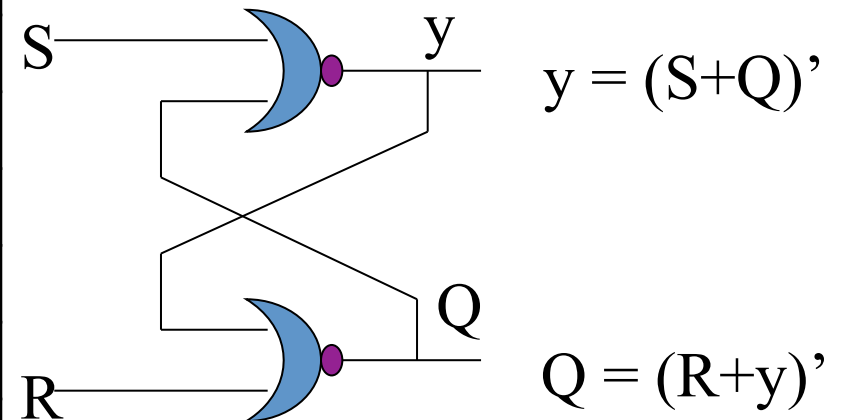


CSE 140 Discussion

			t0		t1		t2		t3	
id	S	R	Qt	yt	Qt	yt	Qt	yt	Qt	yt
0	0	0	0	0						
1	0	0	0	1						
2	0	0	1	0						
3	0	0	1	1						
4	0	1	0	0						
5	0	1	0	1						
6	0	1	1	0						
7	0	1	1	1						
8	1	0	0	0						
9	1	0	0	1						
10	1	0	1	0						
11	1	0	1	1						
12	1	1	0	0						
13	1	1	0	1						
14	1	1	1	0						
15	1	1	1	1						

Truth table of SR latch with incremental steps in time

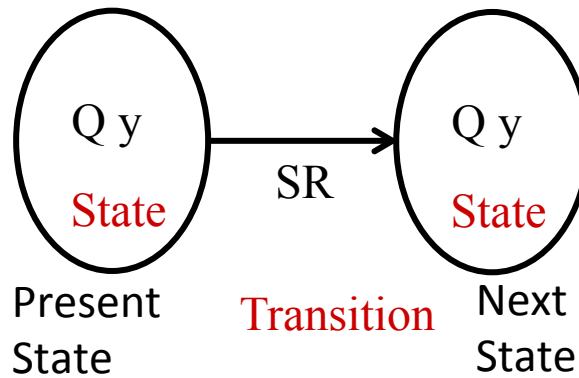


			t0		t1		t2		t3	
id	S	R	Qt	yt	Qt	yt	Qt	yt	Qt	yt
0	0	0	0	0	1	1	0	0	1	1
1	0	0	0	1	0	1	0	1	0	1
2	0	0	1	0	1	0	1	0	1	0
3	0	0	1	1	0	0	1	1	0	0
4	0	1	0	0	0	1	0	1	0	1
5	0	1	0	1	0	1	0	1	0	1
6	0	1	1	0	0	0	0	1	0	1
7	0	1	1	1	0	0	0	1	0	1
8	1	0	0	0	1	0	1	0	1	0
9	1	0	0	1	0	0	1	0	1	0
10	1	0	1	0	1	0	1	0	1	0
11	1	0	1	1	0	0	1	0	1	0
12	1	1	0	0	0	0	0	0	0	0
13	1	1	0	1	0	0	0	0	0	0
14	1	1	1	0	0	0	0	0	0	0
15	1	1	1	1	0	0	0	0	0	0

