Registers and finite state machines

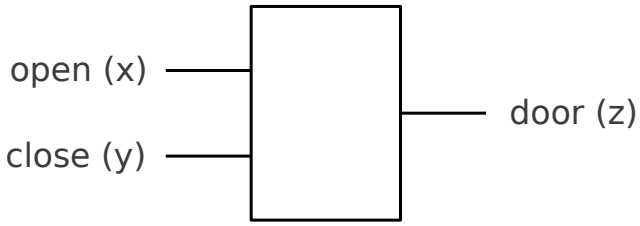
DAPA
E.T.S.I. Informática
Universidad de Sevilla
11/2012

Contents

- Introduction
- Flip-flops
- Registers
- Counters
- Finite state machines

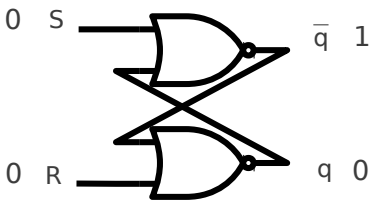
Introduction

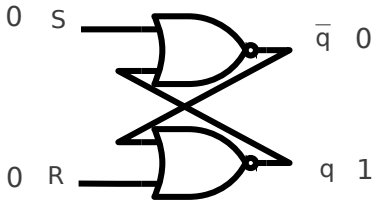
- Design a garage door control system with two push buttons (no switches) separated by a distance:
  - x: open the door
  - y: close the door



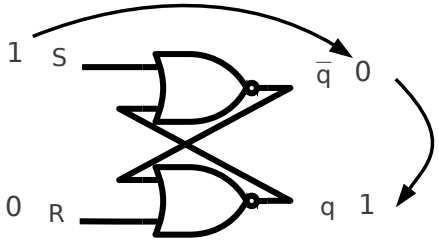
Asynchronous SR latch

- R=S=0 the state is preserved

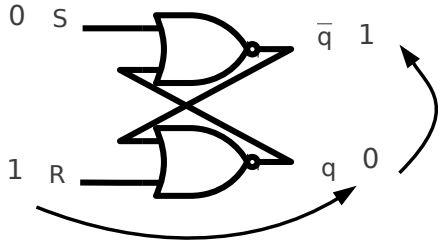




- S=1, R=0 change to 1 (set)

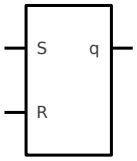


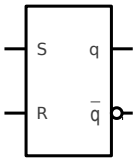
S=0, R=1 change to 0 (reset)



SR Latch. Formal description

Symbol



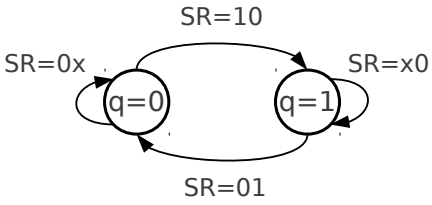


State table

q \ SR	00	01	11	10
0	0	0	-	1
1	1	0	-	1

Q

State diagram



Excitation table

q → Q	SR
0 → 0	0x
0→ 1	10
1 → 0	01
1 → 1	x0

Verilog

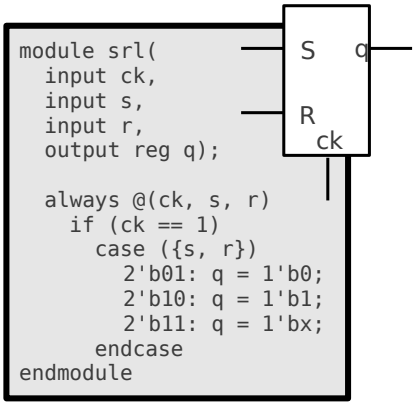
```
module sra(
  input s,
  input r,
  output reg q);

  always @(s, r)
    case ({s, r})
      2'b01: q = 1'b0;
      2'b10: q = 1'b1;
      2'b11: q = 1'bx;
    endcase
endmodule
```


Synchronous latches

- In real circuits with thousands or million latches, it is very useful to control the state change so that it happens in all devices at the same time.
- State change is “synchronized” to a “clock signal” (CK)
- Gated latches
  - State change is only allowed when CK is either high (1) or low (0).
- Edge-triggered latches (flip-flops)
  - State change is only allowed at the instant CK changes from 0 to 1 (positive edge) or from 1 to 0 (negative edge)
  - State change is more precisely determined.
  - Make more robust and easy to design circuits

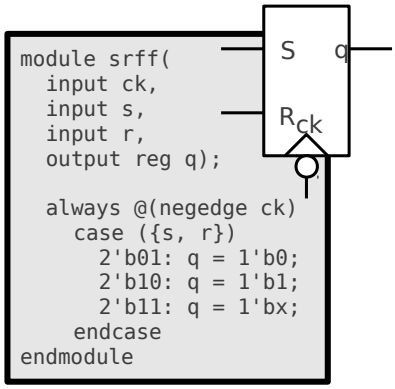
Synchronous latches

Gated latch



State change when ck=1

Flip-flop



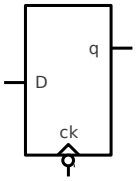
State changes when ck changes from 1 to 0

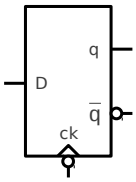
Other flip-flops

- SR
- JK
  - Similar to SR: J~S, K~R
  - Toggle (reverse) function for J=K=1
- D
  - A single input equal to the next state
  - Easy to use and implement
- T
  - A single input to toggle the state
  - Very useful in some special applications: counters

D flip-flop

Synbols





State table

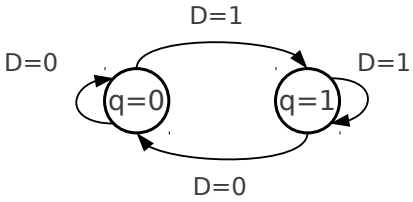
q \ D	0	1
0	0	1
1	0	1

Q

Excitation table

q → Q	D
0 → 0	0
0 → 1	1
1 → 0	0
1 → 1	1

State diagram

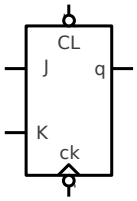


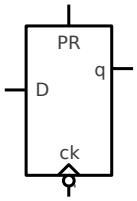
Verilog

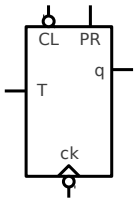
```
module dff(
  input ck,
  input d,
  output reg q);

  always @(negedge ck)
    q <= d;

endmodule
```


