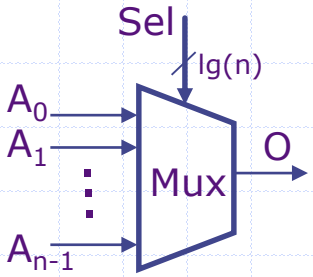
Constructive Computer Architecture

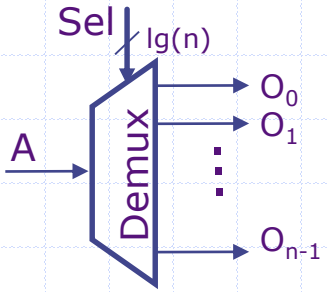
Sequential Circuits:

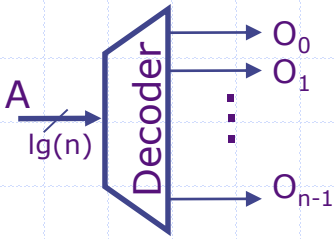
Circuits with state

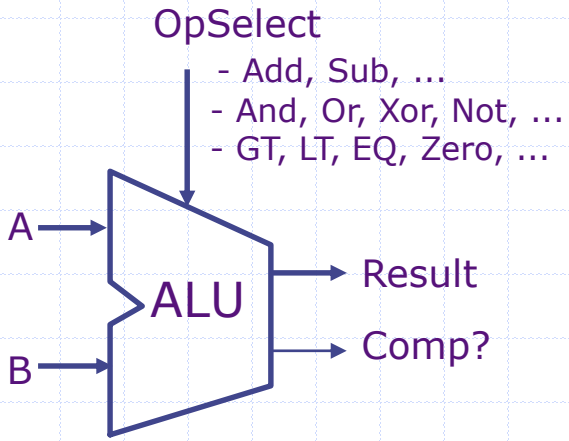
Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Combinational circuits



