Unit 6. Sequential subsystems

Digital Electronic Circuits
E.T.S.I. Informática
Universidad de Sevilla

Jorge Juan-Chico <jjchico@dte.us.es> 2010-2021

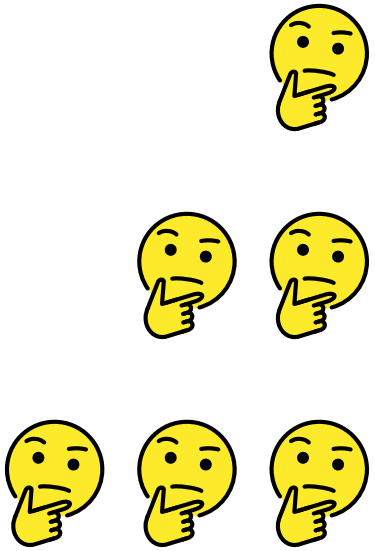
Contents

- Introduction
- Registers
- Counters
- Design with sequential subsystems

Recommended readings and exercises

- Theory reference
  - LaMeres, 7.5: Counters from a FSM perspective.
- Verilog modeling reference
  - verilog-course.v, unit 7.
  - LaMeres: 9.4, 9.5.
- Exercises from course's collection 6 (in Spanish)
  - Register design: 4, 19.
  - Counter design: 2, 10.
  - Counter design and count limit: 20
  - Sequence generator: 5a, 5b.
  - Counters based on other counters: 6.
  - Digital clock: 8.

Examples difficulty level



Easy

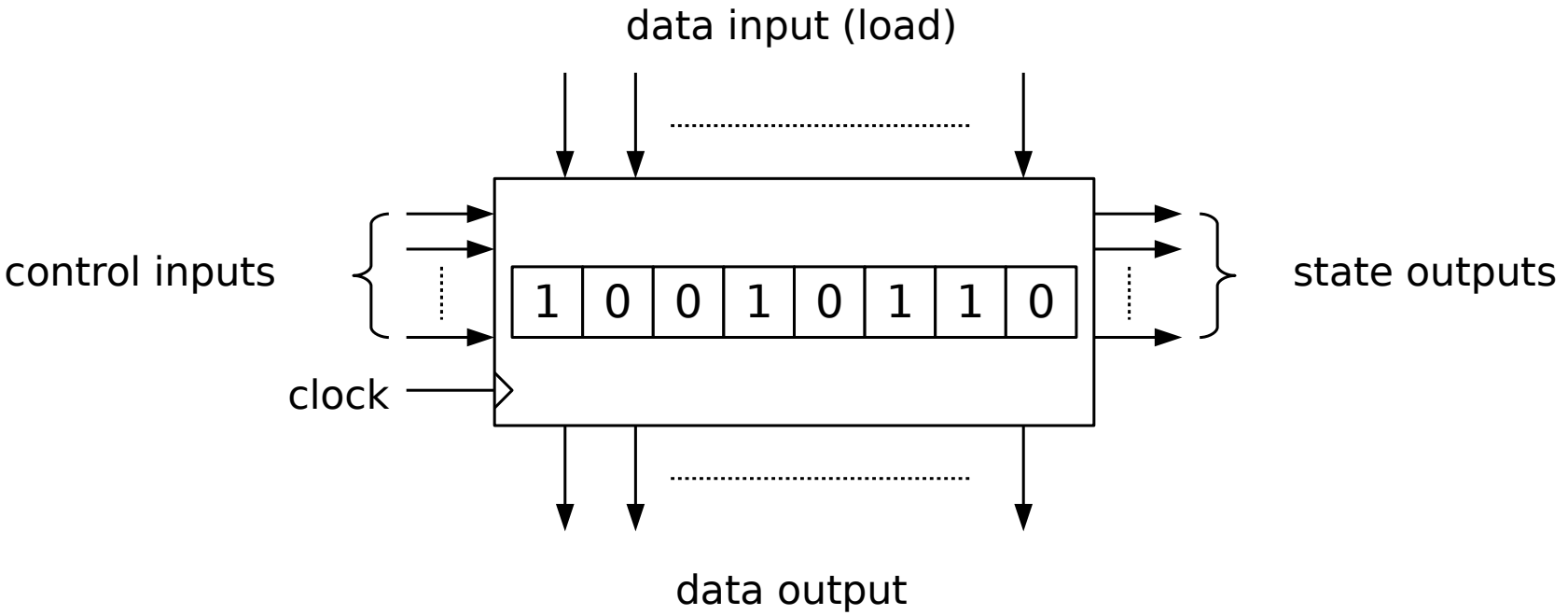
Moderate

Difficult

Introduction

- Sequential subsystem
  - Sequential circuit with n flip-flops that work together to do some task or set of tasks.
  - The operation is related to the n-bit data stored, not to individual bits.
  - The implemented functionality is general enough to find applications in multiple design problems.
- Basic types of sequential subsystems
  - Registers: store a n-bit word for its later use, with the possibility to appy transformation on the stored data.
  - Counters: specialized register that perform counting operations (increment, decrement, etc.) on the stored data.

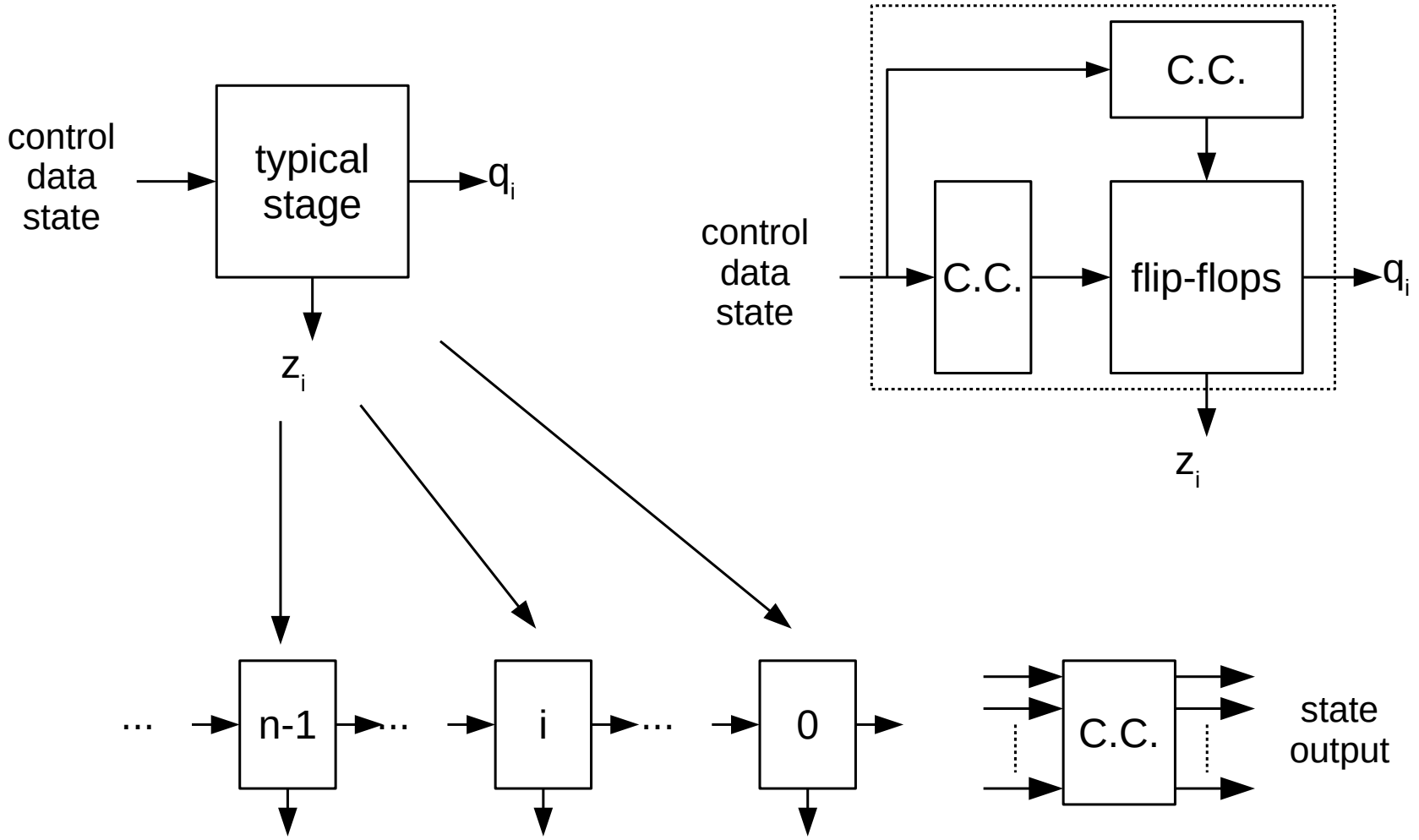
Introduction



Introduction

- Control inputs
  - Select operation to perform.
  - Synchronous operations: take place with the active clock event.
  - Asynchronous operations: take place right after control signal activation.
  - Typical control signals:
    - CL: clear
    - EN: enable
    - LD: data load
- Data inputs: provide the data to be loaded
- Data outputs: give access to the store data
- State outputs: give information about the stored data or system state
  - Eg.: content is zero, count end, etc.