Asynchronous inputs

- Easy way to force a given state
  - CL (clear): set to 0
  - PR (preset): set to 1
- Immediate effect after activation:
  - Active low (0)
  - Active high (1)
- Higher priority than synchronous inputs
  - J, K, D, T, ...
- Solution to the problem of initiating the state in complex digital systems
  - Million of flip-flops
  - Need to start from a known state



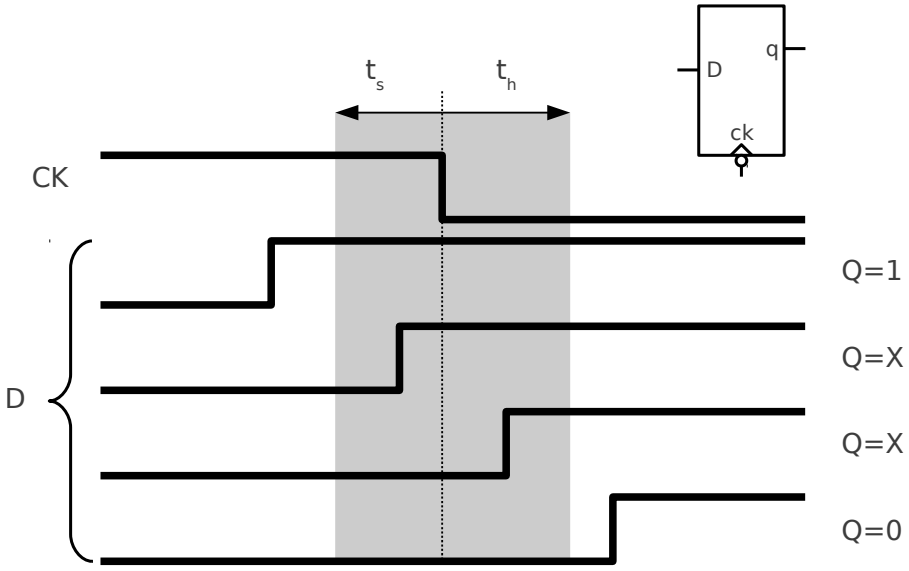




Timing

- Synchronous inputs should not change close to the active edge of the clock signal to avoid an unpredictable state change.
- **Set-up time** (ts)
  - Time **before** the active edge in which inputs should not change.
- **Hold time** (th)
  - Time after the active edge in which inputs should not change.

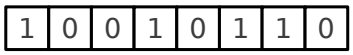
Timing



Contents

- Introduction
- Flip-flops
- Registers
- Counters
- Finite state machines

Registers



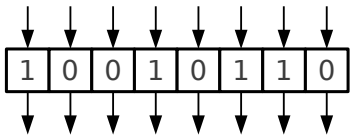
- n-bit storage element (n flip-flops)
  - Content is expressed from the data it represents: number (in hex or decimal), character, etc.
- Basic operations:
  - Write (load): stored data modification.
  - Read: access to the content of the register.

Registers. Classification

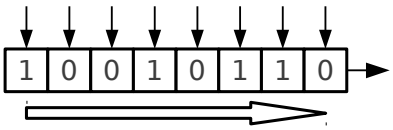
- Parallel input
  - All the bits can be written (loaded) at the same time (with the same clock event).
  - There is one load input signal for each stored bit.
- Serial input
  - Only one bit can be written at each clock cycle.
  - A single input signal for all the stored bits.
- Parallel output
  - All the bits can be read at the same time.
  - One output signal for each stored bit.
- Serial output
  - Only one bit can be read at each clock cycle.
  - A single output signal for all the stored bits.

Registers. Classification

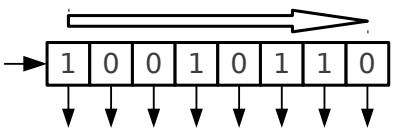
parallel-in/parallel-out



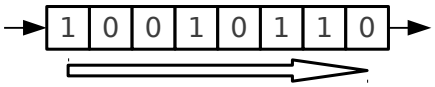
parallel-in/serial-out



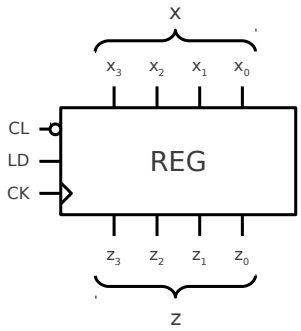
serial-in/parallel-out



serial-in/serial-out



Parallel-in/parallel-out register



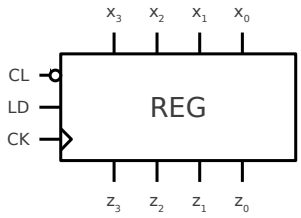
Verilog code

```
module reg(
    input ck,
    input cl,
    input ld,
    input [3:0] x,
    output [3:0] z
    );

    reg [3:0] q;

    always @(posedge ck, negedge cl)
        if (cl == 0)
            q <= 0;
        else if (ld == 1)
            q <= x;

    assign z = q;

endmodule
```

Operation table

CL, LD	Operation	Type
0x	q ← 0	async.
11	q ← x	sync.
10	q ← q	sync.

