Sequential Circuits

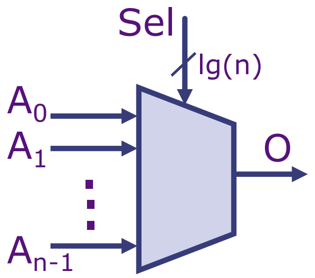
Circuits with state

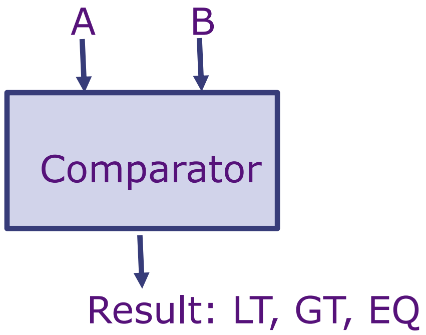
Silvina Hanono Wachman
Computer Science & Artificial Intelligence Lab
M.I.T.

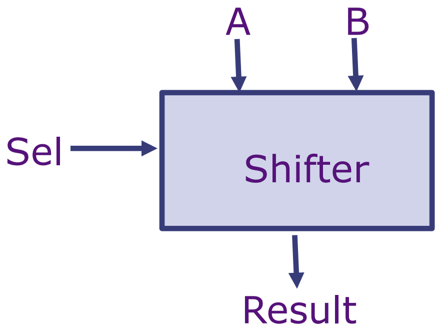
Reminders:

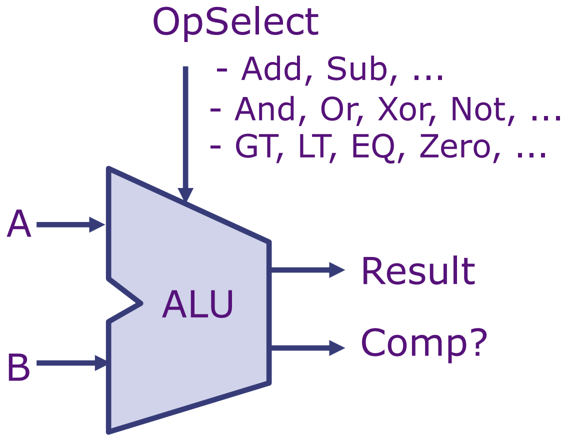
Quiz 1 tonight, 7:30-9:30PM
Last names A-S in 34-101
T-Z in 32-155
Good Luck!

Combinational circuits



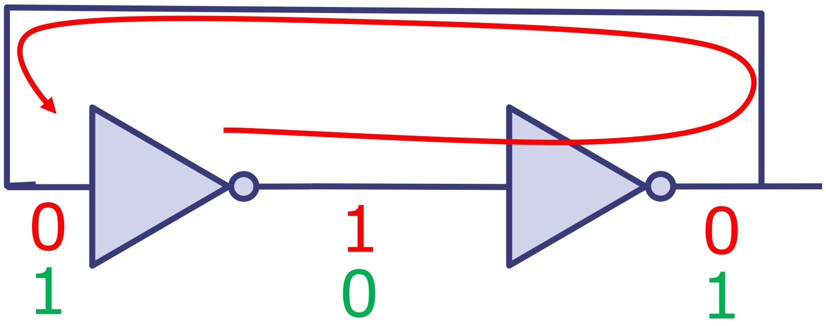






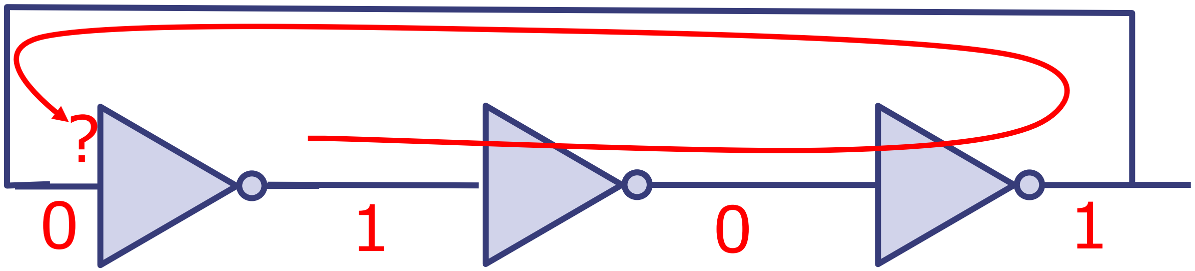
Combinational circuits have no cycles (feedback) or state elements