Modular design



Modular design

- Modular structure
  - All the bits in the subsystem do a similar operation.
  - The operation and complexity of the system does not depend on the number of bits.
- Sequential subsystem design
  - Although a sequential subsystem can be described by a FSM it is not normally a good design methodology: big number of similar states.
  - Modular design approach:
    - Design a typical stage (1 bit).
    - Replicate the typical stage for the n bits in the system.
    - Include special cases: border bits and global signals.
    - The design complexity does not depend on the number of bits.

RT (Register Transfer) notation

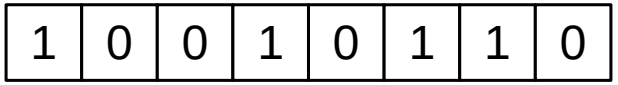
- Used to represent multi-bit data assignment (like the contents of registers and counters).

Type	Operators	Function	Examples	Verilog ex.
Sequential assignment	A ← <expr>	The value of <expr> is stored in register A. Storage takes place with the active clock edge. Registers in <expr> represent the current value.	Q ← Q + A	q <= q + a;
Combinational assignment	A = <expr>	The value of A at each moment is obtained by evaluating <expr>.	A = B + C	assign a = b + c;
Arithmetic	+, -, *, /, ...	Arithmetic operations	A = B * C CNT ← CNT+1	a = b * c; cnt <= cnt + 1;
Logic	&, |, ^, $\overline{\ }$, ...	Bitwise logic operations	A = B&$\overline{C}$	a = b & ~c;
Shift	SHL(A,b) SHR(A,b)	A is shifted one bit. The new bit is taken from 'b'.	Q ← SHL(Q,0) Z = SHR(X,X_R)	q <= q << 1;
Bit selection	A[i], A_i	i-th bit in A is selected.	Z = B_3	z = b[3];
Bit range selection	A[i..j], $A_{i..j}$	Bits from i-th to j-th are selected.	B = $Q_{7..4}$	b = q[7:4];
Chain	{A,B}	The bits of two words are chained together to form a larger word.	A ← {B,C} D = {$A_{3..0}$,$B_{7..4}$}	a <= {b,c}; d = {a[3:0],b[7:4]};

Contents

- Introduction
- Registers
  - Register classification
  - Parallel-in, parallel-out register
  - Shift register
  - Universal register
- Counters
- Design with sequential subsystems

Registers



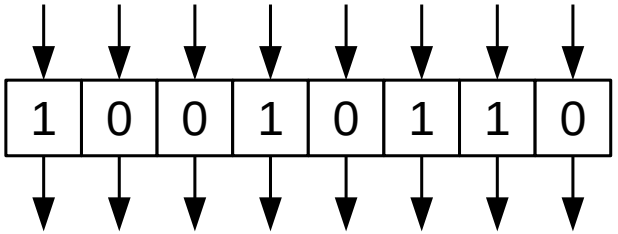
- n-bit storage element (n flip-flops)
  - Content is usually expressed by the data it represents and not by individual bits: number, character, etc.
  - Typically designed using D flip-flops: easier design.
  - All the flip-flops in the register share the same clock and asynchronous signals.
- Basic operations:
  - Write (load): stored data modification.
  - Read: access to the content of the register.

Registers. Classification

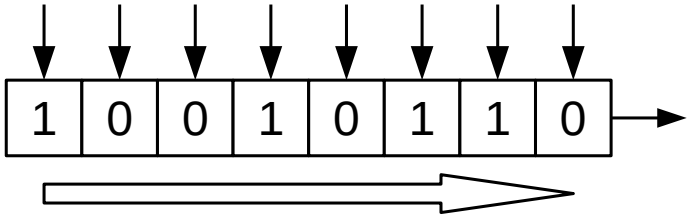
- Parallel input
  - All the bits can be written (loaded) at the same time (with the same clock event).
  - There is one load input signal for each stored bit.
- Serial input
  - Only one bit is loaded at each clock cycle.
  - A single input signal for all the stored bits (needs a bit shift operation).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

Registers. Classification

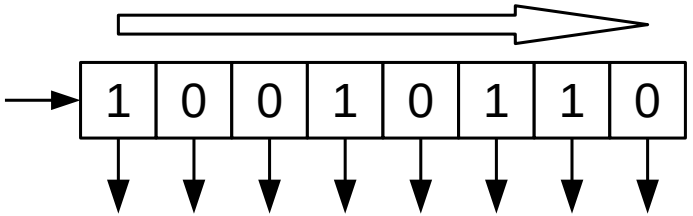
parallel-in/parallel-out



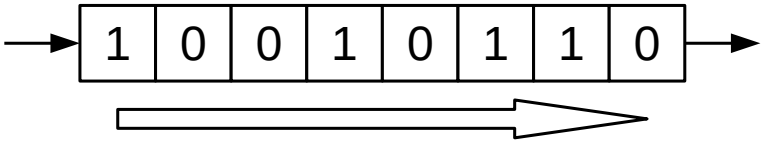
parallel-in/serial-out



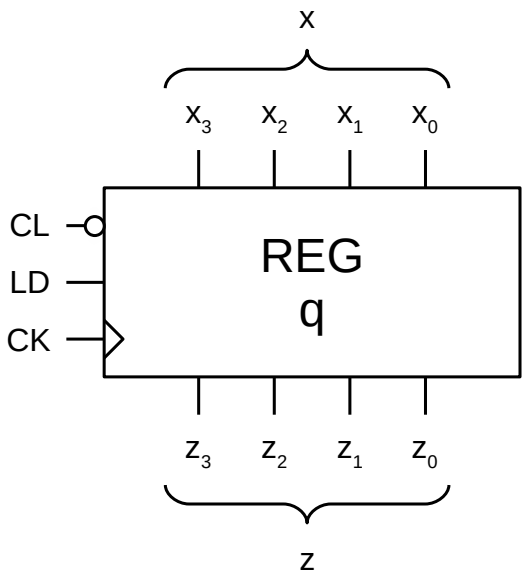
serial-in/parallel-out



serial-in/serial-out



Parallel-in/parallel-out register



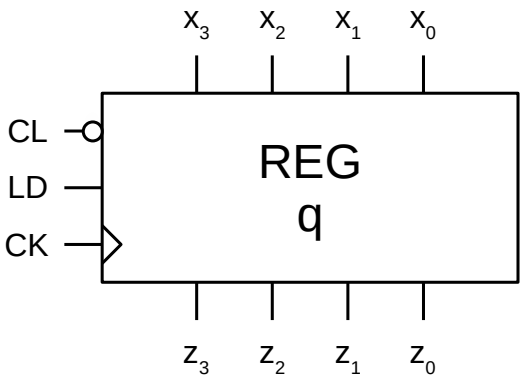
Verilog code

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

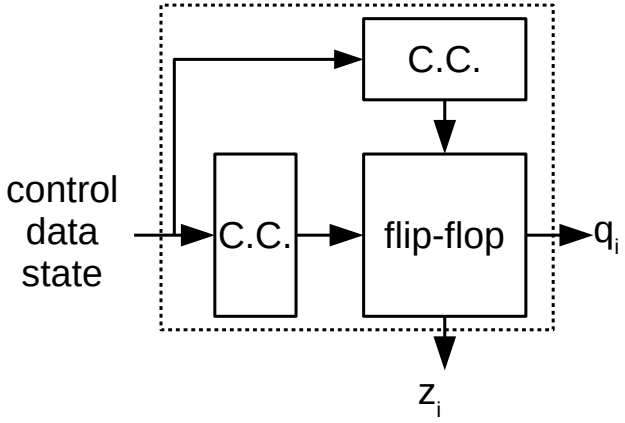


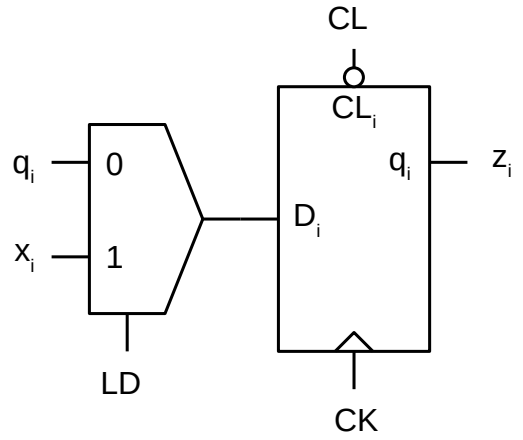
Asynchronous operation table

CL	Operation	Typ. stage	Excitation
0	$q \leftarrow 0$	$q_i \leftarrow 0$	$CL_i = 0$
1	$q \leftarrow q$	$q_i \leftarrow q_i$	$CL_i = 1$