Parallel-in/parallel-out register

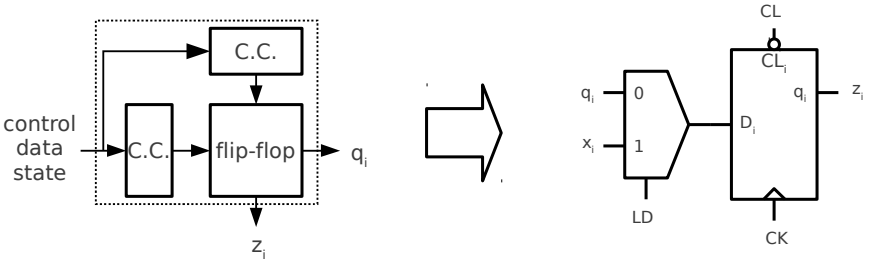


Asynchronous operation table

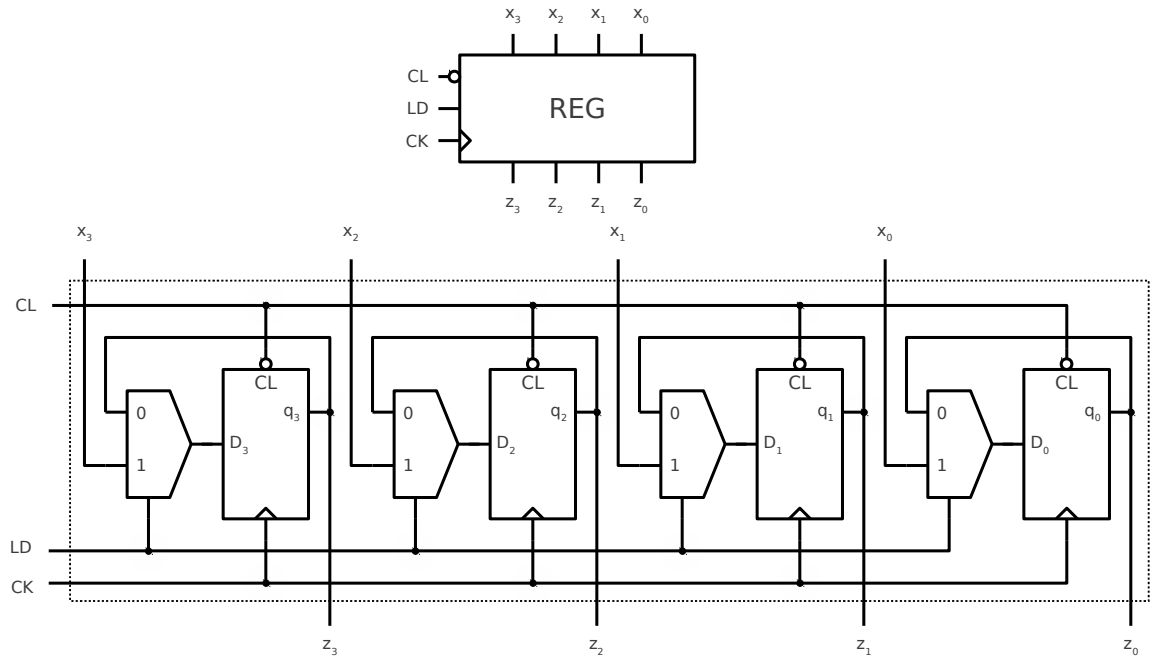
CL	Operation	Typ. stage	Excit.
0	q ← 0	$Q_i = 0$	$CL_i = 0$
1	q ← q	$Q_i = q_i$	$CL_i = 1$

Synchronous operation table

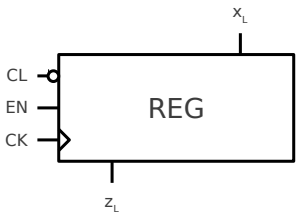
LD	Operation	Typ. stage	Excit.
1	q ← x	$Q_i = x_i$	$D_i = x_i$
0	q ← q	$Q_i = q_i$	$D_i = q_i$



Parallel-in/parallel-out register



Shift register



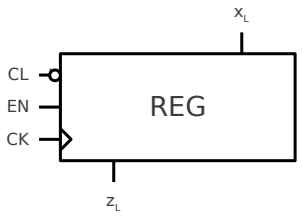
Operation table

CL, EN	Operation	Type
0x	q ← 0	async.
11	q ← SHL(q)	sync.
10	q ← q	sync.

Verilog code

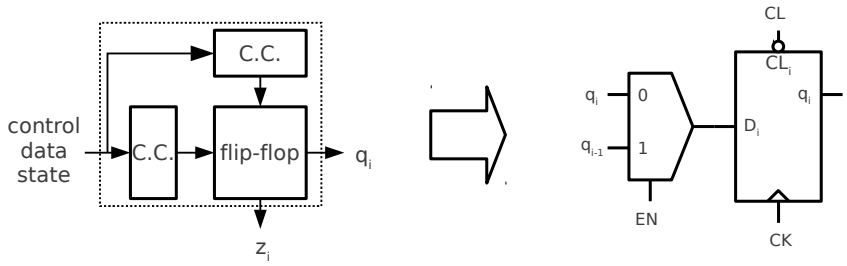
```
module reg_shl(
    input ck,
    input cl,
    input en,
    input xl,
    output zl
    );

    reg [3:0] q;

    always @(posedge ck, negedge cl)
        if (cl == 0)
            q <= 0;
        else if (en == 1)
            q <= {q[2:0], xl};

    assign zl = q[3];

endmodule
```

