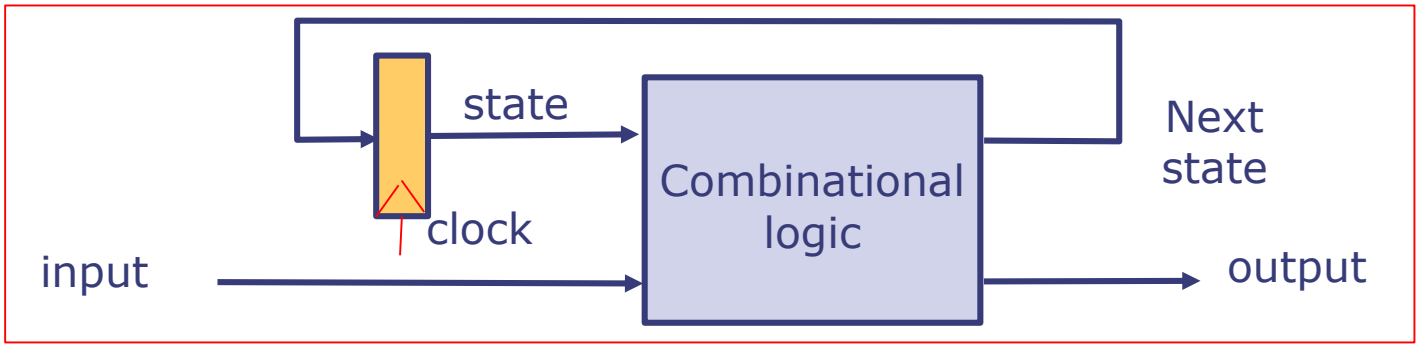
These constraints are spread over many pages of the documentation...

A different view of Digital Hardware

- Complex Digital Systems are a collection of cooperating sequential machines, which all operate concurrently
- A sequential machine is like an object in an Object-Oriented language like C++ or Java
- A sequential machine can be manipulated only via its interface methods

…but first the basics

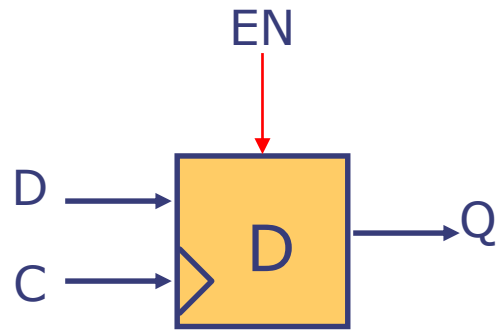
Finite State Machines (FSMs)

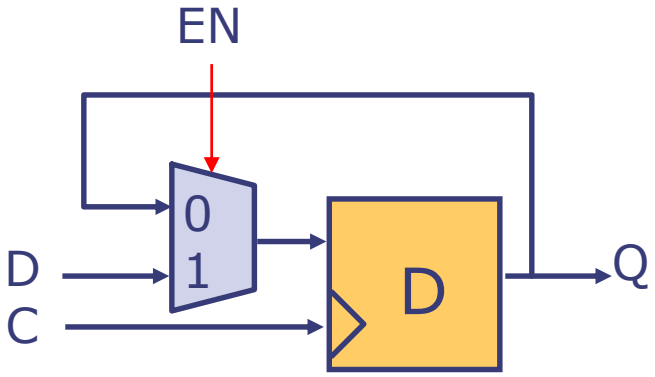


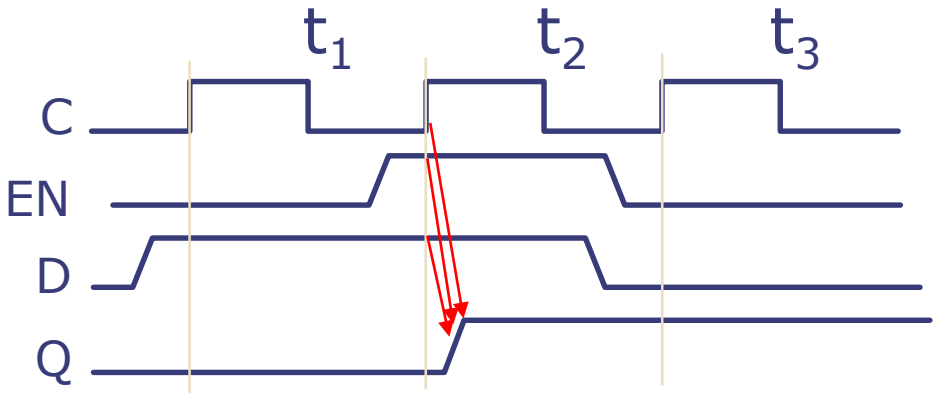
- FSMs are a much studied mathematical object like the Boolean Algebra
  - FSMs are used extensively in software as well
  - A computer (in fact any digital hardware) is an FSM, though we don't think of it as such!
- Synchronous Sequential Circuits are a method to implement FSMs in hardware