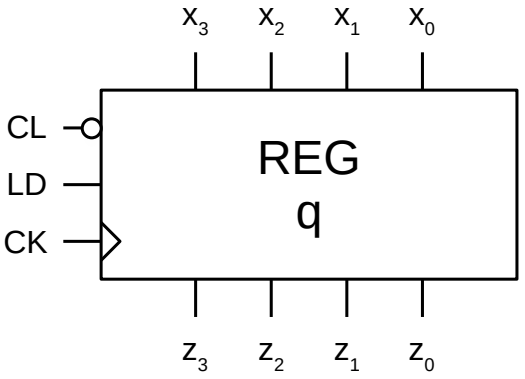
Synchronous operation table

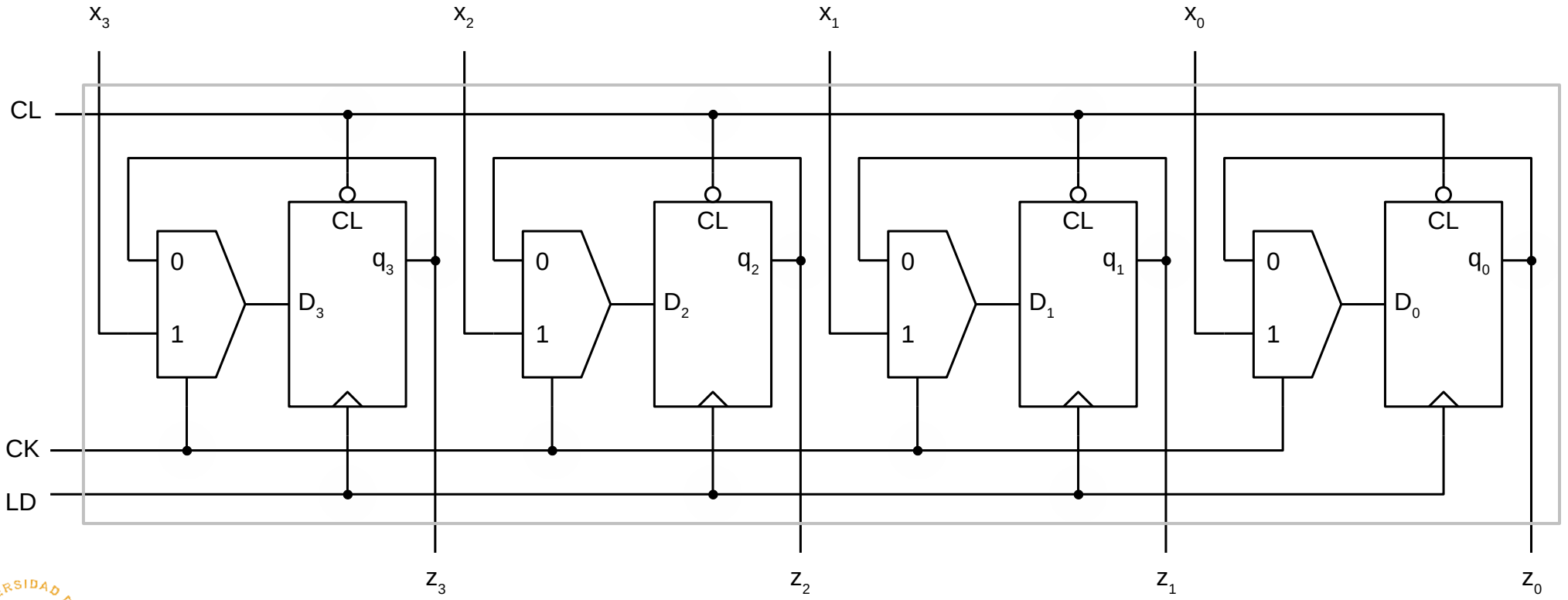
LD	Operation	Typ. stage	Excitation
1	$q \leftarrow x$	$q_i \leftarrow x_i$	$D_i = x_i$
0	$q \leftarrow q$	$q_i \leftarrow q_i$	$D_i = q_i$



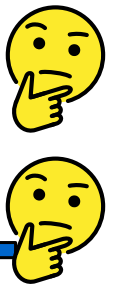


Parallel-in/parallel-out register





Design alternatives

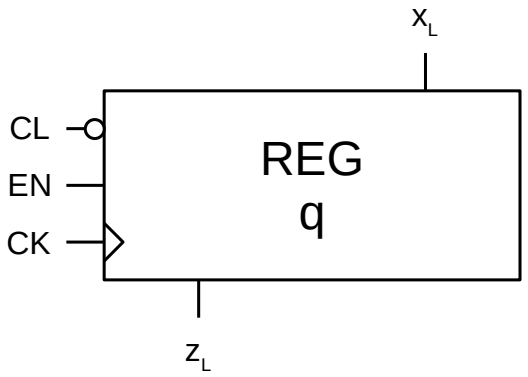


- The most straightforward method to design a register is using:
  - D flip-flops: input D is the next state.
  - Multiplexers: direct implementation of the operation table.
- But a register may be designed with any type of flip-flop and any type of combinational design technique.

Example 1

a) Design a parallel-in/parallel-out register using D flip-flops and logic gates.
b) Design a parallel-in/parallel-out register using JK flip-flops and multiplexers.
c) Design a parallel-in/parallel-out register using JK flip-flops and logic gates.

Shift register



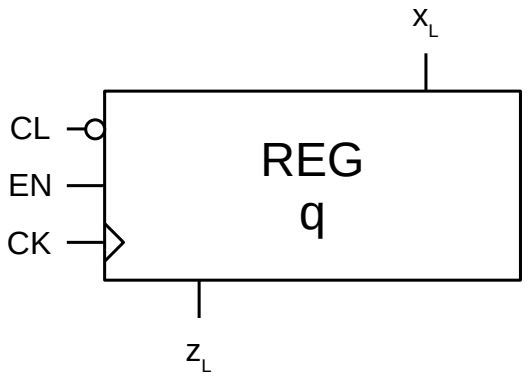
Operation table

CL, EN	Operation	Type
0x	$q \leftarrow 0$	async.
11	$q \leftarrow SHL(q,x_L)$	sync.
10	$q \leftarrow q$	sync.

Verilog code

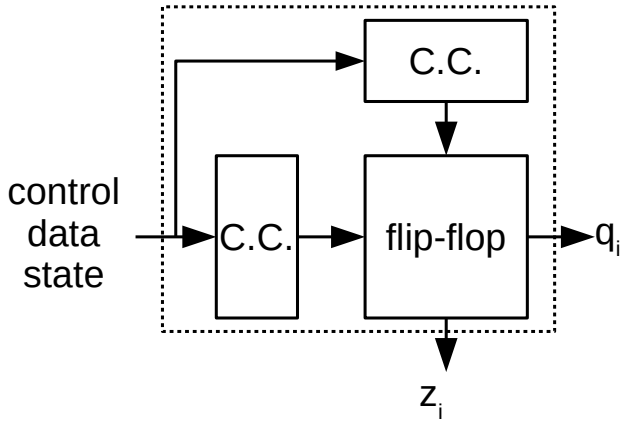
```
module reg_shl(
    input  wire ck,
    input  wire cl,
    input  wire en,
    input  wire xl,
    output wire zl
);

    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
```