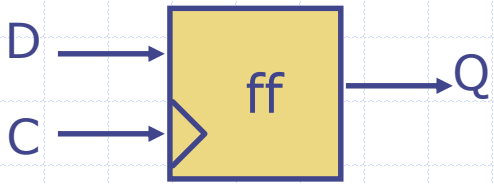


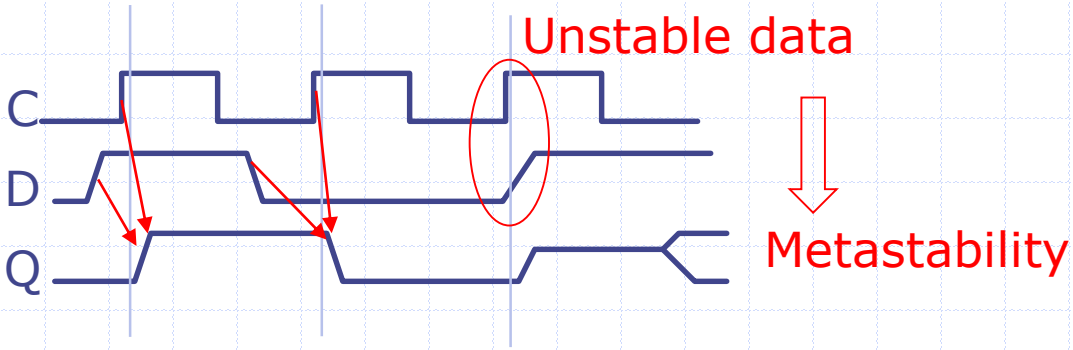




Such circuits have no cycles (feedback) or state elements

Edge-Triggered Flip flop: the basic storage element

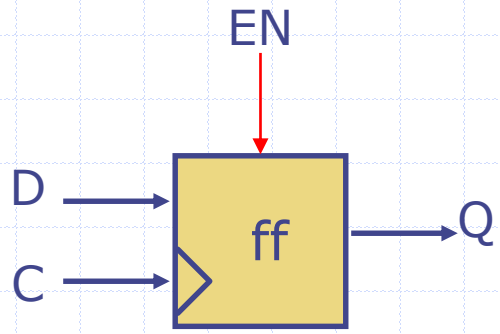


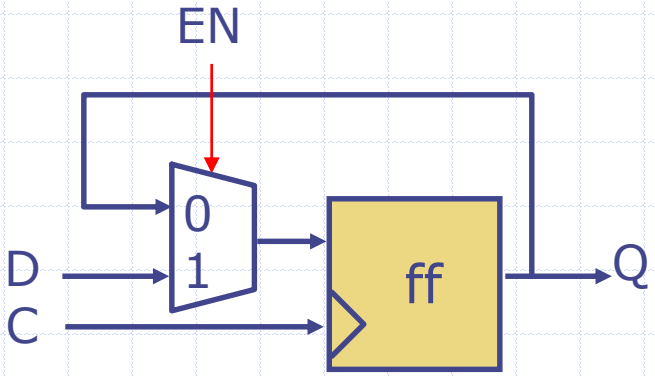


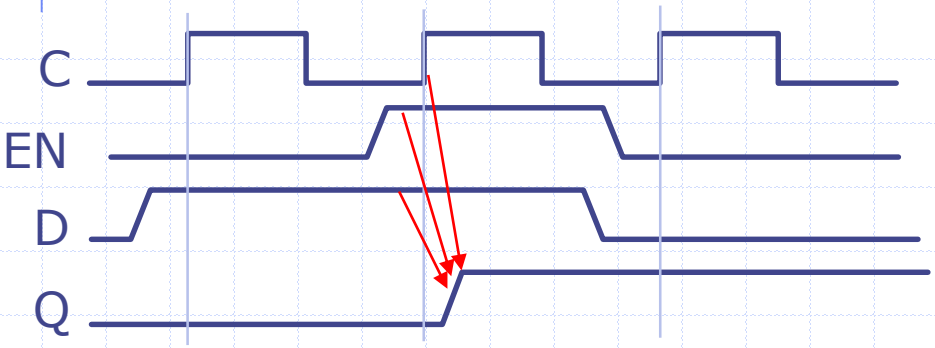
Data is sampled at the rising edge of the clock and must be stable at that time

Flip-flops with Write Enables:

The building block of Sequential Circuits





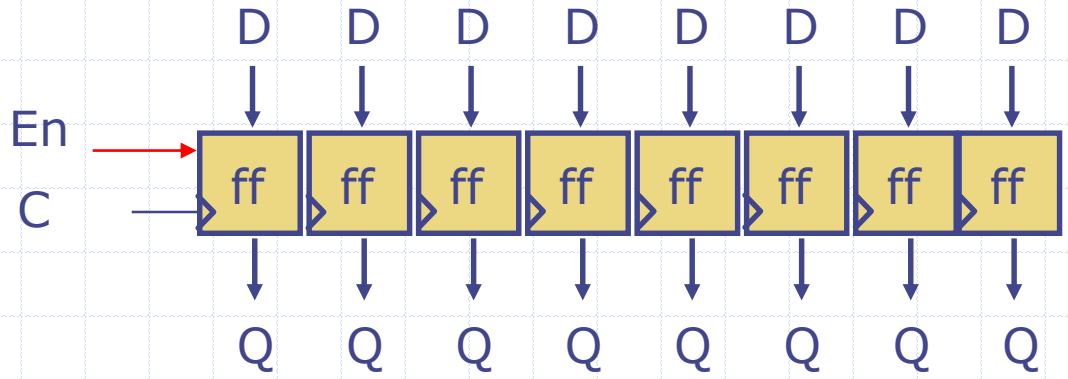


EN	D	Q^t	Q^{t+1}	
0	X	0	0	hold
0	X	1	1	
1	0	X	0	copy input
1	1	X	1	

Data is captured only if EN is on

No need to show clock explicitly

Registers



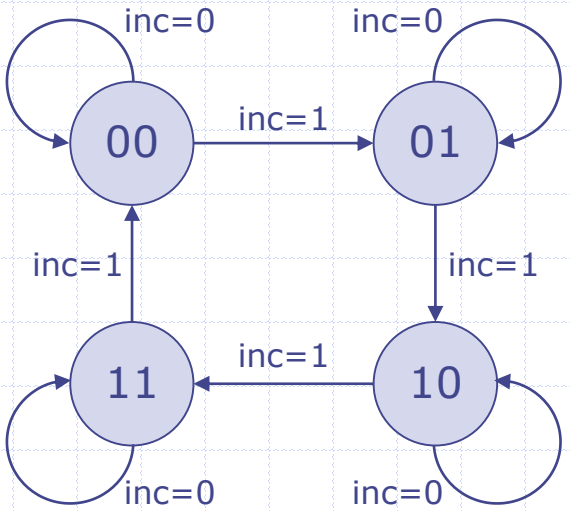
Register: A group of flip-flops with a common clock and enable

Register file: A group of registers with a common clock, input and output port(s)