D Flip-flop with Write Enable

The building block of Sequential Circuits





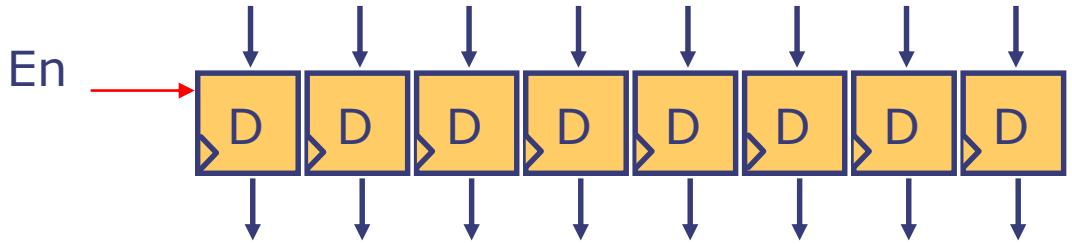


EN	D	Q^t	Q^{t+1}	
0	X	0	0	hold
0	X	1	1	
1	0	X	0	copy input
1	1	X	1	

Data is captured only if EN is on

No need to show the clock explicitly

Registers



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

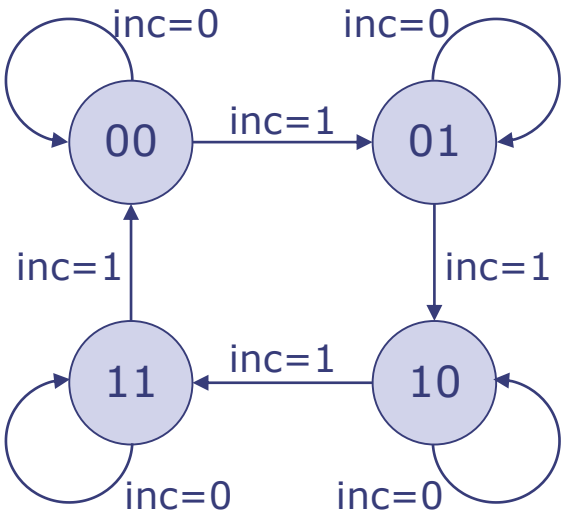
Register file: A group of registers with a common clock, a shared set of input and output ports

Clocked Sequential Circuits

- In this class we will deal with only clocked sequential circuits
- We will also assume that all flip-flops are connected to the same clock
- To avoid clutter, the clock input will be implicit and not shown in diagrams
- Clock inputs are not needed in BSV descriptions unless we design multi-clock circuits

An example
Modulo-4 counter

Prev State	NextState	
q1q0	inc = 0	inc = 1
00	00	01
01	01	10
10	10	11
11	11	00



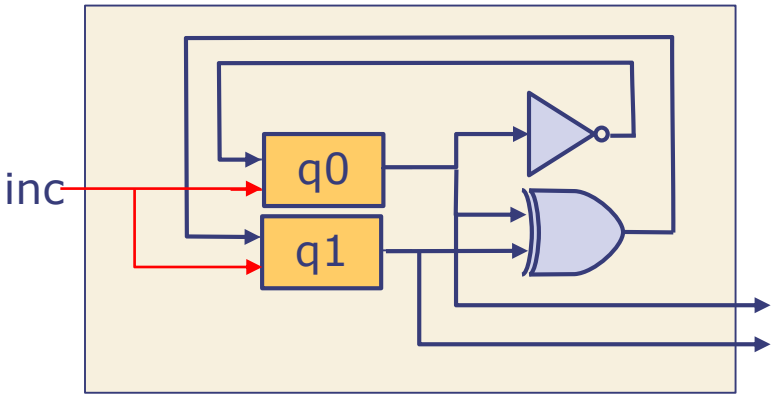
Finite State Machine (FSM) representation