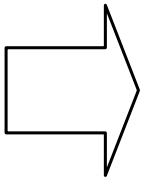
Shift register

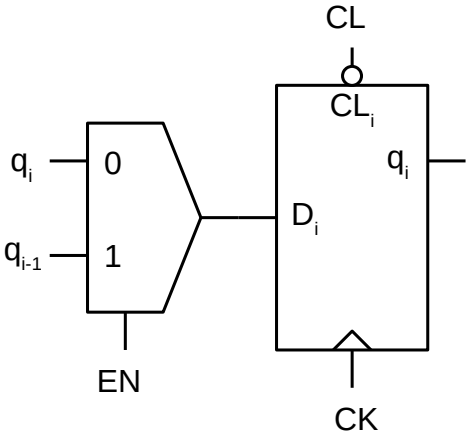


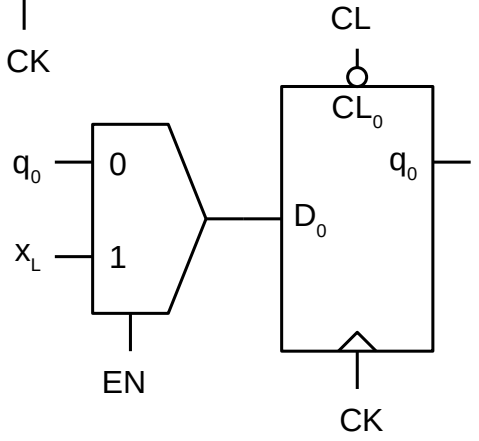
Synchronous operation table

EN	Operation	Typ. st.	St. 0	Typ. ex.	St. 0 ex.
1	$q \leftarrow SHL(q,x_L)$	$q_i \leftarrow q_{i-1}$	$\mathbf{q_0 \leftarrow x_L}$	$D_i = q_{i-1}$	$\mathbf{D_0 = x_L}$
0	$q \leftarrow q$	$q_i \leftarrow q_i$	$q_0 \leftarrow q_0$	$D_i = q_i$	$D_0 = q_0$

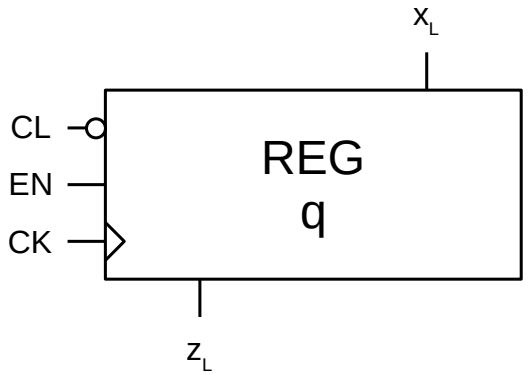


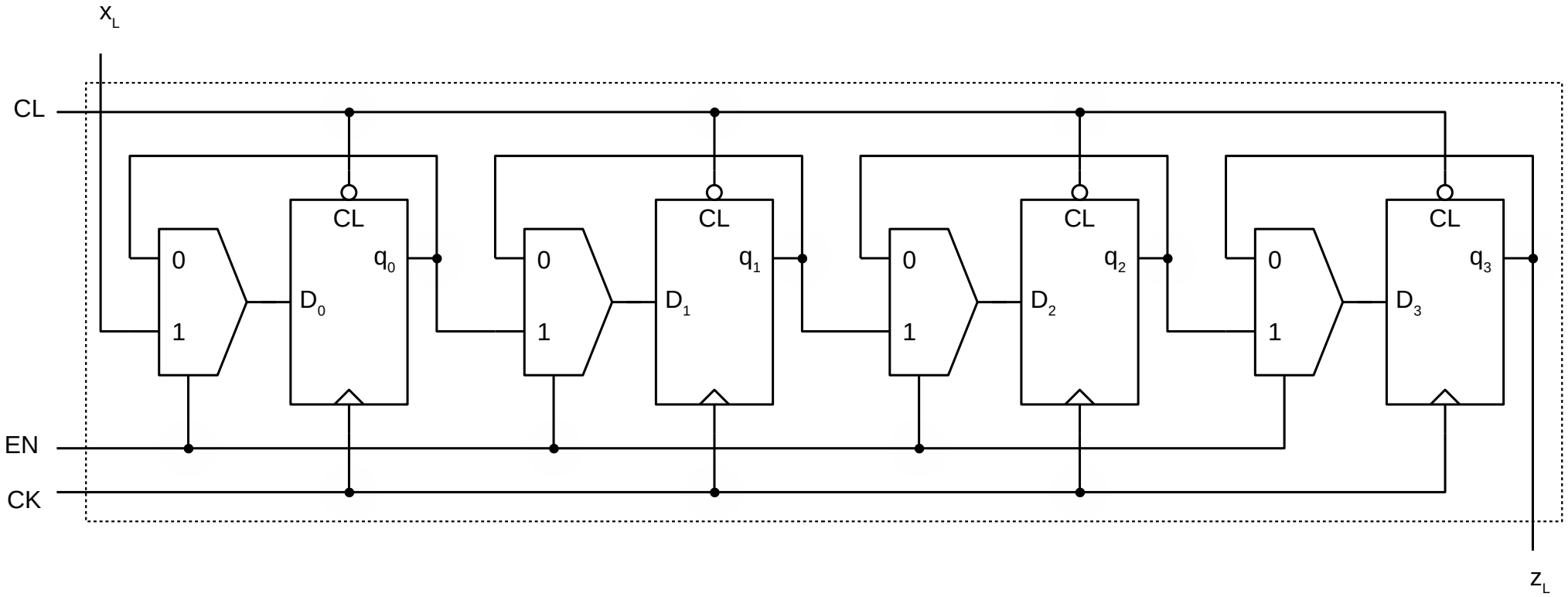






Shift register





Shift register

How to design a right-shift register?

Universal register

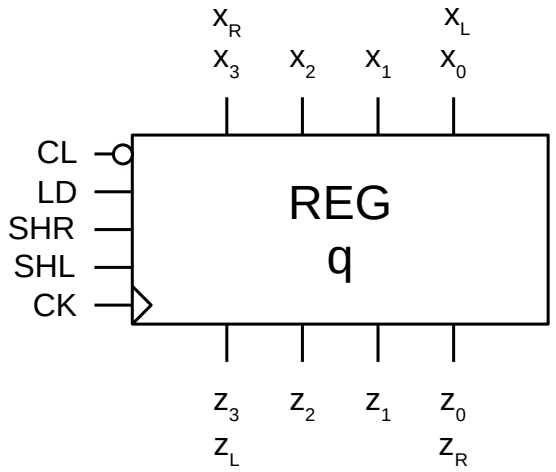


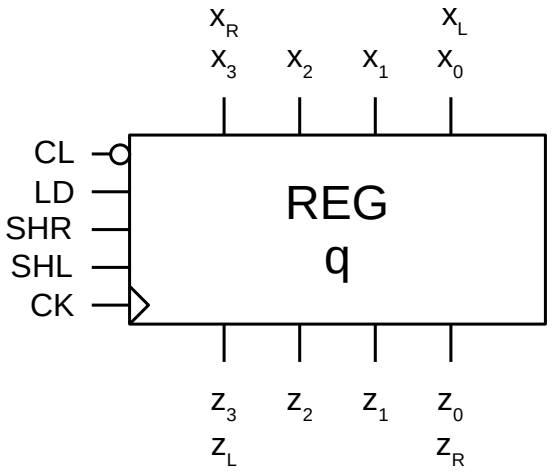
Tabla de operación

CL,LD,SHR,SHL	Operation	Type
0xxx	$q \leftarrow 0$	async.
11xx	$q \leftarrow x$	sync.
101x	$q \leftarrow SHR(q,x_R)$	sync.
1001	$q \leftarrow SHL(q,x_L)$	sync.
1000	$q \leftarrow q$	sync.