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

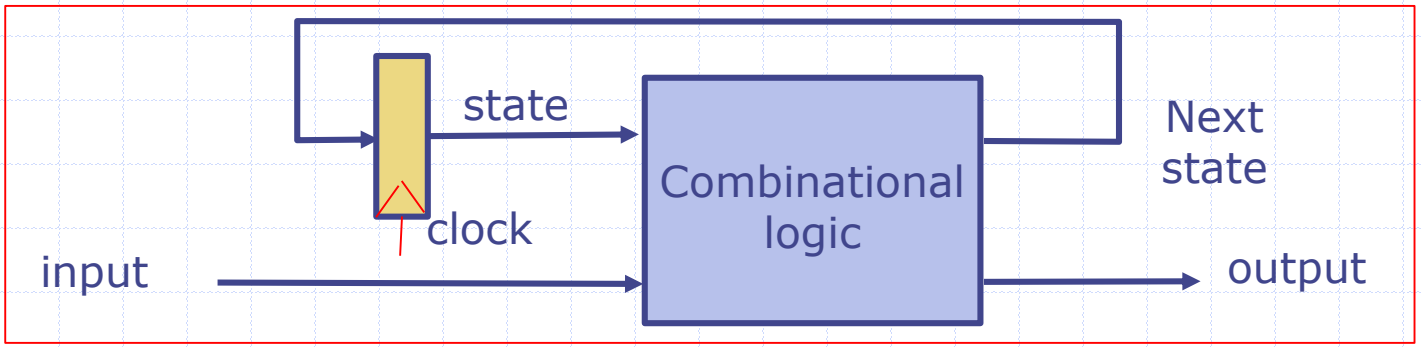


Finite State Machine (FSM) representation

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

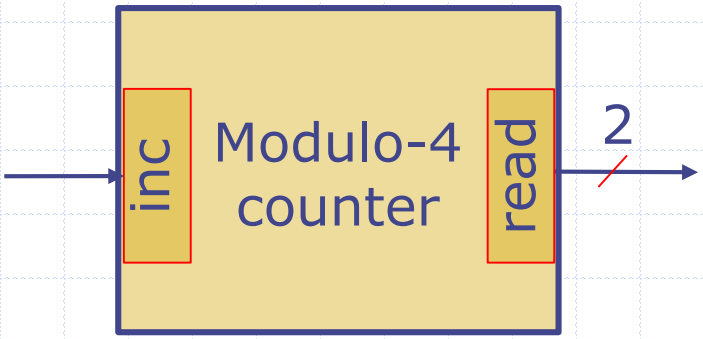
Finite State Machines (FSM) and Sequential Ckts

- 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*
  - State diagrams and next-state tables are not suitable for such descriptions

Modulo-4 counter in BSV




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

```
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
```

State specification

Initial value

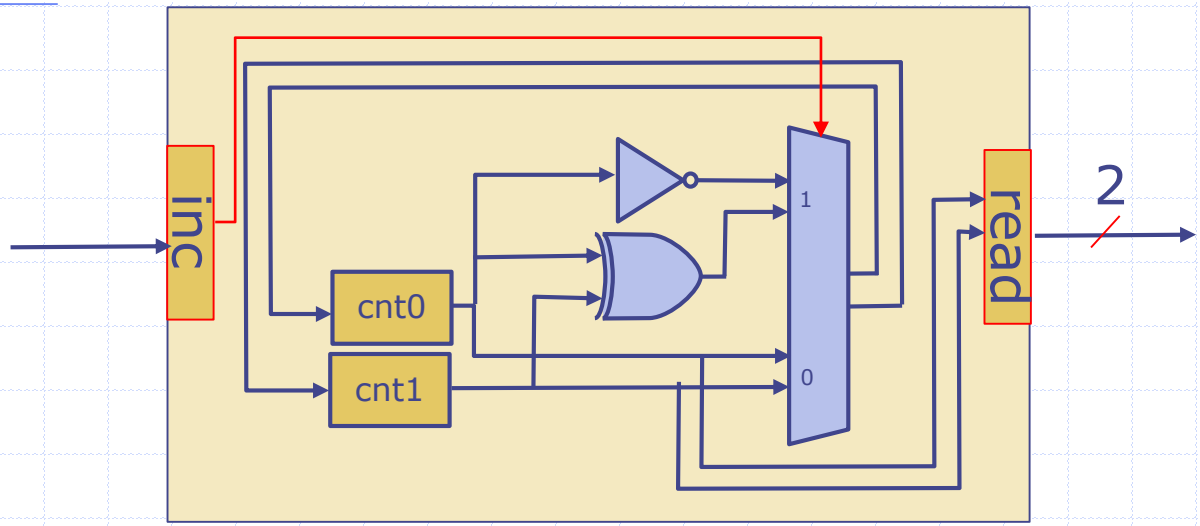
An *action* to specify how the value of the cnt is to be set