“State Table” of SR latch

Qy\SR	00	01	10	11
00				
01				
10				
11				

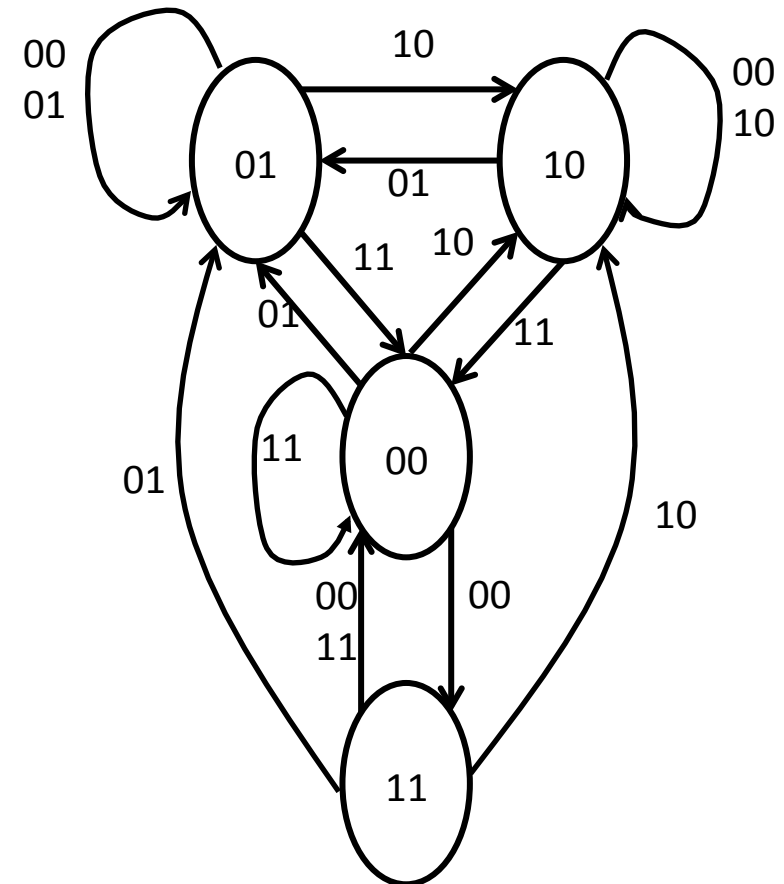
State Table and State Diagram



State diagram

State Table

\SR Qy	00	01	10	11
00	11	01	10	00
01	01	01	10	00
10	10	01	10	00
11	00	01	10	00



SR Latch

SR=01 (RESET) : $(Q,y) = (0,1)$

SR=10 (SET) : $(Q,y) = (1,0)$

SR=00 (NO CHANGE)

So (Q,y) should not become $(0,0)$ or $(1,1)$

Solutions: Avoid the case that $SR = (1,1)$.

SR=11 (ILLEGAL) : $(Q,y) = (0,0)$

\SR Qy	00	01	10	11
00	11	01	10	00
01	01	01	10	00
10	10	01	10	00
11	00	01	10	00

State table

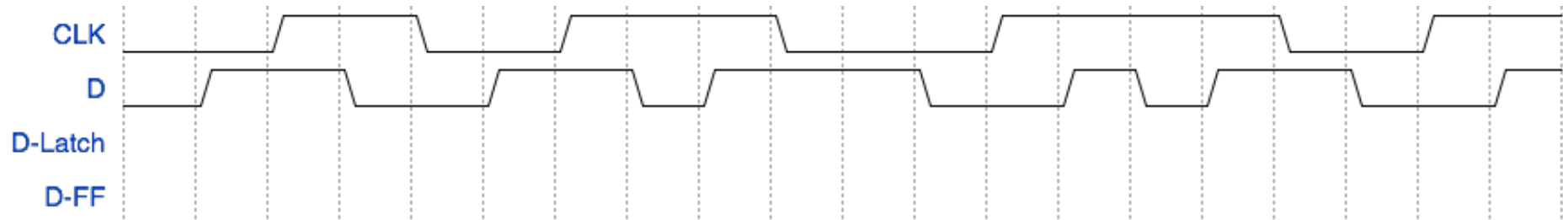
		SR			
		00	01	11	10
Q(t)	0	0	0	-	1
	1	1	0	-	1

Q(t+1) NS (next state)