Simple circuits with feedback, i.e., a cycle



This circuit can hold a 0 or 1

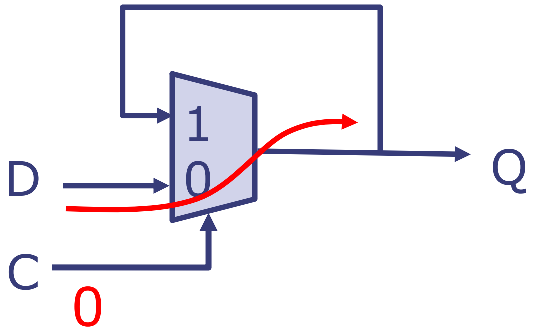
But how do we change its state?

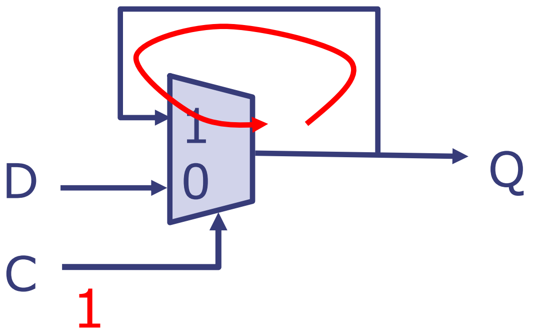


This circuit will oscillate between 0 and 1

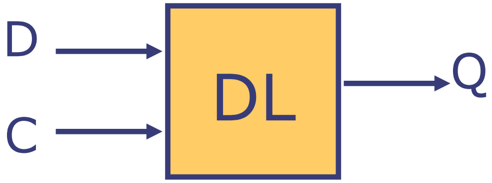
- Circuits with cycles can hold state
- Generally behavior is difficult to analyze and requires paying attention to propagation delays

D Latch: a famous circuit that can hold state





if C=0, the value of D passes to Q
if C=1, the value of Q holds



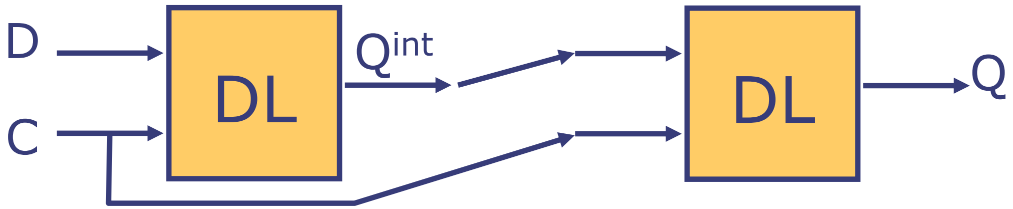
C	D	Q^t	Q^{t+1}
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

pass

hold

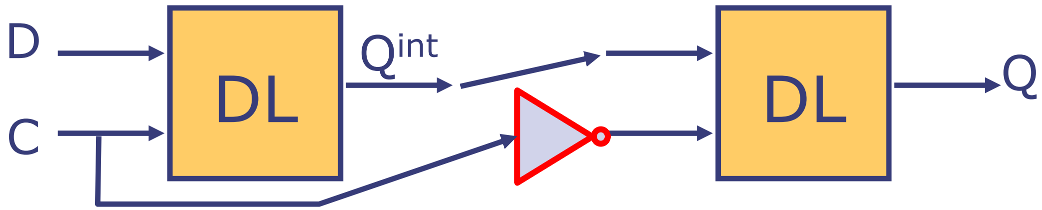
Let Q^t represent the current value held in DL;
Q^{t+1} represents the next value.