Modules

- A module in BSV is like a class definition in Java or C++
  - It 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
```

Inside the Modulo-4 counter




```
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
```

Examples

A hardware module for computing GCD

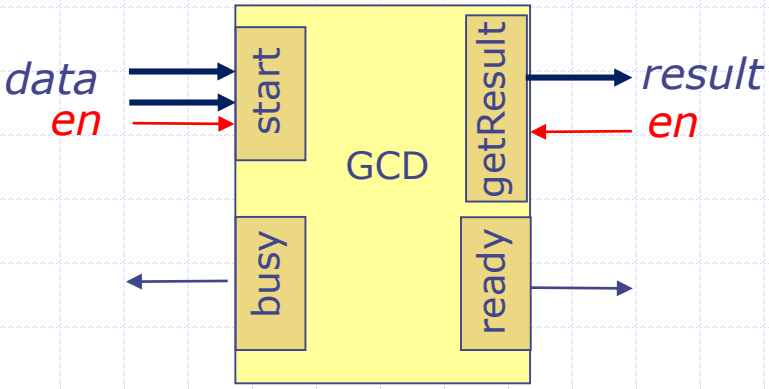
Euclid's algorithm for computing the Greatest Common Divisor (GCD):

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

answer

```
gcd (a,b) = if a==0 then b                    \\ stop
            else if a>=b then gcd(a-b,b) \\ subtract
            else gcd (b,a)                    \\ swap
```

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 in BSV

```
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 busy = busy_flag;
method ready = x==0;
endmodule
```

Assume b /= 0

`start` should be called only if the module is not `busy`; `getResult` should be called only when ready is `true`.

Rule

A module may contain rules

parallel composition of actions

```
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

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
```

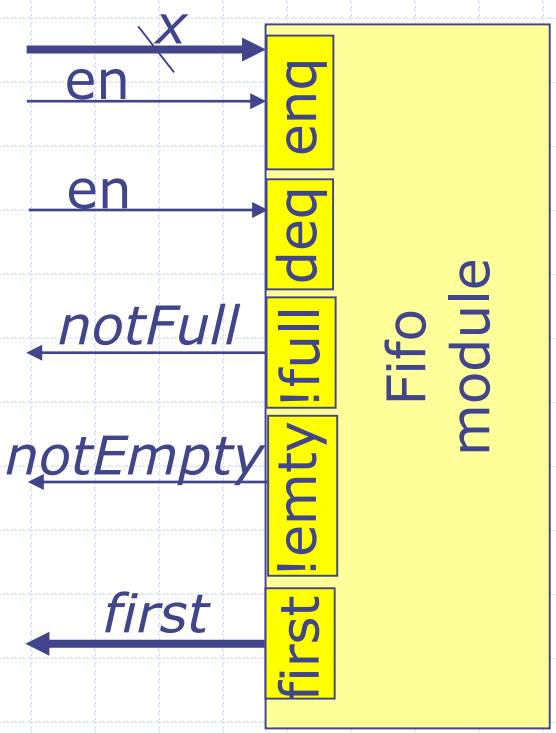
Possibility of a double write

- Parallel composition, and consequently a rule containing it, is illegal if a double-write possibility exists
- The BSV compiler rejects a program if it there is a possibility of a double write

Defining FIFOs and it's uses

FIFO Module Interface

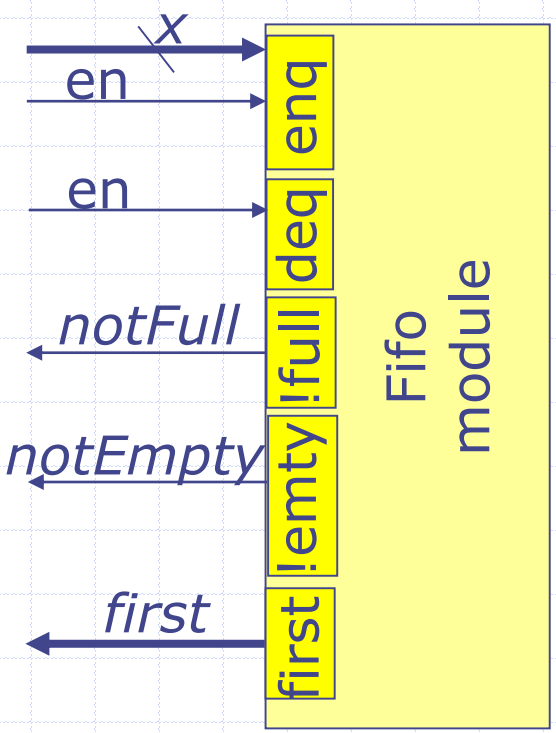
```
interface Fifo#(numeric type size, type t);
  method Bool notFull;
  method Bool notEmpty;
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface
```



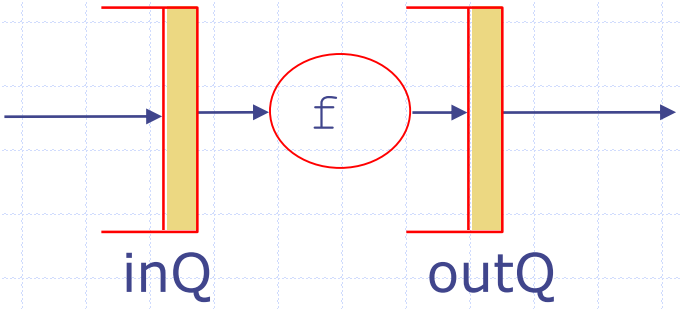
- `enq` should be called only if `notFull` returns True;
- `deq` and `first` should be called only if `notEmpty` returns True

An Implementation: One-Element FIFO