Verilog code

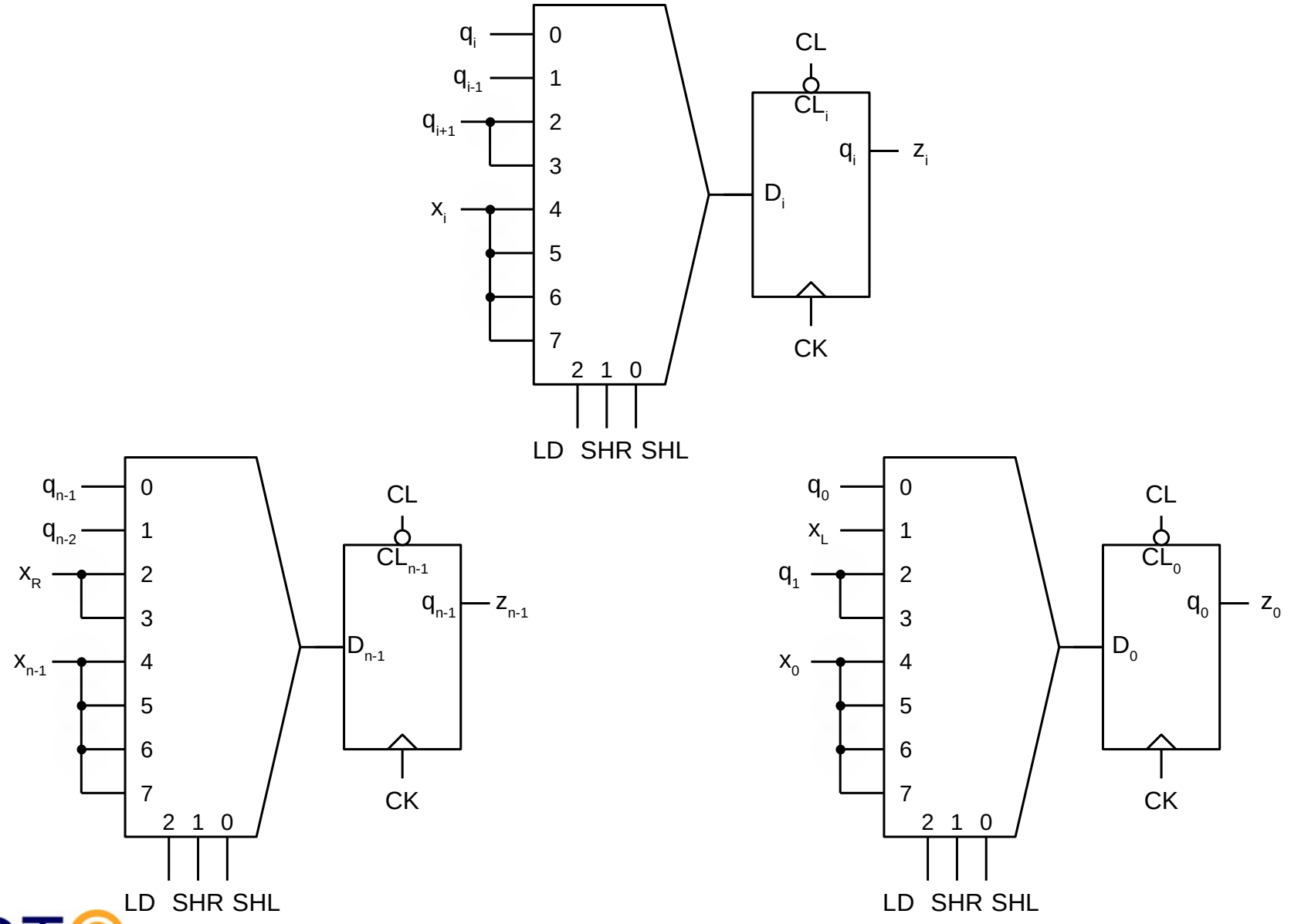
```
module ureg(
    input  wire ck,
    input  wire cl,
    input  wire ld,
    input  wire shr,
    input  wire shl,
    input  wire [3:0] x,
    output wire [3:0] z
);

    reg [3:0] q;

    always @(posedge ck, negedge cl)
        if (cl == 0)
            q <= 0;
        else if (ld == 1)
            q <= x;
        else if (shr == 1)
            q <= {x[3], q[3:1]};
        else if (shl == 1)
            q <= {q{2:0}, x[0]};

    assign z = q;

endmodule
```

Universal register



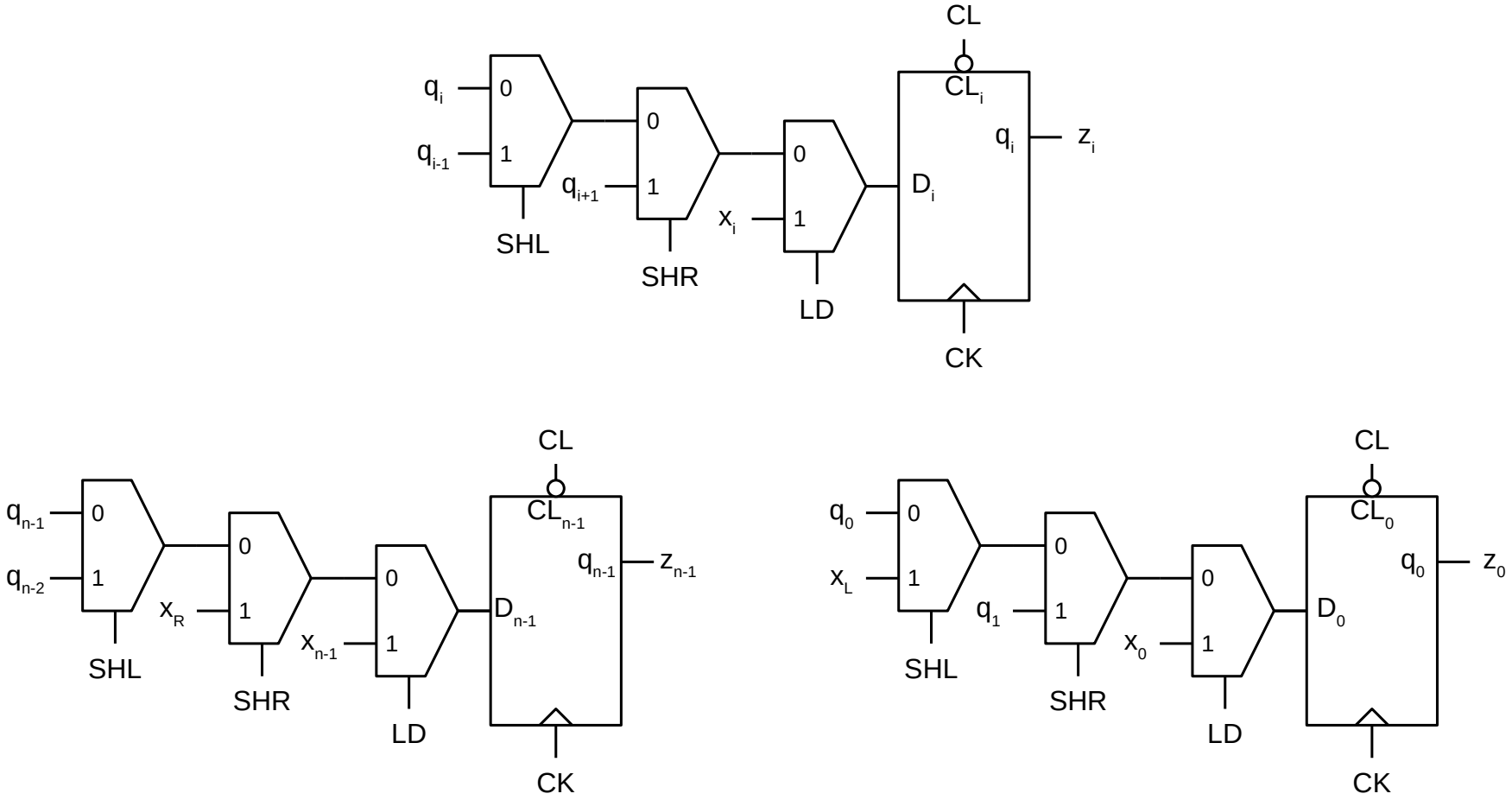
LD,SHR,SHL	Operation	Typ. stage	St. n-1	St. 0	Typ. exc.	Exc. n-1	Exc. 0
1xx	$q \leftarrow x$	$q_i \leftarrow x_i$	$q_{n-1} \leftarrow x_{n-1}$	$q_0 \leftarrow x_0$	$D_i = x_i$	$D_{n-1} = x_{n-1}$	$D_0 = x_0$
01x	$q \leftarrow SHR(q,x_R)$	$q_i \leftarrow q_{i+1}$	$\mathbf{q_{n-1} \leftarrow x_R}$	$q_0 \leftarrow q_1$	$D_i = q_{i+1}$	$\mathbf{D_{n-1} = x_R}$	$D_0 = q_1$
001	$q \leftarrow SHL(q,x_L)$	$q_i \leftarrow q_{i-1}$	$q_{n-1} \leftarrow q_{n-2}$	$\mathbf{q_0 \leftarrow x_L}$	$D_i = q_{i-1}$	$D_{n-1} = q_{n-2}$	$\mathbf{D_0 = x_L}$
000	$q \leftarrow q$	$q_i \leftarrow q_i$	$q_{n-1} \leftarrow q_{n-1}$	$q_0 \leftarrow q_0$	$D_i = q_i$	$D_{n-1} = q_{n-1}$	$D_0 = q_0$