Building circuits with D latches



- If two latches are driven by the same C signal, they *pass* signals at the same time and *hold* signals at the same time.
- The composed latches look just like a single D latch (assuming signals aren’t changing too fast)

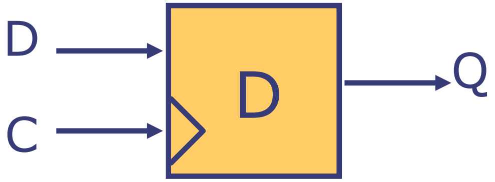
Building circuits with D latches
continued

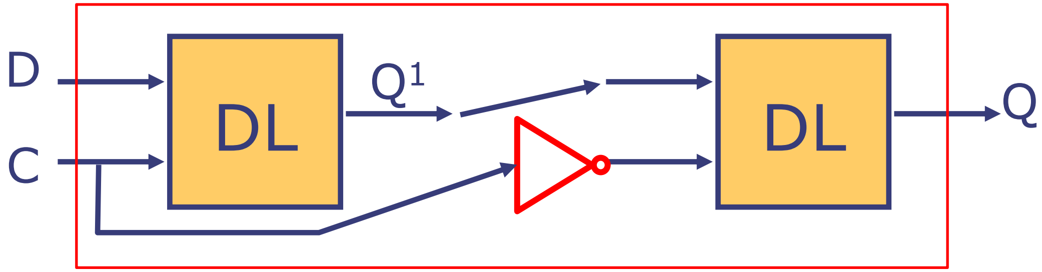


- If latches are driven by inverted C signals, one is always holding, and one is always passing
- How does this circuit behave?
  - When C = 0, Q holds its old value, but Q^{int} follows the input D
  - When C = 1, Q^{int} holds its old value, but Q follows Qint
  - Q doesn't change when C = 0 or C = 1, but it changes its value when C transitions from 0 to 1 (a *rising-edge* of C)

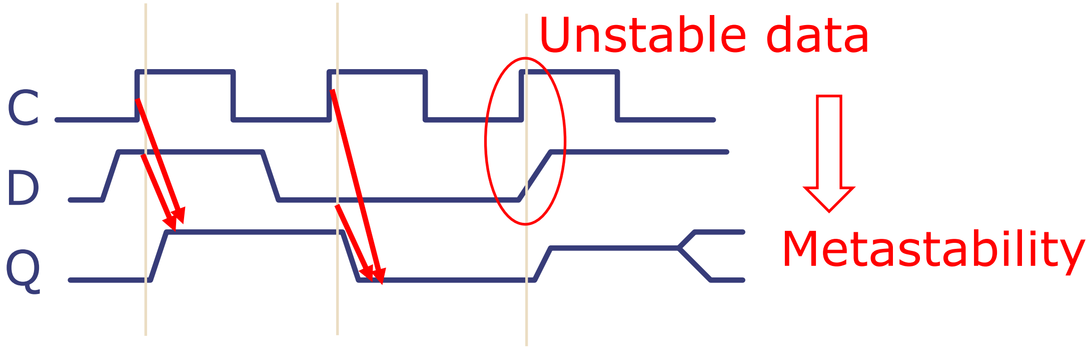
Edge-Triggered D flip-flop

A basic storage element





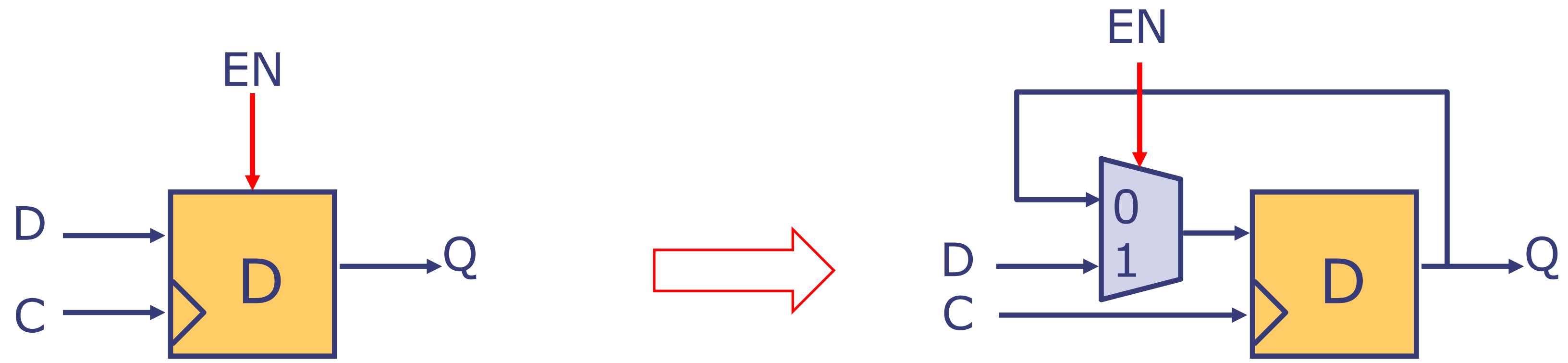
Suppose C changes periodically (called a *Clock* signal)