```
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
```



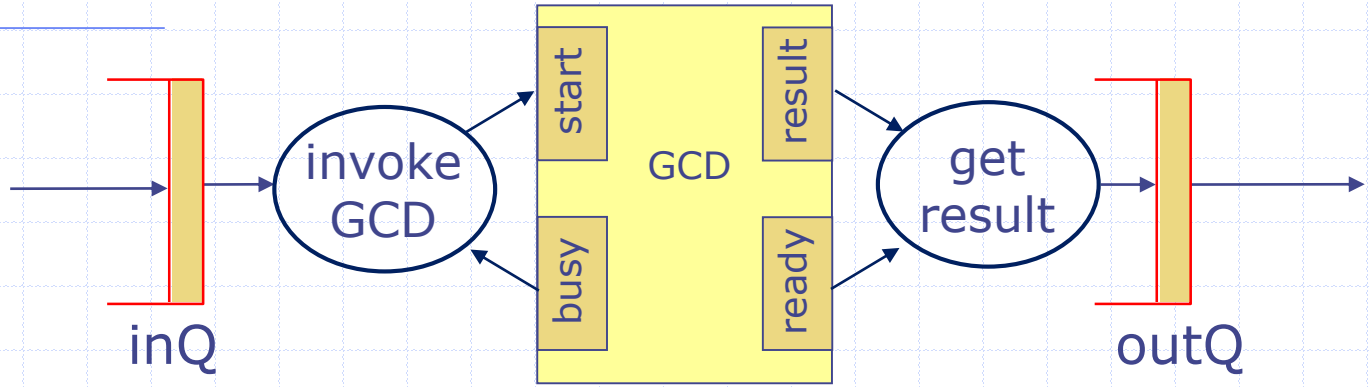
Streaming a function




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

Boolean & ("AND") operation

Streaming a module



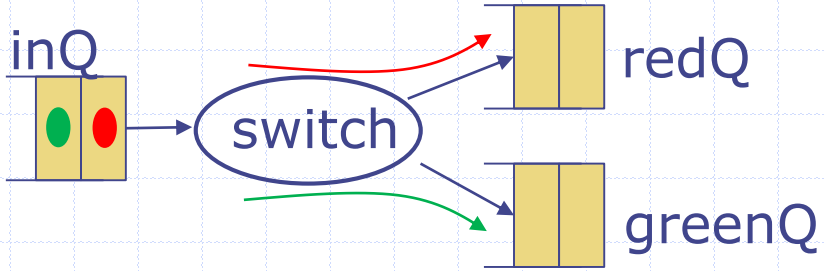