Universal register



Universal register

Alternative implementation



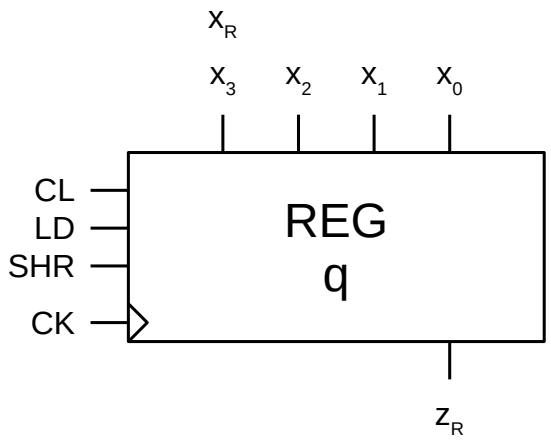
Excitation equations (“receips”) for register design using D flip-flops

Description	Operation	Stages	Excitations
Inhibition	$q \leftarrow q$	$q_i \leftarrow q_i$	$D_i = q_i$
Parallel load	$q \leftarrow x$	$q_i \leftarrow x_i$	$D_i = x_i$
Left shift	$q \leftarrow SHL(x,x_L)$	$q_i \leftarrow q_{i-1}$ $q_0 \leftarrow x_L$	$D_i = q_{i-1}$ $D_0 = x_L$
Right shift	$q \leftarrow SHR(x,x_R)$	$q_i \leftarrow q_{i+1}$ $q_{n-1} \leftarrow x_R$	$D_i = q_{i+1}$ $D_{n-1} = x_R$
Synchronous clear	$q \leftarrow 0$	$q_i \leftarrow 0$	$D_i = 0$
Asynchronous clear	$q \leftarrow 0$	$q_i \leftarrow 0$	$CL_i = 1$

Example

Example 2

Design the register specified below using D flip-flops.
Note: the clear input (CL) is synchronous.



CL,LD,SHR	Operación
1xx	$q \leftarrow 0$
01x	$q \leftarrow x$
001	$q \leftarrow SHR(q,x_R)$
000	$q \leftarrow q$

Contents

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

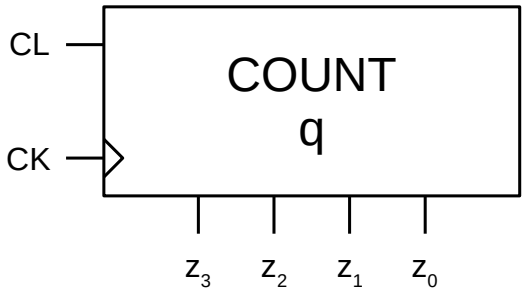
Counters

- Similar to the register: adds count operation
- Design
  - Same 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: current value of the count.
  - End-of-count (EOC): counter in the last count value.

Binary modulus 2^n up counter

- Modulus
  - Number of counter's states.
- Binary
  - Count states are consecutive base-2 numbers.
- Modulus 2^n
  - Counts from 0 to 2^n-1 (n bits).
- Up
  - Count increases: 0, 1, 2, 3, ...
- Cyclic count
  - First count state follows last count state (overflow).
  - Eg. 4-bit counter: … 13, 14, 15, 0, 1, 2, ...

Binary modulus 2^n up counter



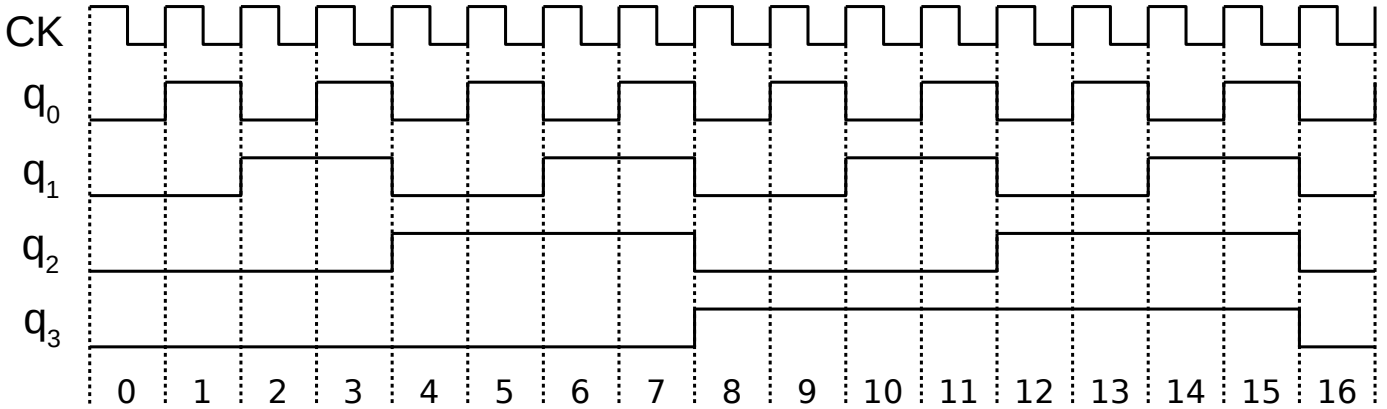
Operation table

CL	Operation	Type
1	$q \leftarrow 0$	async.
0	$q \leftarrow (q+1) \bmod 2^n$	sync.

Verilog code

```
module count_mod16(
    input  wire ck,
    input  wire cl,
    output wire [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
```

Binary modulus 2^n up counter
Count operation



JK / q	00	01	11	10
0	0	0	1	1
1	1	0	0	1

q ←

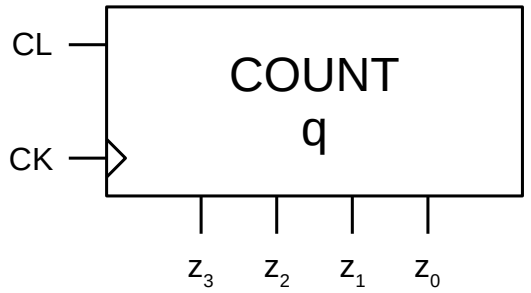
$$q_i = \begin{cases} \overline{q_i}, \text{if } q_j = 1 \; \forall \; j<i \\ q_i, \text{in other case} \end{cases} \longrightarrow q_i = \begin{cases} \overline{q_i}, \text{si } q_{i-1}\, q_{i-2} \ldots q_0 = 1 \\ q_i, \text{si } q_{i-1}\, q_{i-2} \ldots q_0 = 0 \end{cases} \longleftarrow q_i = \begin{cases} \overline{q_i}, \text{if } J_i = K_i = 1 \\ q_i, \text{if } J_i = K_i = 0 \end{cases}$$

$$q_j = 1 \; \forall \; j<i \Leftrightarrow q_{i-1}\, q_{i-2} \ldots q_0 = 1 \qquad J_i = K_i = q_{i-1} \ldots q_0$$

Operation	Typical stage	Stage 1	Stage 0
q ← (q+1) mod 2^n	$q_i \leftarrow \overline{q_i}$ si $q_{i-1} \ldots q_0=1$, q_i si $q_{i-1} \ldots q_0=0$	$q_1 \leftarrow \overline{q_1}$ si $q_0=1$, q_1 si $q_0=0$	$q_0 \leftarrow \overline{q_0}$

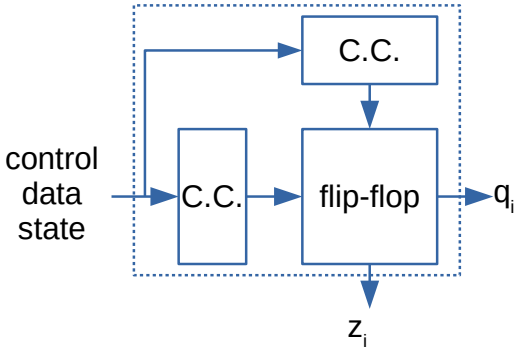
Operation	Typ. stage excitation	St.1 excitation	St. 0 excitation
q ← (q+1) mod 2^n	$J_i = K_i = q_{i-1} \ldots q_0$	$J_1 = K_1 = q_0$	$J_0 = K_0 = 1$