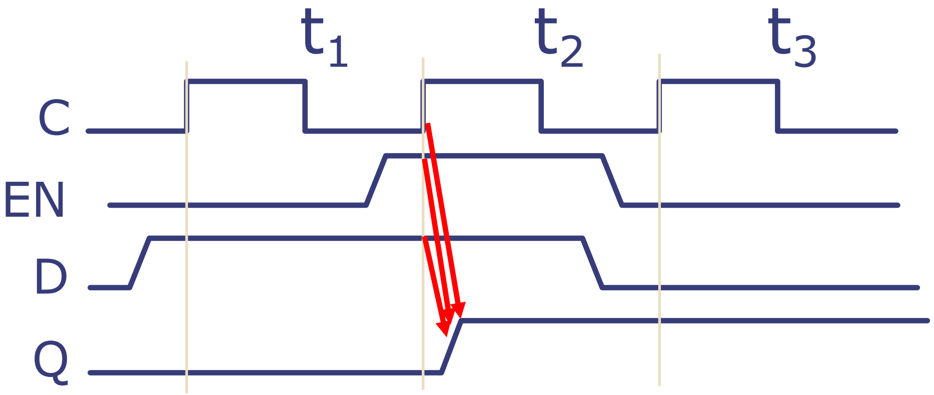


Data is sampled at the rising edge of the clock and must be stable at that time

D Flip-flop with Write Enable

The building block of Sequential Circuits





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

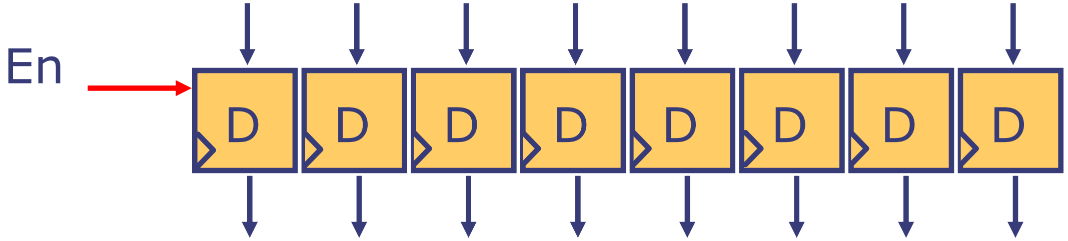
hold

copy input

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

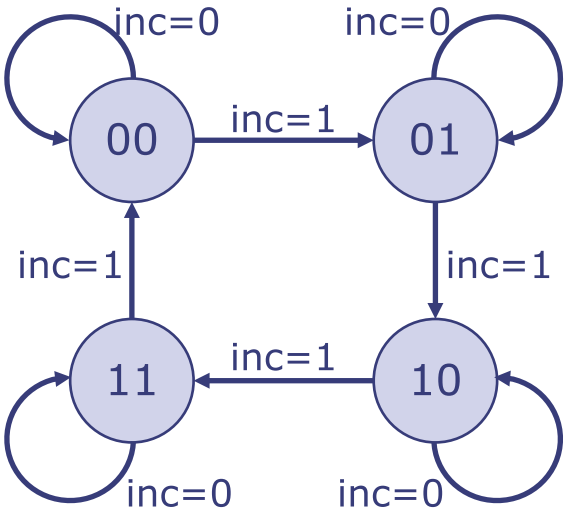
Clocked Sequential Circuits

- 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

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