Characteristic Expression

$$Q(t+1) = S(t) + R'(t)Q(t)$$

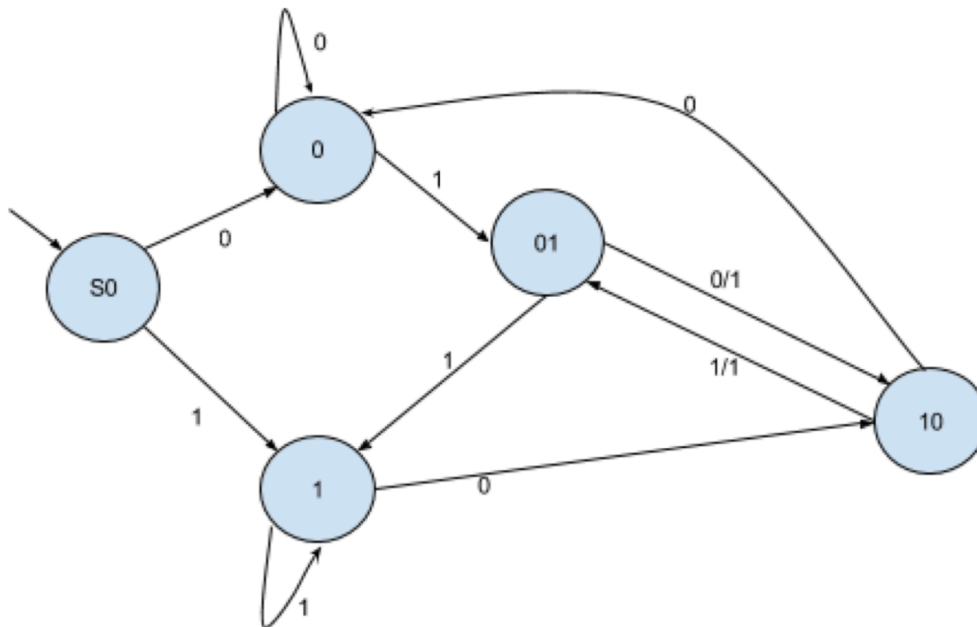
D-Latch vs. D-FF



- D-Latch
 - Analogy: Door
 - $\text{CLK} = 1 : Q = D$
 - $\text{CLK} = 0 : Q = Q_{\text{prev}}$
- D-FF
 - Analogy: Camera
 - $\text{CLK} = 0 \text{ to } 1 : Q = D$
 - Otherwise : $Q = Q_{\text{prev}}$

Pattern Recognizer FSM

- Example:
 - $y = 1$ if $x(t - 2, t) = 101$ or 010
 - $y = 0$ otherwise



State Diagram

PS \ Input		X = 0	X = 1
--	S0	S1, 0	S2, 0
0	S1	S1, 0	S3, 0
1	S2	S4, 0	S2, 0
01	S3	S4, 1	S2, 0
10	S4	S1, 0	S3, 1

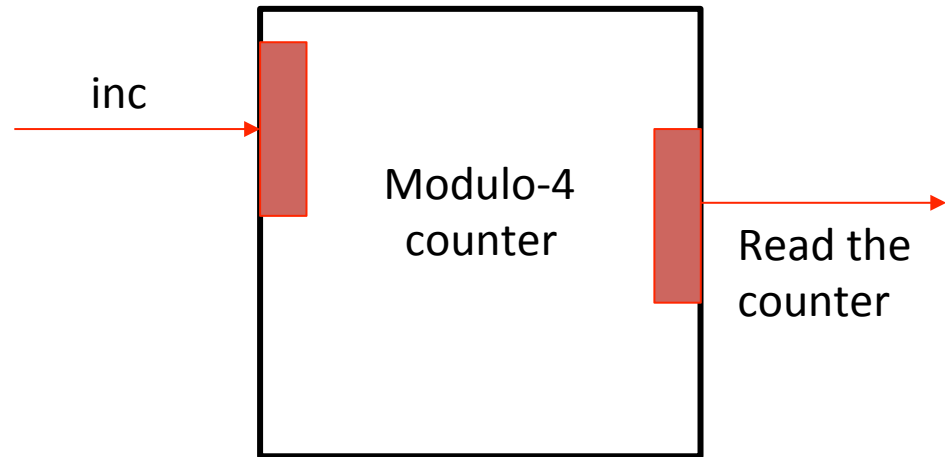
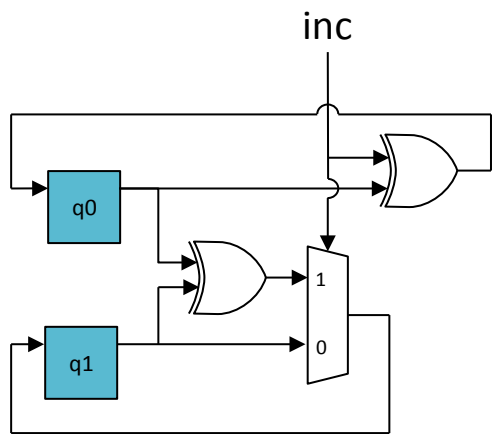
State Table