$$q0^{t+1} = \sim inc \cdot q0^{t} + inc \cdot \sim q0^{t}$$
$$= inc \oplus q0^{t}$$
$$q1^{t+1} = \sim inc \cdot q1^{t} + inc \cdot \sim q1^{t} \cdot q0^{t} + inc \cdot q1^{t} \cdot \sim q0^{t}$$
$$= \sim inc \cdot q1^{t} + inc \cdot (q1^{t} \oplus q0^{t})$$

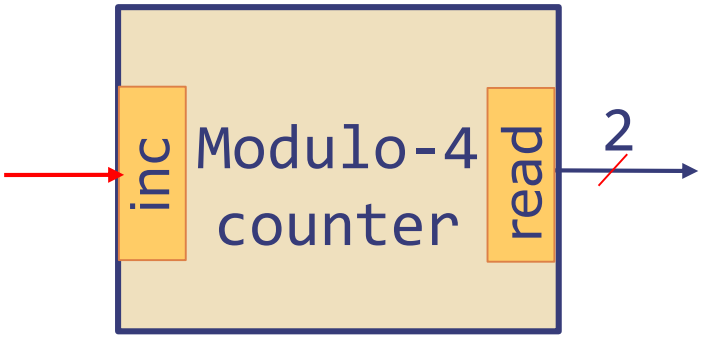
Circuit for the modulo counter using D flip-flops with enables

- Use two D flip-flops, q0 and q1, to store the counter value
- Notice, the state of flip-flop changes only when inc is true

$$\{q1^{t+1}, q0^{t+1}\} = \{(q1^{t} \oplus q0^{t}),\ q0^{t}\}$$ (assume inc is True)



Sequential Circuit as a module with Interface

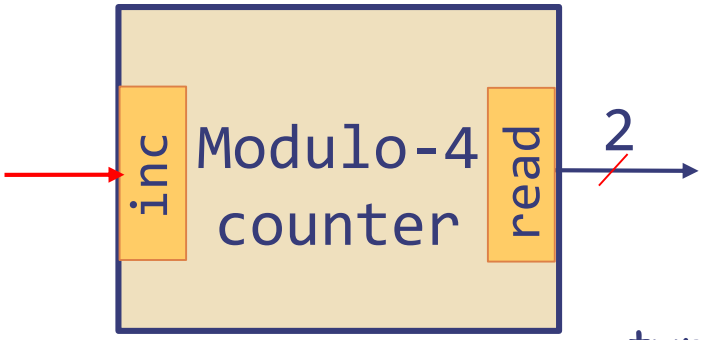



```
interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface
```

- A module has internal state and an interface
- The internal state can be read and manipulated only by its interface methods
- An *action* method specifies which state elements are to be modified; it has an *enable* wire which must be true to execute the action
- Actions are *atomic* -- either all the specified state elements are modified or none of them are modified (no partially modified state is visible)
- Informally we refer to the *interface* of a module as its type