Shift register

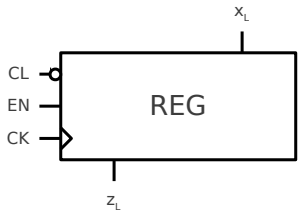


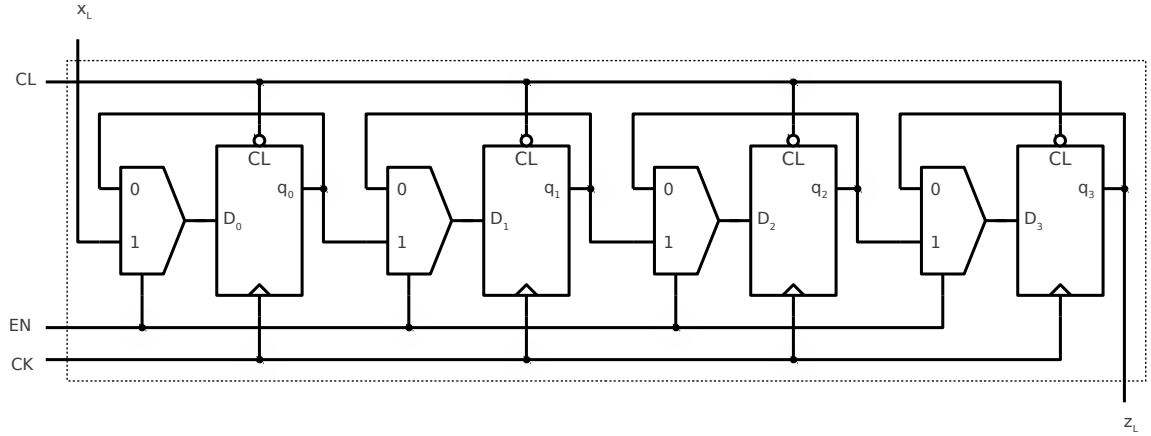
Synchronous operation table

EN	Operación	Et. típica	Et. 0	Ex. típ.	Ex. et. 0
1	q ← SHL(q)	$Q_i = q_{i-1}$	$Q_0 = x_L$	$D_i = q_{i-1}$	$D_0 = x_L$
0	q ← q	$Q_i = q_i$	$Q_0 = q_0$	$D_i = q_i$	$D_0 = q_0$



Shift register





Contents

- Introduction
- Flip-flops
- Registers
- Counters
- Finite state machines

Counters

- Introduction
- Registers
- Counters
  - Binary up counter modulus 2n
  - Count limiting
  - Down counter
  - Up/down counter
  - Non-binary counters
- Design with sequential subsystems

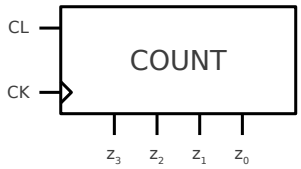
Counters

- Similar to the register: adds count operation
- Design
  - Modular design principles
  - Easier implementation with T or JK flip-flops (simplified count operation).
- Typical operations
  - Up counting
  - Down counting
  - Reset (clear)
  - Count state loading
- Typical output
  - Count state
  - Count end: counter in the last count value.

Binary modulus 2^n up counter

- Modulus
  - Number of counter states
- Binary
  - Consecutive base-2 numbers
- Modulus 2n
  - Counts from 0 to 2n-1 (n bits)
- Cyclic count
  - First count state follows last count state (overflow)

Binary modulus 2^n up counter



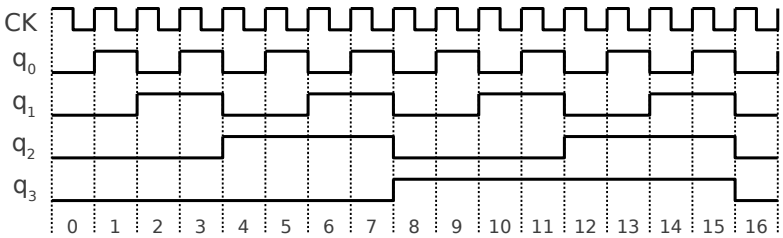
Verilog code

```
module count_mod16(
    input ck,
    input cl,
    output [3:0] z
    );

    reg [3:0] q;

    always @(posedge ck, posedge cl)
        if (cl == 1)
            q <= 0;
        else
            q <= q + 1;

    assign z = q;

endmodule
```

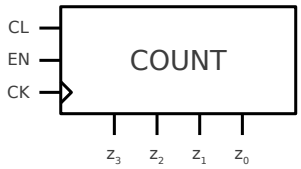
Operation table

CL	Operation	Type
1	$q \leftarrow 0$	async.
0	$q \leftarrow q+1\|_{\text{mod } 16}$	sync.

Binary modulus 2^n up counter
Count operation



Binary modulus 2^n up counter with "clear" and "enable" inputs



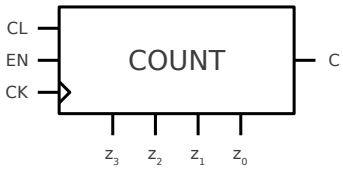
Verilog code

```
module count_mod16(
    input ck,
    input cl,
    input en,
    output [3:0] z
    );

    reg [3:0] q;

    always @(posedge ck)
        if (cl == 1)
            q <= 0;
        else if (en == 1)
            q <= q + 1;

    assign z = q;

endmodule
```

Operation table

CL, EN	Operation	Type
1x	$q \leftarrow 0$	sync.
01	$q \leftarrow q+1\|_{\text{mod } 16}$	sync.
00	$q \leftarrow q$	sync.

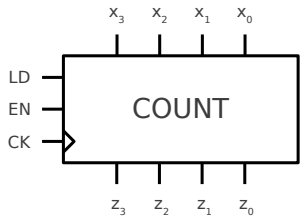
End-of-count output



Verilog code

```
module count_mod16(
    input ck,
    input cl,
    input en,
    output [3:0] z,
    output c
    );

    reg [3:0] q;

    always @(posedge ck)
        if (cl == 1)
            q <= 0;
        else if (en == 1)
            q <= q + 1;

    assign z = q;
    assign c = &q;

endmodule
```

Binary modulus 2^n up counter with "load" and "enable" inputs



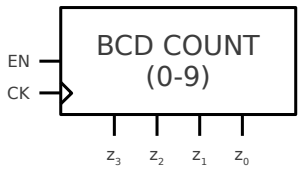
Verilog code

```
module count_mod16(
    input ck,
    input ld,
    input en,
    input [3:0] x,
    output [3:0] z
    );

    reg [3:0] q;

    always @(posedge ck)
        if (ld == 1)
            q <= x;
        else if (en == 1)
            q <= q + 1;

    assign z = q;

endmodule
```

Operation table

LD, EN	Operation	Tipo
1x	$q \leftarrow x$	sync.
01	$q \leftarrow q+1\|_{\text{mod } 16}$	sync.
00	$q \leftarrow q$	sync.

Count limiting. BCD counter



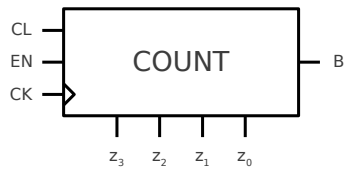
Verilog code

```
module count_mod10(
    input ck,
    input cl,
    input en,
    output [3:0] z,
    );

    reg [3:0] q;

    always @(posedge ck)
        if (cl == 1)
            q <= 0;
        else if (en == 1)
            if (q == 9)
                q <= 0;
            else
                q <= q + 1;

    assign z = q;

endmodule
```

Operation table

CL, EN	Operation	Type
1x	$q \leftarrow 0$	sync.
01	$q \leftarrow q+1\|_{\text{mod } 10}$	sync.
00	$q \leftarrow q$	sync.