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

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

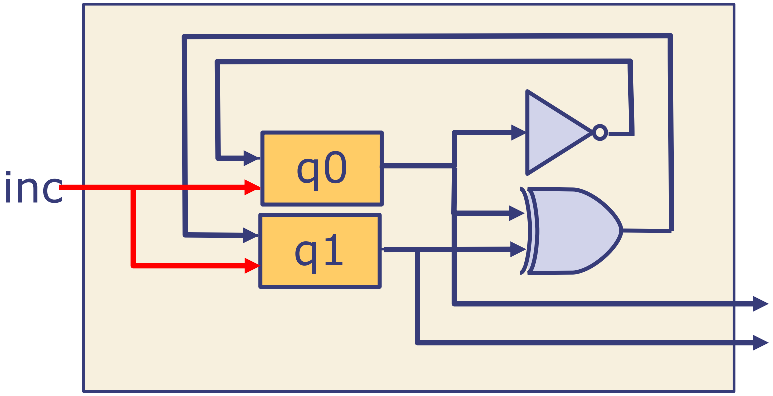
- Use two D flip-flops with enables to store q0 and q1
- Notice, the state of a flip-flop changes only when its is enable is true, and thus, the next-state input to a flip-flop matters only when its enable is true

```
q0en = inc;
q1en = inc
```

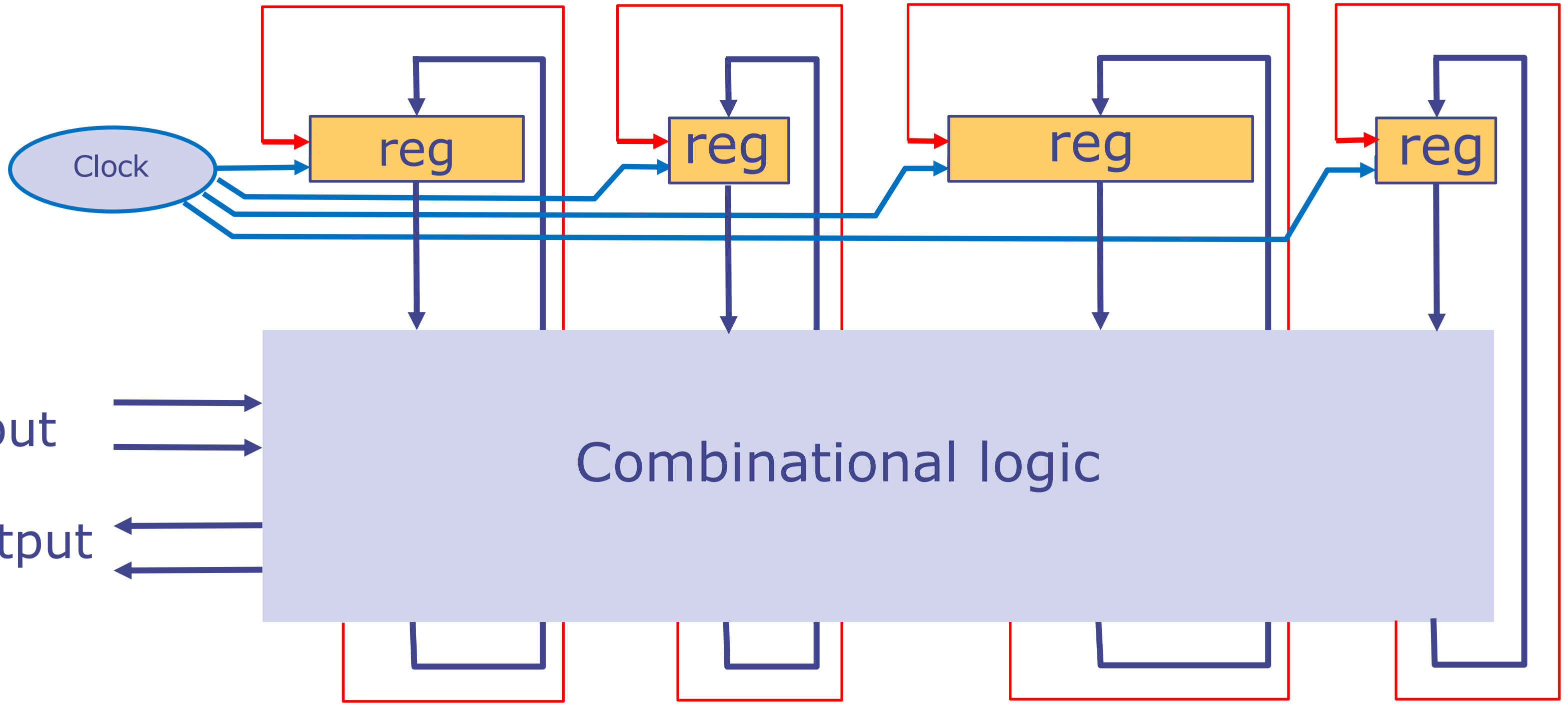
$$q0^{t+1} = (\sim inc \cdot q0^{t} + inc \cdot \sim q0^{t}) \cdot q0en$$
$$q1^{t+1} = (\sim inc \cdot q1^{t} + inc \cdot (q1^{t} \oplus q0^{t})) \cdot q1en$$