A module in Bluespec is like a class definition in Java or C++

Modulo-4 counter: An implementation 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^t+1 <= {cnt^t[1]^cnt^t[0],~cnt^t[0]};
  endmethod
  method Bit#(2) read;
    return cnt;
  endmethod
endmodule
```

type instantiate

State specification

Initial value

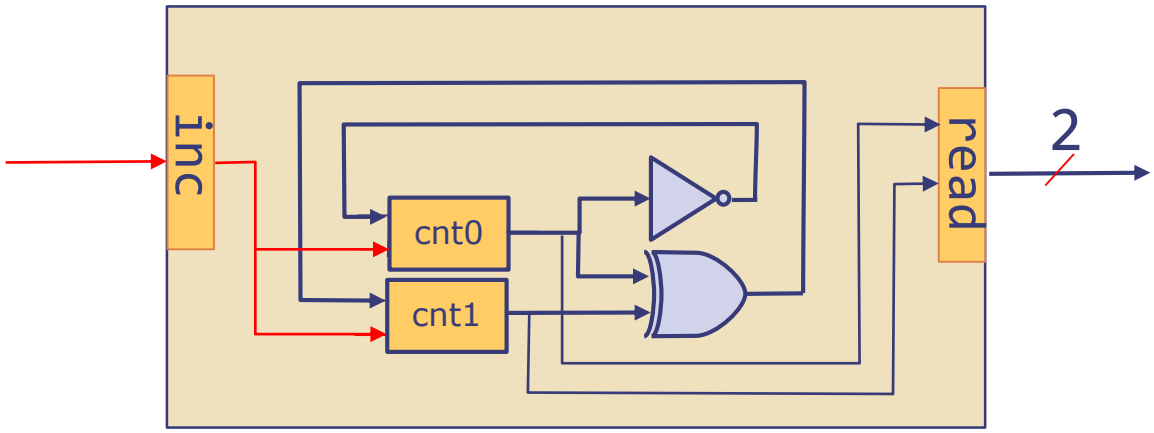
Register assignment

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

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

Modulo-4 counter

The generated circuit



1

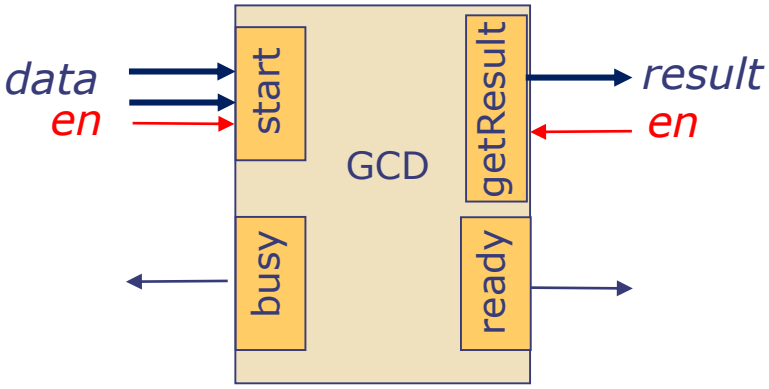
```
module mkCounter(Counter);
   Reg#(Bit#(2)) cnt <- mkReg(0);
   method Action inc;
      cnt <={cnt[1]^cnt[0],~cnt[0]};
   endmethod
   method Bit#(2) read;
      return cnt;
   endmethod
endmodule
```

GCD: Euclid's Algorithm

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

a: 15	b: 6	
9	6	*subtract*
3	6	*subtract*
6	3	*swap*
3	3	*subtract*
0	3	*subtract*
	answer	

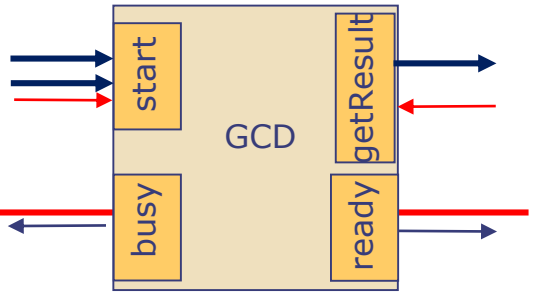
GCD module



GCD can be started if the module is not *busy*; Results can be read when *ready*

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

GCD implementation



Instantiate state

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

  rule gcd;
     if (x >= y) begin x <= x - y; end //subtract
     else if (x != 0) begin x <= y; y <= x; end //swap
  endrule
  method Action start(Bit#(32) a, Bit#(32) b) ;
    x <= a; y <= b; busy_flag <= True;
  endmethod
  method ActionValue#(Bit#(32)) getResult ;
    busy_flag <= False; return y;
  endmethod
  method Bool busy
         = busy_flag;
  method Bool ready
         = (x==0);
endmodule
```

Type

Rule gcd will execute repeatedly until x becomes 0

Assume b != 0

start should be called only if the busy is false; getResult should be called only when ready is true.