Modulus 2^n down counter with "clear", "enable" and "borrow"



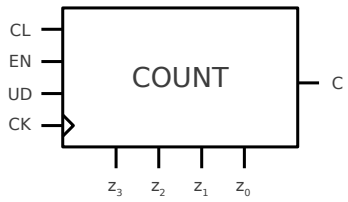
Verilog code

```
module count_mod16(
    input ck,
    input cl,
    input en,
    output [3:0] z,
    output b
    );

    reg [3:0] q;

    always @(posedge ck)
        if (cl == 1)
            q <= 0;
        else if (en == 1)
            q <= q - 1;

    assign z = q;
    assign b = &~q;

endmodule
```

Operation table

CL, EN	Operation	Type
1x	$q \leftarrow 0$	sync.
01	$q \leftarrow q\text{-}1\|_{\text{mod } 16}$	sync.
00	$q \leftarrow q$	sync.

Up/down counter



Operation table

CL, EN, UD	Operation	Type
1xx	$q \leftarrow 0$	async.
00x	$q \leftarrow q$	sync.
010	$q \leftarrow q+1\|_{\text{mod } 16}$	sync.
011	$q \leftarrow q\text{-}1\|_{\text{mod } 16}$	sync.

Up/down counter

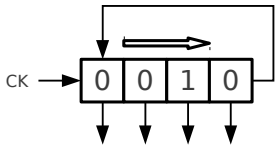
Verilog code

```
module rev_counter1(
    input ck,
    input cl, input en, input ud,
    output [3:0] z, output c
    );

    reg [3:0] q;

    always @(posedge ck, posedge cl)
    begin
        if (cl == 1)
            q <= 0;
        else if (en == 1)
            if (ud == 0)
                q <= q + 1;
            else
                q <= q - 1;
    end

    assign z = q;
    assign c = ud ? ~(|q) : &q;

endmodule
```

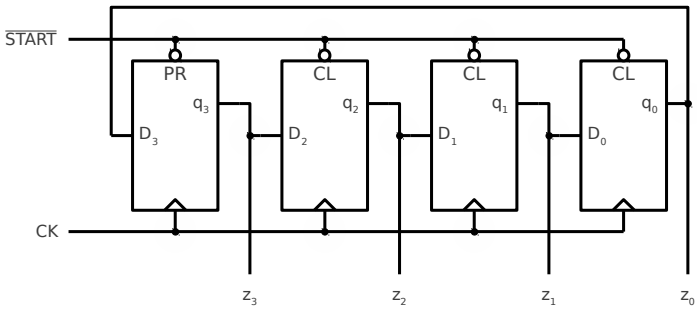
Verilog code

```
module rev_counter2(
    input ck,
    input cl, input en, input ud,
    output [3:0] z, output c
    );

    reg [3:0] q; reg c;

    always @(posedge ck, posedge cl)
    begin
        if (cl)
            q <= 0;
        else if (en)
            if (!ud)
                q <= q + 1;
            else
                q <= q - 1;
    end

    always @*
        if (ud)  c = ~(|q);
        else     c = &q;

    assign z = q;

endmodule
```


Ring counter

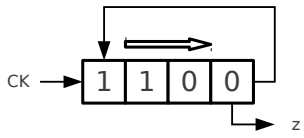


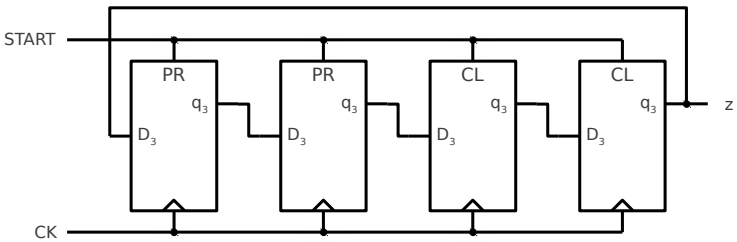
z_3 z_2 z_1 z_0
1000
0100
0010
0001
1000
...



Sequence generator

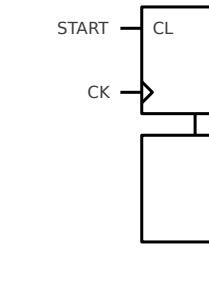
- “0011” sequence generator with shift register.

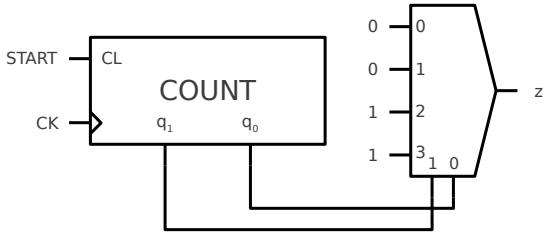




Sequence generator

- With counter and C.C.

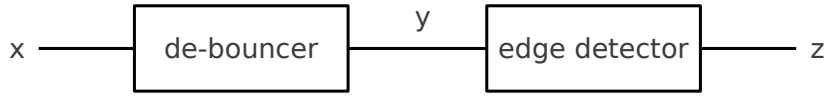


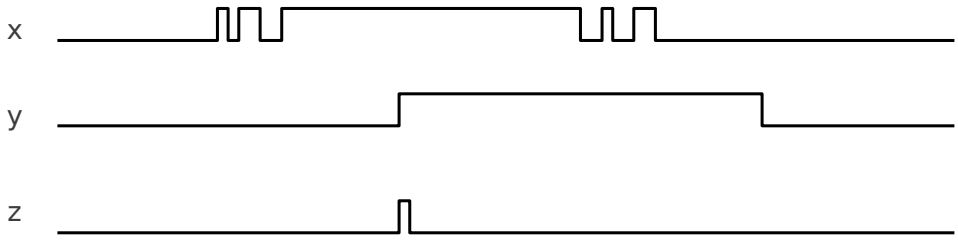


Verilog code

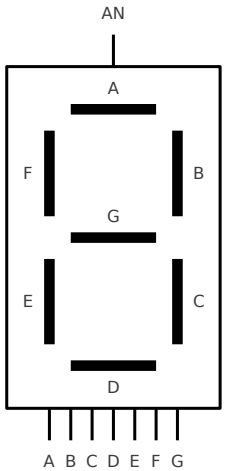
```
module seq_gen(
    input ck,
    input start,
    output z
    );

    reg [1:0] q; reg z;

    always @(posedge ck)
        if (start == 1)
            q <= 0;
        else
            q <= q + 1;

    case (q)
      2'h0: z = 1'b0;
      2'h1: z = 1'b0;
      2'h2: z = 1'b1;
      2'h3: z = 1'b1;
    endcase

endmodule
```