HW3 QUESTIONS?

Describing Sequential Ckts

- State diagrams and next-state tables nor are suitable for describing very large digital designs
 - large circuits must be described in a modular fashion -- as a collection of cooperating FSMs
- BSV is a modern programming language to describe cooperating FSMs

Modulo-4 counter circuit



$$q0^{t+1} = \sim inc \cdot q0^t + inc \cdot \sim q0^t$$

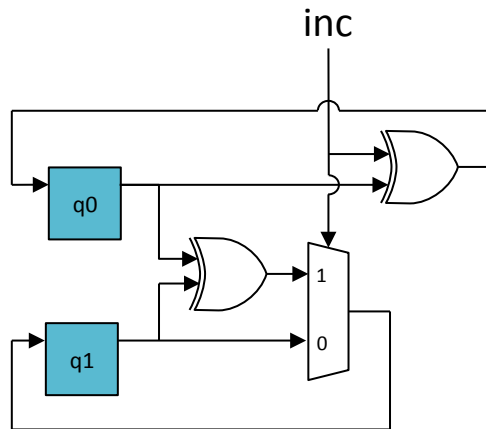
$$q1^{t+1} = \sim inc \cdot q1^t + inc \cdot \sim q1^t \cdot q0^t + inc \cdot q1^t \cdot \sim q0^t$$

“Optimized” logic

$$q0^{t+1} = inc \oplus q0^t$$

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

Modulo-4 counter circuit



$$q0^{t+1} = \sim inc \cdot q0^t + inc \cdot \sim q0^t$$

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

“Optimized” logic

$$q0^{t+1} = inc \oplus q0^t$$

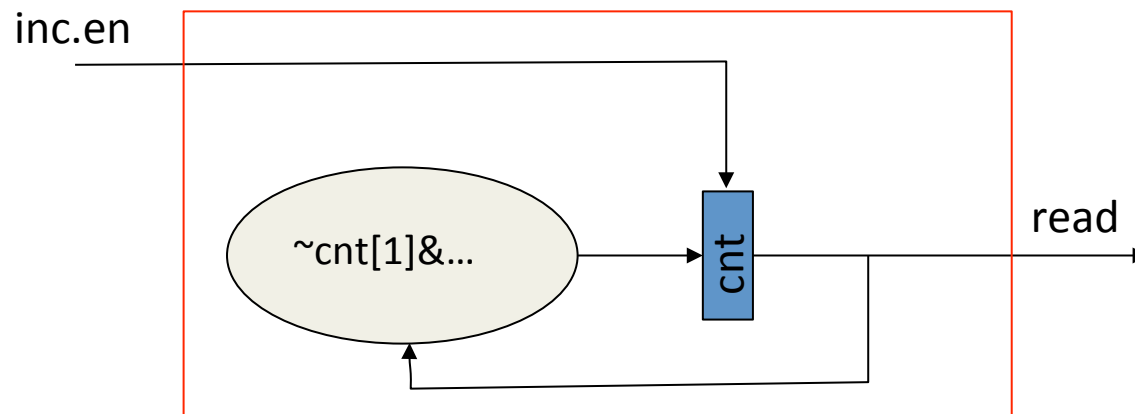
$$q1^{t+1} = (inc == 1) ? q0^t \oplus q1^t : q1^t$$