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

Method calls

- Value method: Only observe the internal state
  - `let counterValue = mod4counter.read;`
  - `Bool isGcdBusy = gcd.busy;`
- Action method: Updates the state of the module
  - `mod4counter.inc;`
  - `gcd.start(13,27);`
- ActionValue#(t): Updates the state and returns a value
  - `let resultGcd <- gcd.getResult;`

Notice the use of '<-' instead of '='

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

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
```

What is meaning of this?

Swap!

- A rule has a name (e.g., gcd)
- A rule is a collection of actions, which invoke methods
- All actions in a rule execute in parallel
- A rule can execute any time and when it executes all of its actions must execute

atomicity

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
```

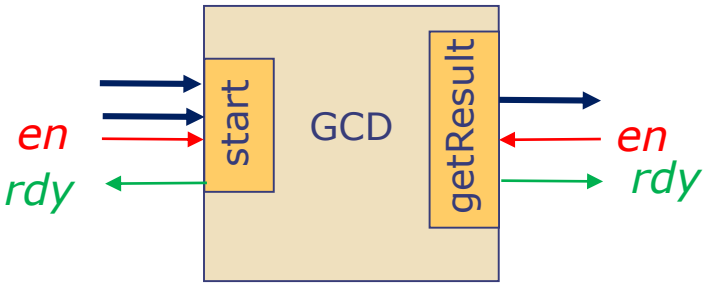
Possibility of a double write

- Parallel composition, 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

Guarded interfaces

- Make the life of the programmers easier: Include some checks (not busy, ready, ...) in the method definition itself, so that the user does not have to test the applicability of the method explicitly from outside

- Guarded Interface:
  - Every method has a *guard* (*rdy* wire)
  - The value returned by a method is meaningful only if its guard is true
  - Every action method has an *enable signal* (*en* wire) and it can be invoked (en can be set to true) only if its guard is true




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

en and rdy wires are implicit; derived by the type of the method

GCD with Guards

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

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

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

Assume b != 0

Guard?

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

Rules with guards

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

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

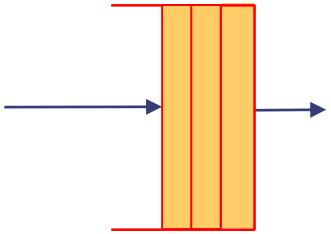
Syntax: In rules, "if" is optional before the guard!

- 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. Equivalently, the absence of a guard means the guard is always true
- For example we can attach a guard to gcd to prevent its unnecessary execution:

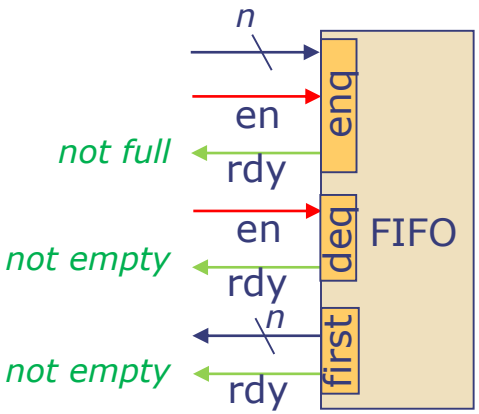
```
rule gcd if (busy_flag);
    if (x >= y) begin x <= x - y; end //subtract
    else if (x != 0) begin x <= y; y <= x; end //swap
endrule
```

First-In-First-Out queue (FIFO)

- FIFO data structure is used extensively both in hardware and software to connect *things*



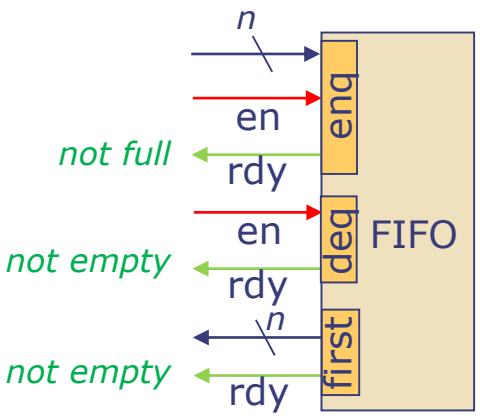
- In hardware, fifo have fixed size which is often as small as 1, and therefore the producer blocks when enqueuing into a full fifo and the consumer blocks when dequeueing from an empty fifo