Example: de-bouncer and edge detector

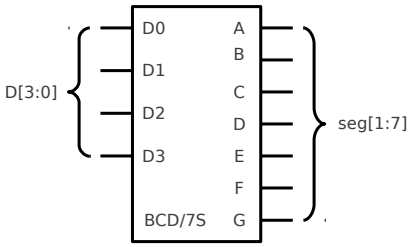




Example: 7S controller

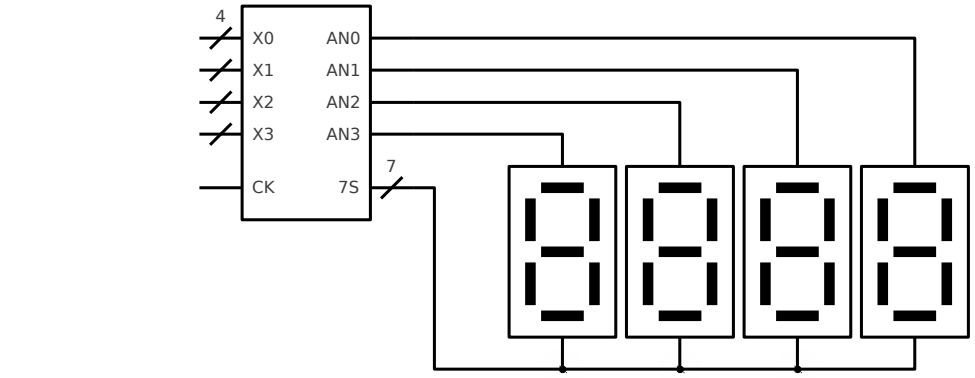


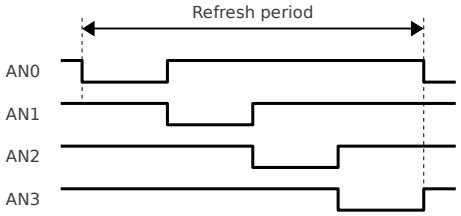
AN should be '0' for the device to work.



$D_3D_2D_1D_0$	D	seg[1:7] ABCDEFG
0000	0	0000001
0001	1	1001111
0011	2	0010010
0010	3	0000110
0110	4	1001100
0111	5	0100100
0101	6	0100000
0100	7	0001111
1100	8	0000000
1101	9	0001100

Example: 7S controller



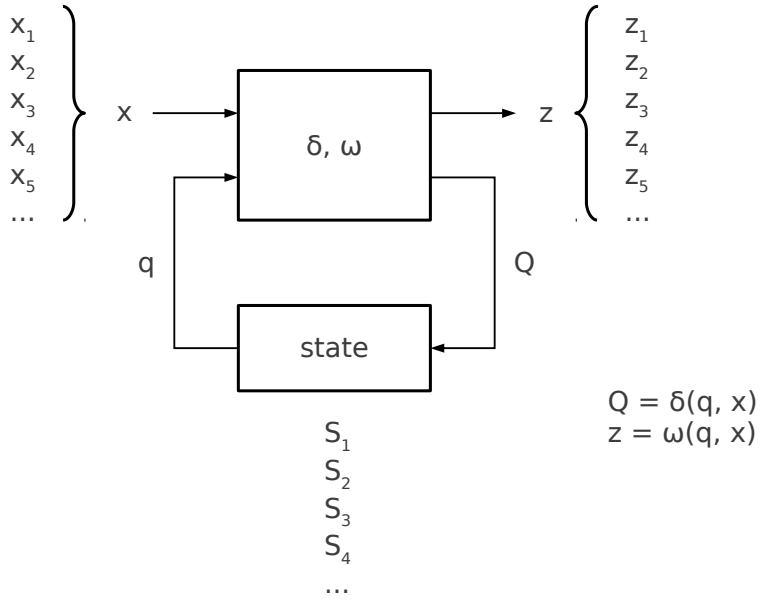


CK: 50MHz
Refresh frequency: 100Hz - 1KHz

Contents

- Introduction
- Flip-flops
- Registers
- Counters
- Finite state machines

Finite State Machines (FSM)