- simplifying the next-state equations

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

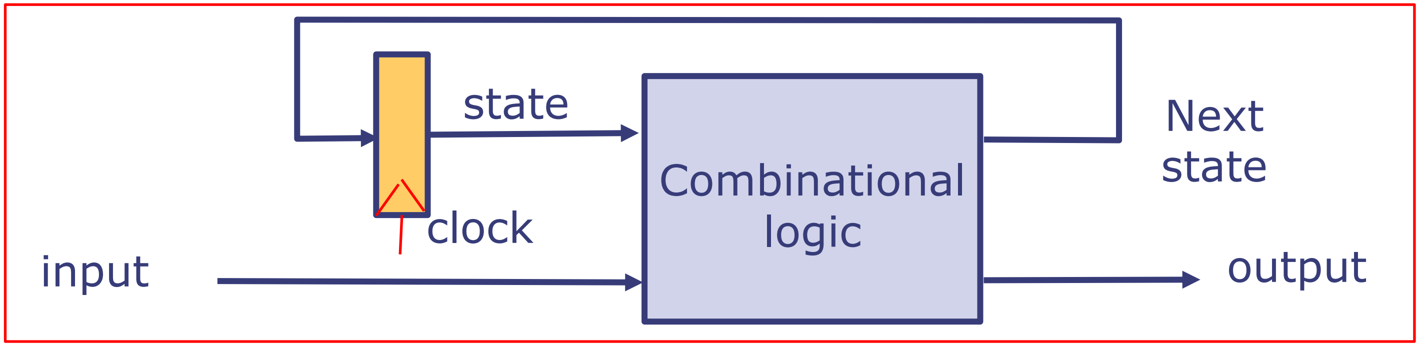


Synchronous Sequential Circuit using registers



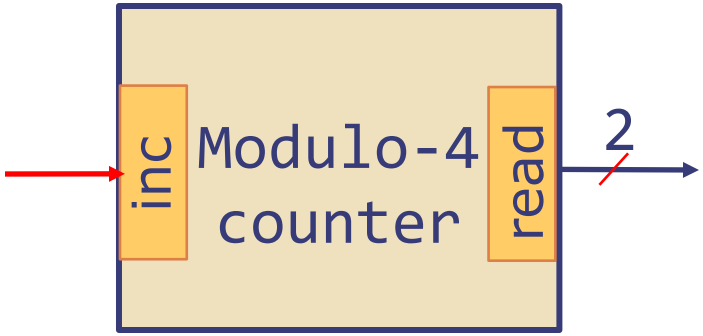
In our designs all registers are connected to the same clock input and therefore, we will not draw the clock wires in future

Finite State Machines (FSMs)



- Synchronous Sequential Circuits are a method to implement FSMs in hardware
- 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!