One-Element FIFO Implementation
with guards

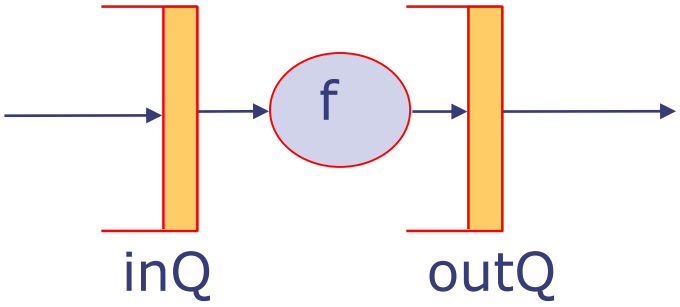
```
module mkFifo (Fifo#(1, Bit#(n)));
  Reg#(Bit#(n))    d  <- mkRegU;
  Reg#(Bool) v  <- mkReg(False);
  method Action enq(Bit#(n) x) if (!v);
    v <= True; d <= x;
  endmethod
  method Action deq  if (v);
    v <= False;
  endmethod
  method Bit#(n) first if (v);
    return d;
  endmethod
endmodule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

Syntax: lack of semicolon turns the if into a guard



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

Streaming a function using a FIFO with guarded interfaces



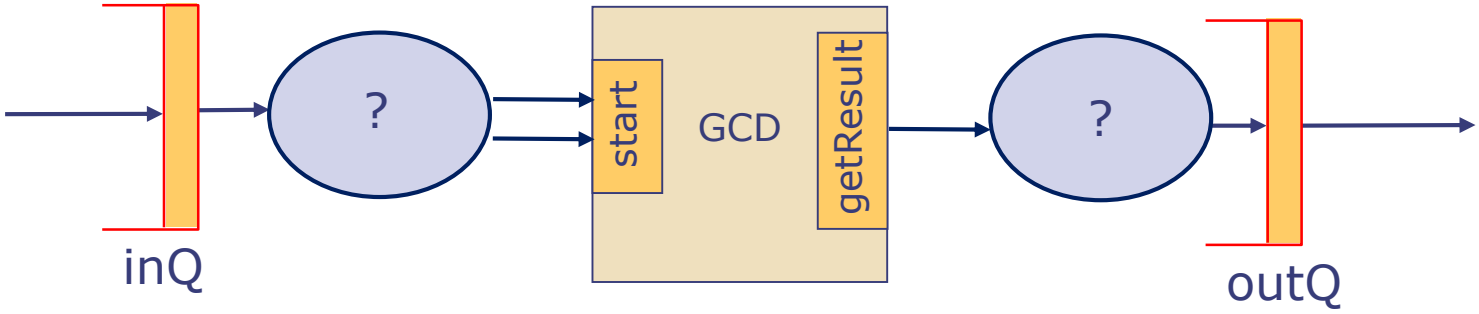

```
if(inQ.notEmpty && outQ.notFull)
```

implicit guard

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

Streaming a module

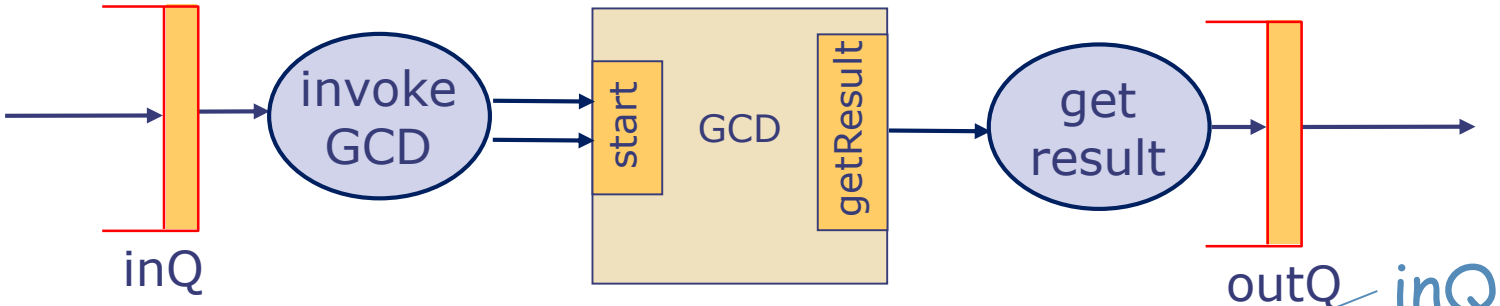


- Suppose we have a queue of pairs of numbers and we want to compute their GCDs and put the results in an output queue
- We can build such a system by creating the following modules