$Q = \delta(q, x)$
$z = \omega(q, x)$

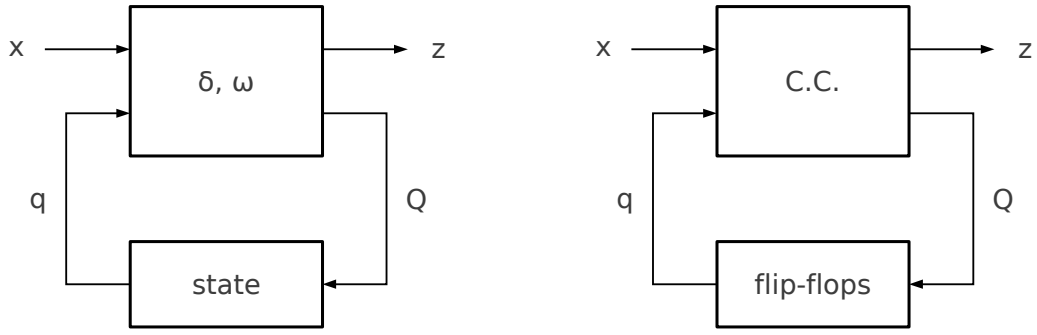
Finite State Machine properties

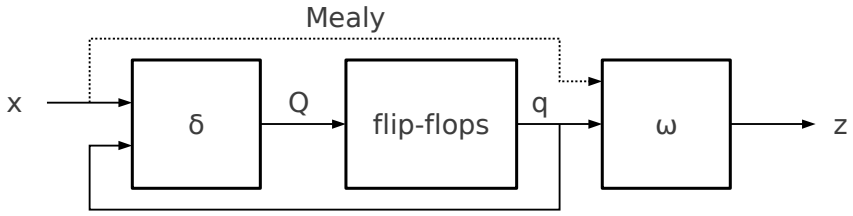
- Starting at the same state, deterministic FSM's always generate the same output sequence for the same input sequence.
- Two FSM's are equivalent if they generate the same output sequence for the same input sequence.
- FSM's can be optimized: an equivalent FSM with a reduced number of states.
- The state of the machine changes depending on the input sequence so the state represents the whole set of old input symbols (history of the machine)
- FSM's may not be completely specified: a next state may not be defined for a given current state and input value.

Synchronous Sequential Circuits (SSC)

- FSM's are an excellent tool to model digital circuits with memory.
- Digital circuits with memory elements (latches) are an excellent technology to implement FSM's:
  - Inputs/outputs: digital signals of one or more bits.
  - State: n-bit word stored in latches
  - Next state function: combinational functions that are applied to the inputs of the latches.
  - Output function: combinational function.
- Synchronous sequential circuits are digital circuits that implement finite state machines by using combinational circuits and latches.
- For practical reasons, the state change is controlled by a clock signal: edge-triggered flip-flops are normally used.

Synchronous Sequential Circuits (SSC)





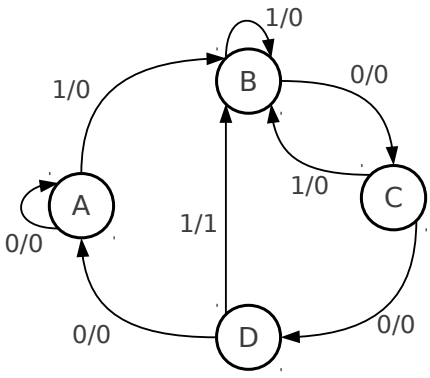
Formal representations

- State diagrams
- State tables

Example 1: barrier control
x=1: open
y=1: close
z: door control (0: close, 1: open)

Example 2: single-input barrier control
x=1 one cycle: open
x=1 two cycles: close
z: door control (0: close, 1: open)

State diagram. Mealy

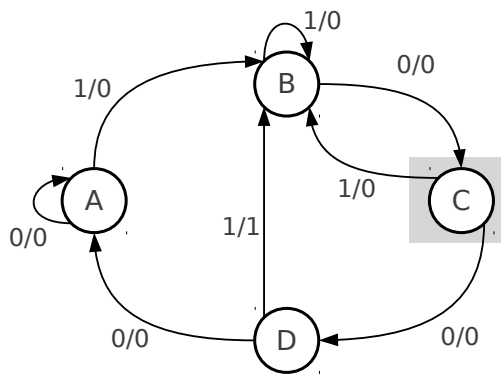


- Nodes
  - Represent states. State names are more or less intuitive. E.g. {A, B, C, ...}, {S0, S1, S2, ...}, {wait, start, receiving, ...}
- Arcs
  - Represent possible state transitions from every state (S).
  - Named as x/z, where:
    - x: input value that triggers the transition from state S.
    - z: output value of the machine when in state S and input is x.

State table. Mealy

- Double input table (rows and columns) with information equivalent to the states diagram.
  - Rows: possible states.
  - Columns: possible input values.
  - In each cell: corresponding next state and output.
- Each node in the diagram and the arcs starting at that node correspond to a row in the states table.
- Converting a states diagram to a states table and vice-versa is trivial.

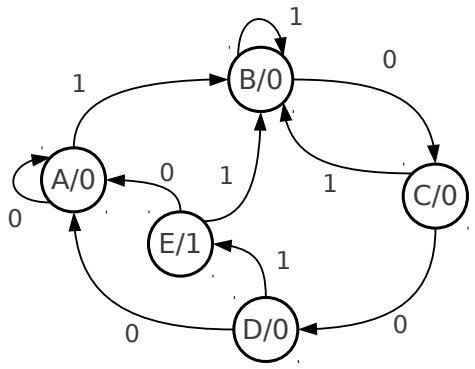
State table. Mealy



S \ X	0	1
A	A,0	B,0
B	C,0	A,0
C	D,0	B,0
D	A,0	B,1

Q,z

State diagram. Moore

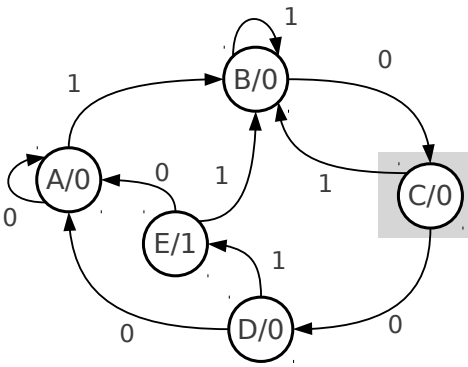


- Nodes
  - Represent states. State names are more or less intuitive. E.g. {A, B, C, ...}, {S0, S1, S2, ...}, {wait, start, receiving, ...}
  - Each node includes the output value corresponding to each state.
- Arcs
  - Represent possible state transitions from every state (S).
  - Named as x: input value that triggers the transition from state S.

State table. Moore

- Double input table (rows and columns) with information equivalent to the states diagram.
  - Rows: possible states.
  - Columns: possible input values.
  - Output associated to every state in last column. Optionally, output value in each cell like Mealy. Same output for every input value.
- Each node in the diagram and the arcs starting at that node correspond to a row in the states table.
- Converting a states diagram to a states table and vice-versa is trivial.