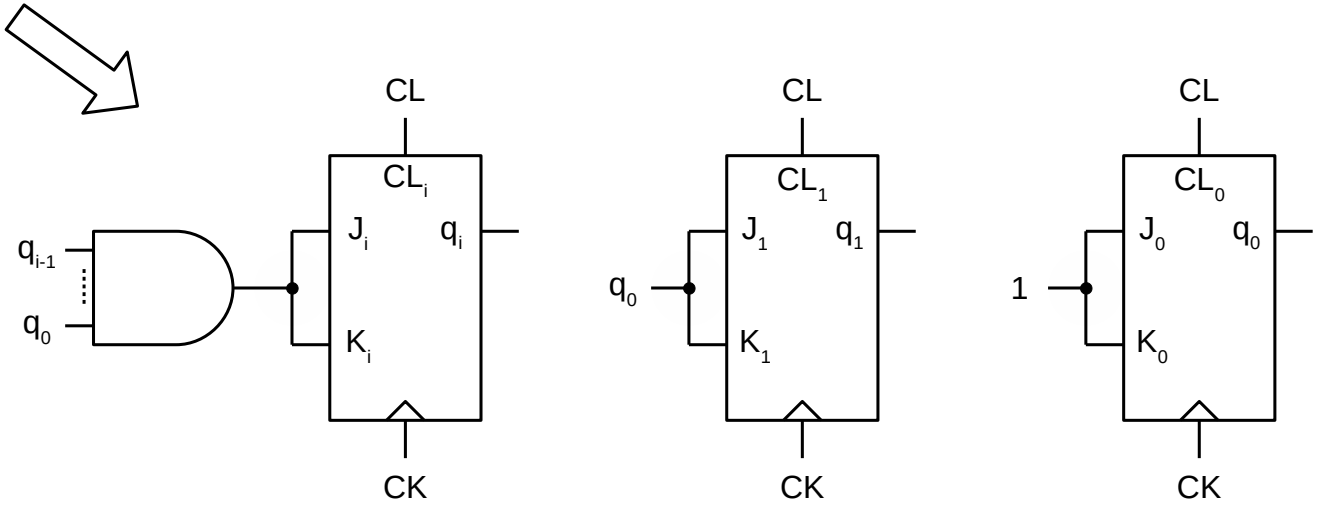
Binary modulus 2^n up counter



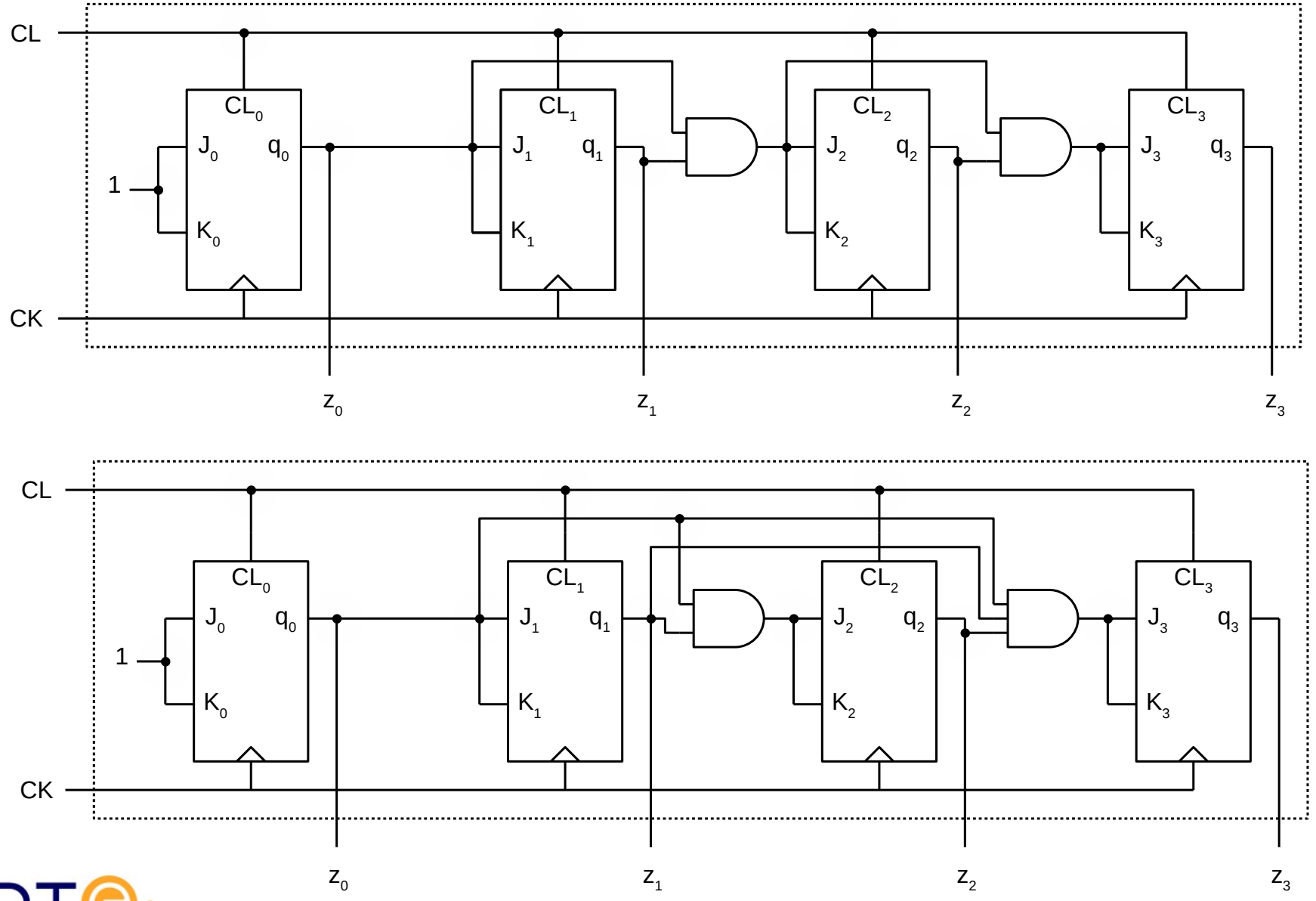
Synchronous operation table

Operation	Typ. st. excitation	St. 1 exc.	St. 0 exc.
$q \leftarrow (q+1) \bmod 2^n$	$J_i=K_i=q_{i-1} \dots q_0$	$J_1=K_1=q_0$	$J_0=K_0=1$

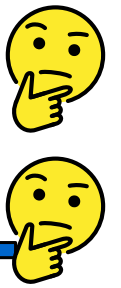




Binary modulus 2^n up counter. Circuit



Design alternatives



- JK flip-flops are convenient to design counter by hand because:
  - It is easy to implement the counting operation: $J_i = K_i = q_0 \ldots q_{i-1}$
  - It is easy to implement other operations as well.
- As with registers, counters can actually be implemented using any type of flip-flop.

Example 3

a) Design a binary modulus 2^n up counter using T flip-flops.
b) Design a binary modulus 2^n up counter using D flip-flops.

FPGA's building blocks only typically include D flip-flops, so all counters, registers and sequential logic in general are implemented using only D flip-flops in the end.

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

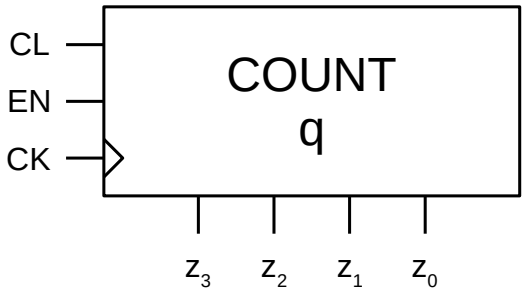


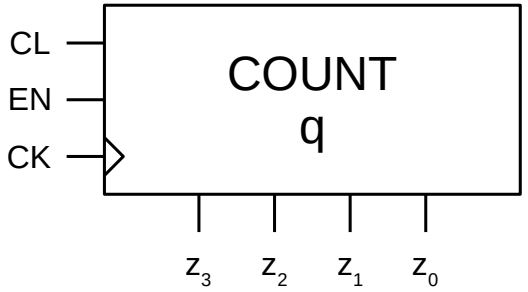
Tabla de operación

CL, EN	Operation	Type
1x	$q \leftarrow 0$	sync.
01	$q \leftarrow (q+1) \bmod 2^n$	sync.
00	$q \leftarrow q$	sync.