```
Fifo#(1,Vector#(2,t)) inQ <- mkFifo;
Fifo#(1,t) outQ <- mkFifo;
GCD gcd <- mkGCD;
```

- To glue these modules together we define two rules:
  - `invokeGCD` to push data from `inQ` into `gcd`
  - `getResult` to fetch result from `gcd` and put it into `outQ`

Streaming a module: code




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

inQ is not empty

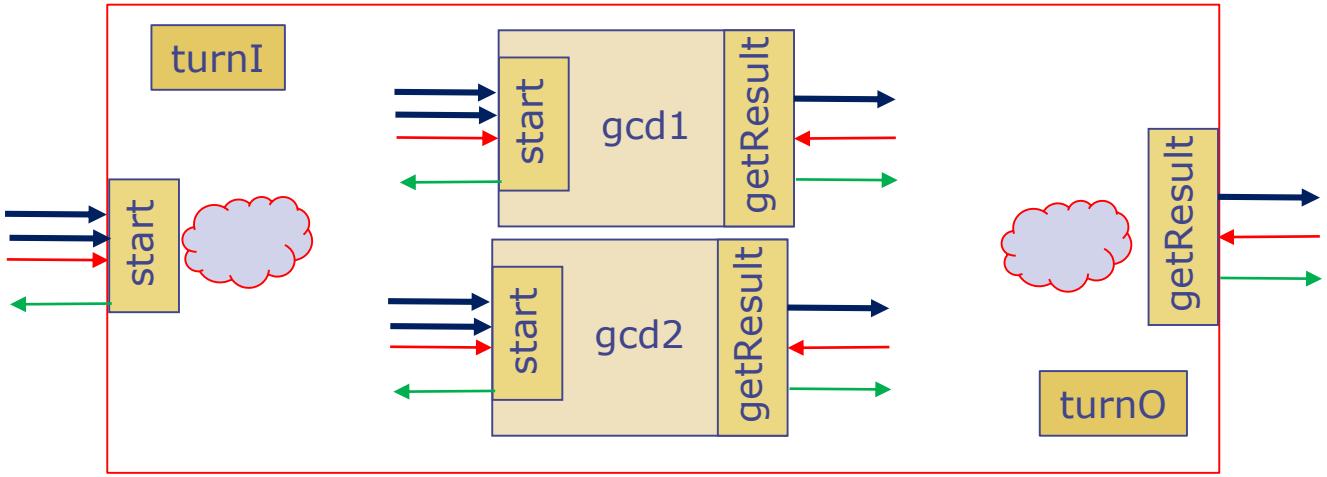
gcd is not busy

Implicit guards?

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

Action value method

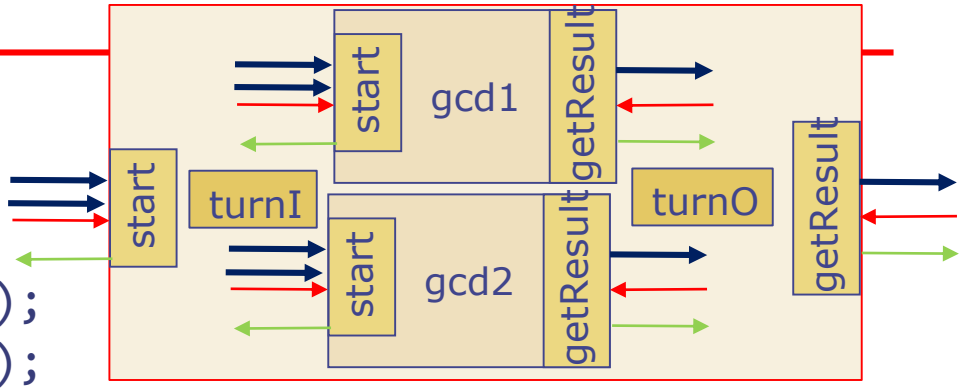
Power of Abstraction:
Another GCD implementation



- A GCD module with the same interface but with twice the throughput; uses 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
```

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
```

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