State table. Moore



S \ x	0	1	z
A	A	B	0
B	C	A	0
C	D	B	0
D	A	E	0
E	A	B	1

Q

Applications of the SSC

- Sequence detectors
  - Output activates only when a given sequence of symbols arrive to the input.
- Sequence generators
  - The output generates a fixed sequence of symbols, or a variable one depending on the input.
- Control units
  - The inputs modify the state and the state defines an actuation over an external system: barrier control, temperature control, presence control, liquid level control, etc.
- Sequential processing
  - The output sequence is the result of applying some operation to the input sequence: parity calculation, sequential arithmetic operations, sequential coding/decoding, etc.

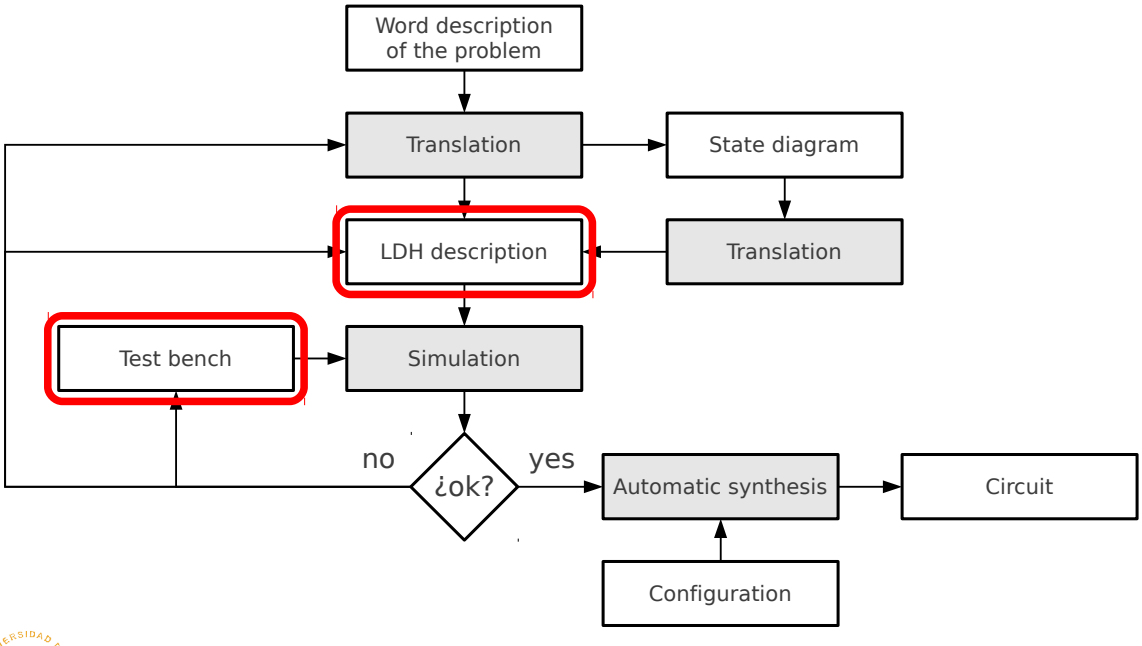
FSM design objective

- Objective
  - Define a FSM that solves a given problem.
  - Implement the FSM using a synchronous sequential circuits.
- Cost
  - Various cost criteria drive the design process
    - Minimizing the number of memory elements
    - Minimizing the number of devices
    - Operation frequency
    - Energy consumption
    - ...
  - Need to compromise different criteria

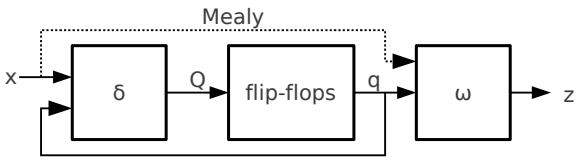
Design processes

- Hand process
  - Can be done with paper and pencil.
  - Starts with a description of the problem using a state diagram or a state table.
  - The states table is transformed in different steps to lead to a circuit representation.
- Design process using Computer Aided Design (CAD) tools
  - The problem is translated to a formal description using a hardware description language.
  - Simulation tools are used to check that the operation of the described system is correct.
  - Automatic synthesis tools are used to implement the actual circuit.

Design process using CAD tools



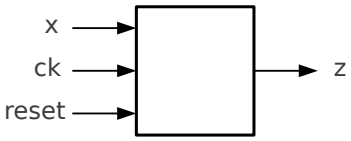
Verilog FSM descriptions



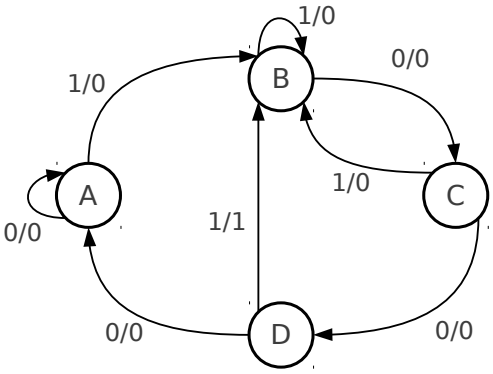
- Three processes
  - State change: represents the flip-flop block.
  - Next state calculation: excitation equations (δ)
  - Output calculation: output equations (ω).
- Only the state change process is sequential (includes memory elements)

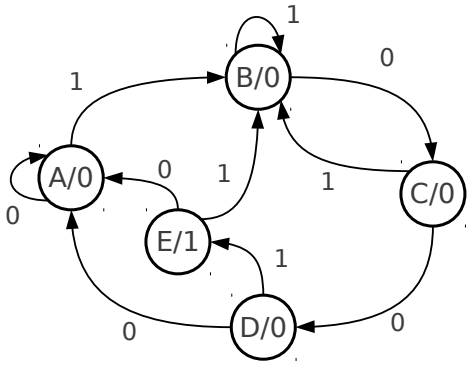
```
// State change process (sequential)
always @(posedge ck, posedge reset)
    if (reset)
        state <= A;
    else
        state <= next_state;

// Next state calculation process
// (combinational)
always @* begin
    case (state)
    A:
        next_state = . . .;
    B:
        next_state = . . .;
    . . .
    endcase
end

// Output calculation process
// (combinational)
always @* begin
    z = . . .;
end
```

Verilog FSM. Example



See Verilog course





Other FSM design styles