PS\input	inc=0	inc=1
00	00	01
01	01	10
10	10	11
11	11	00

PS: $q1^t q0^t$, NS: $q1^{t+1} q0^{t+1}$

modulo4 counter in BSV

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



Interface

- Modulo counter has the following interface, i.e., type

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

- An interface can have many different implementations
 - For example, the numbers may be represented as Gray code

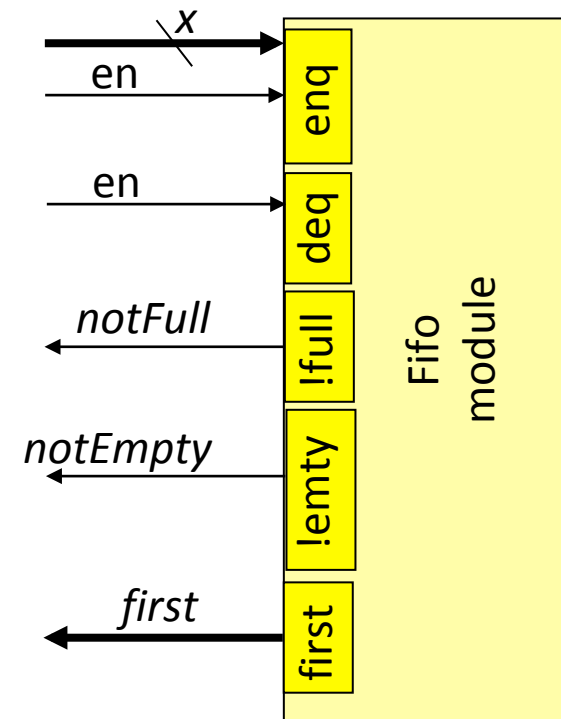
Modules

- 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)

FIFO 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

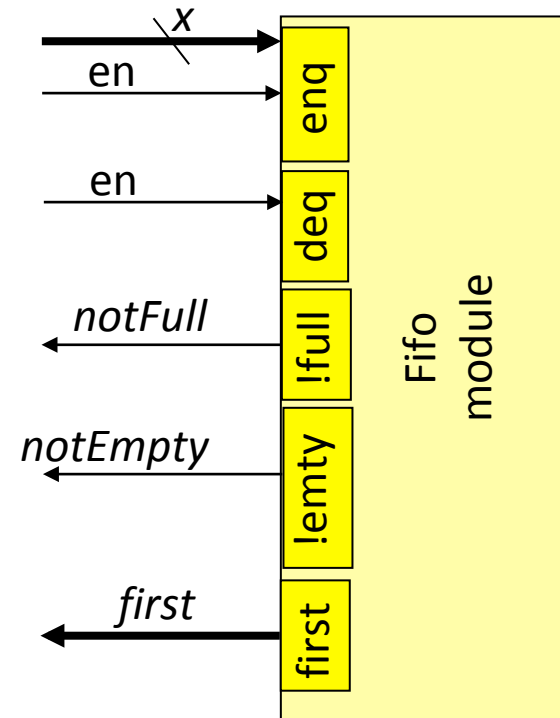


One-Element FIFO Implementation

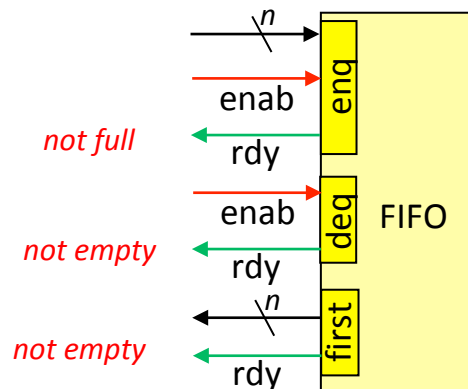
```

module mkCFFifo (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

```



FIFO Module: methods with guarded interfaces



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

- Every method has a *guard* (rdy wire); the value returned by a value method is meaningful only if its guard is true
- Every action method has an *enable signal* (en wire); an action method is invoked (en is set to true) only if the guard is true
- Guards make it possible to transfer the responsibility of the correct use of a method from the user to the compiler
- Guards are extraordinarily convenient for programming and also enhance modularity of the code

One-Element FIFO Implementation with guards

```
module mkCFFifo (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
```