```
rule invokeGCD;
  if(inQ.notEmpty && !gcd.busy)
    begin gcd.start(inQ.first); inQ.deq; end
endrule
```

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

Action value method

Switch

red messages go into redQ, green into greenQ

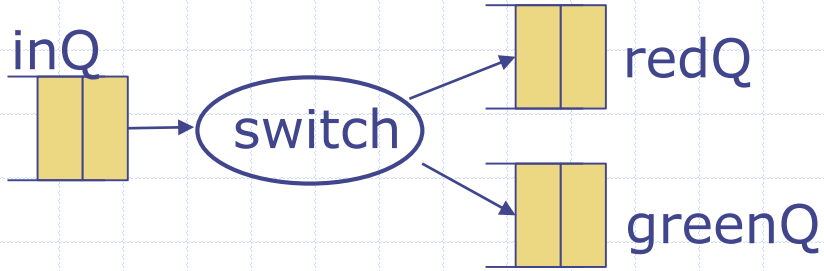


red messages go into redQ, green into greenQ

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

The code is not correct because it does not include tests for empty inQ or full redQ or full greenQ conditions!

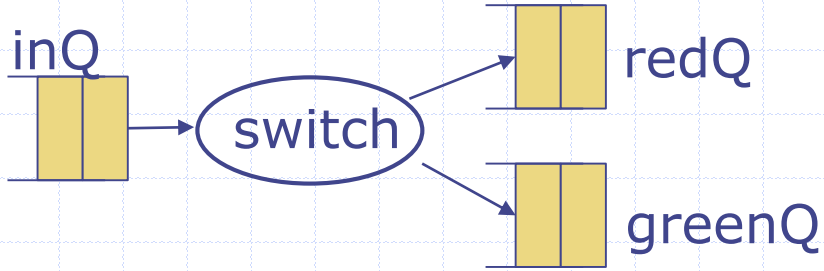
Switch with empty/full tests on queues - 1




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

first and deq operations can be performed only if inQ is not empty

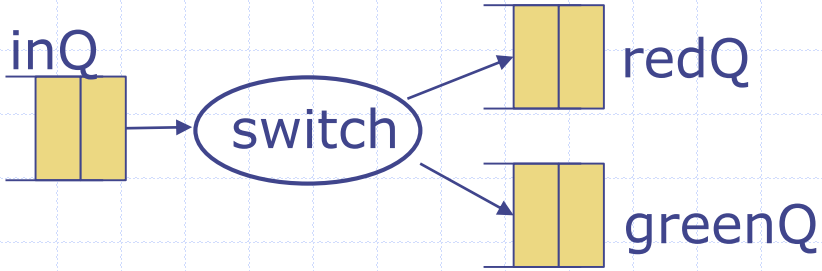
Switch with empty/full tests on queues -2




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

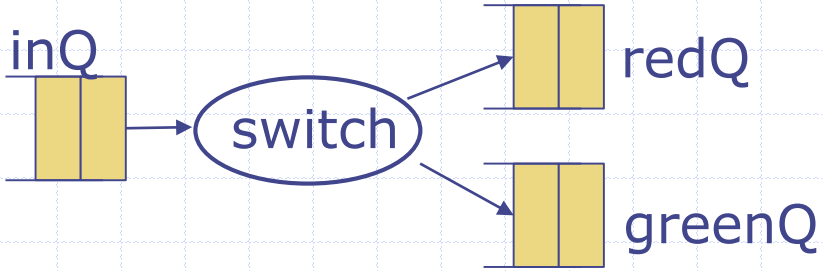
When can an enq operation be performed on redQ?

Switch with empty/full tests on queues




```
rule switch;
 if (inQ.notEmpty)
  if (inQ.first.color == Red) begin1
   if (redQ.notFull) begin2
    redQ.enq(inQ.first.value); inQ.deq;
    end2
   end1
  else begin3
   if (greenQ.notFull) begin4
    greenQ.enq(inQ.first.value); inQ.deq;
    end4
  end3
 endrule
```

A wrong optimization




```
rule switch;
 if (inQ.notEmpty)
  if (inQ.first.color == Red) begin1
   if (redQ.notFull) begin2
    redQ.enq(inQ.first.value); inQ.deq;
    end2
   end1
  else begin3
   if (greenQ.notFull) begin4
    greenQ.enq(inQ.first.value); inQ.deq;
    end4
  end4
  inQ.deq;
endrule
```

inQ value may get lost if redQ (or greenQ) is full

Atomicity violation!

Can we move the deq here?

Observations

- These sample programs are not very complex and yet it would have been tedious to express these programs in a state table or as a circuit directly
- The meaning of double-write errors is not standardized across Verilog tools
- Interface methods are not available in Verilog/VHDL