Sequential Circuit as a module with Interface

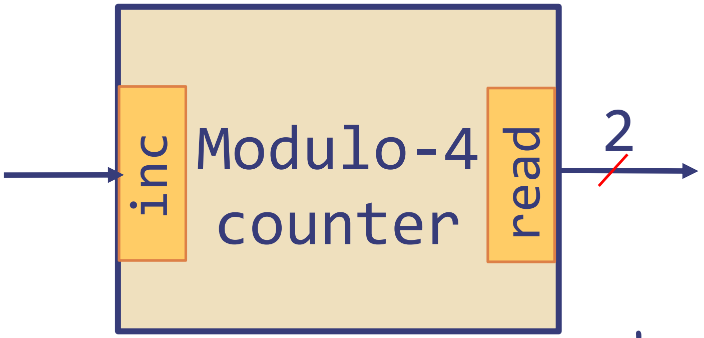


```
interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface
```

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

type

instantiate

State specification

Initial value

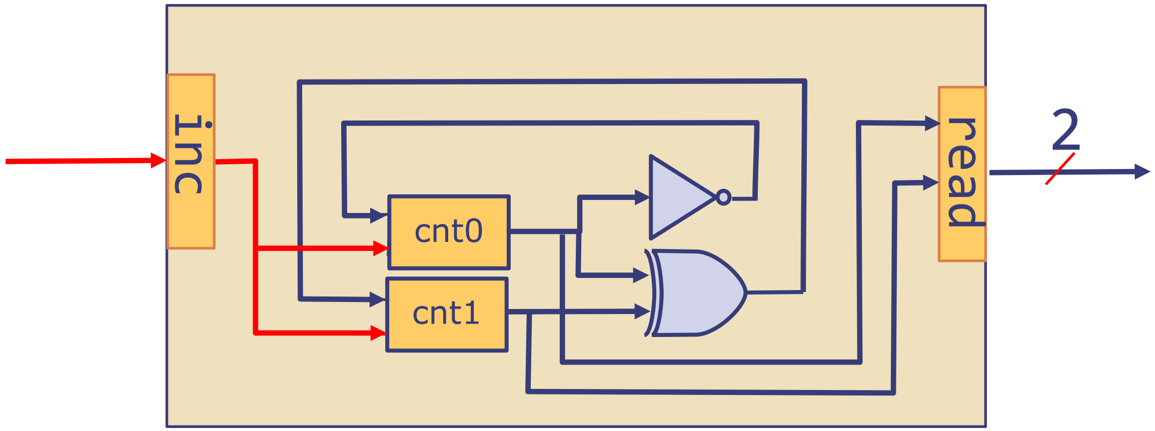
Register assignment

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



```
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
```

Summary

- Sequential circuits have state and are built using registers to hold state
- A register has an enable signal and its state can be changed only ifs enable is true
- A sequential circuit is represented by a *module* in Bluespec and has a well-defined set of interface methods to read and modify its state
  - A register is a primitive module type in Bluespec
- A module can be instantiated repeatedly to create objects, i.e., sequential circuits, of that type

Next more examples …

Take-home problems

- Write a module that can increment, decrement, or not change the value of a counter.
- Does this module need a busy method?

A method for computing GCD

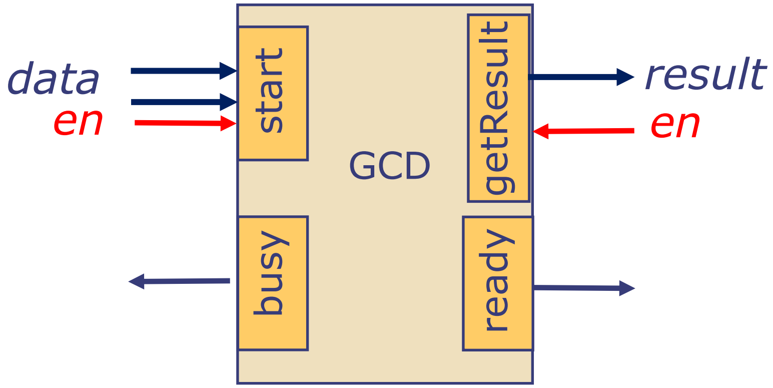
Euclid’s algorithm for computing the Greatest Common Divisor (GCD):

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

answer

```
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



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 Bool busy = busy_flag;
method Bool 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

parallel composition
of actions

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

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 there is a possibility of a double write

Stay tuned…