Verilog code

```
module count_mod16(
    input  wire ck,
    input  wire cl,
    input  wire en,
    output wire [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
```

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



JK / q	00	01	11	10
0	0	0	1	1
1	1	0	0	1

q ←

Description	Operation	Stage	St. excitation
Inhibition	$q \leftarrow q$	$q_i \leftarrow q_i$	$J_i=K_i=0$
Synchronous clear	$q \leftarrow 0$	$q_i \leftarrow 0$	$J_i=0, K_i=1$

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

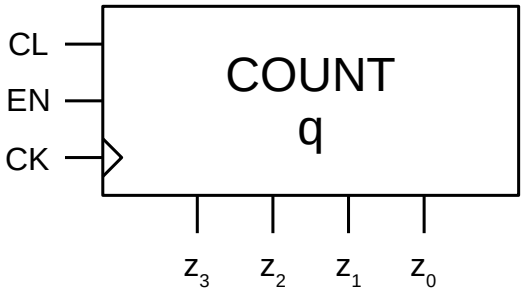
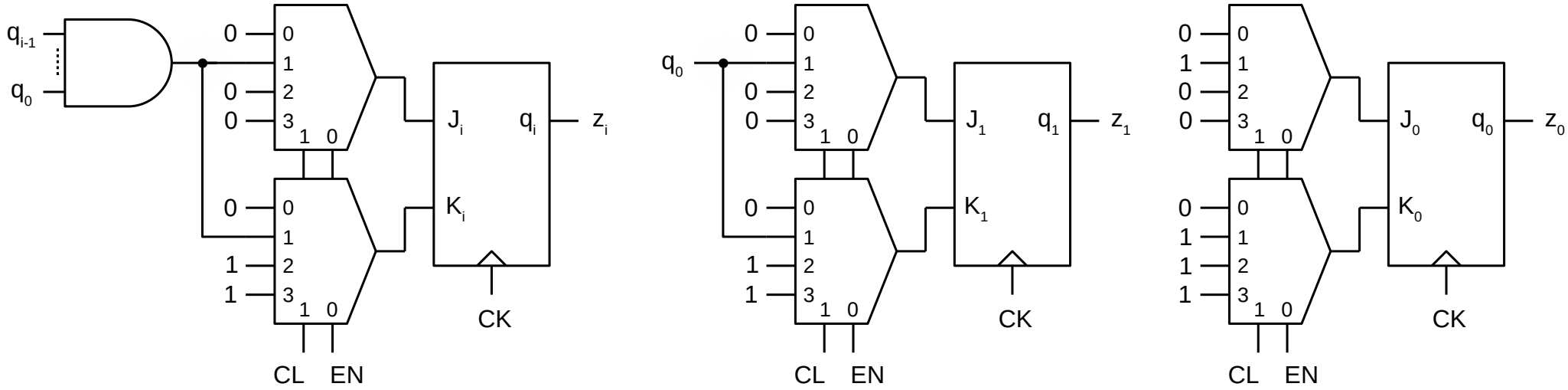


Tabla de operación síncrona

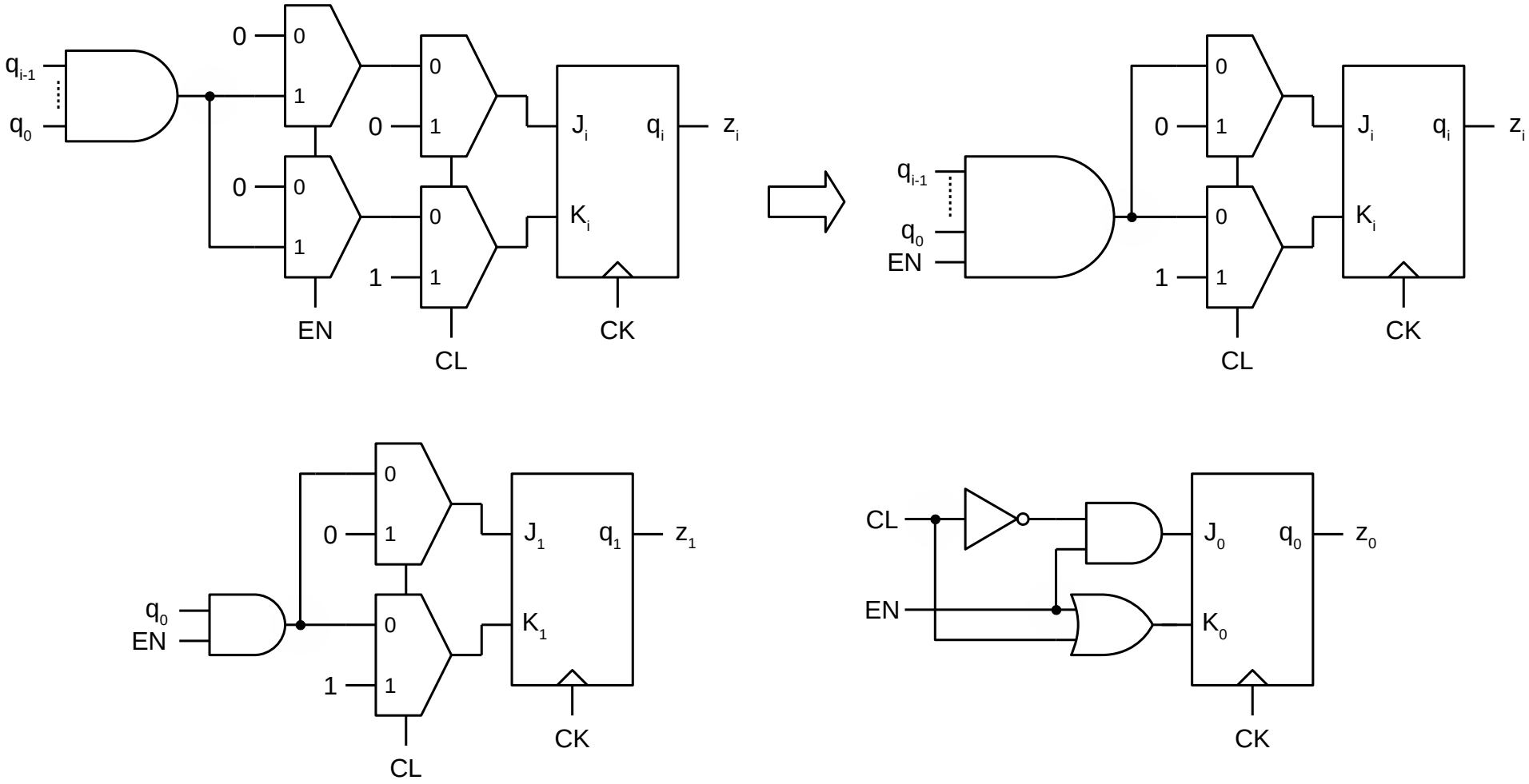
CL,EN	Operation	Typ. stage	Stage 1	Stage 0
1x	$q \leftarrow 0$	$J_i=0, K_i=1$	$J_1=0, K_1=1$	$J_0=0, K_0=1$
01	$q \leftarrow (q+1) \bmod 2^n$	$J_i=K_i=q_{i-1} \dots q_0$	$\mathbf{J_1=K_1=q_0}$	$\mathbf{J_0=K_0=1}$
00	$q \leftarrow q$	$J_i=K_i=0$	$J_1=K_1=0$	$J_0=K_0=0$

MUX 4:1 implementation



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

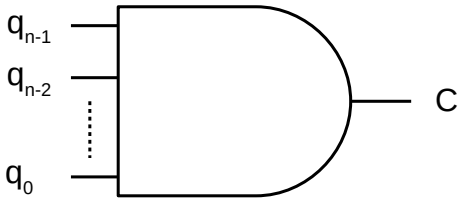
Alternative implementations



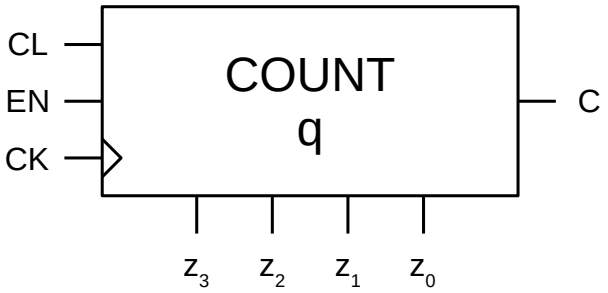
End-of-count output

- Up counter end-of-count (carry)
  - C = 1 if and only if $q = 2^n - 1$

$$C = q_{n-1}\, q_{n-2} \cdots q_0$$



End-of-count output



Verilog code

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

Counter combination

- Objective: combine two or more counters to obtain a bigger (more states) counter.
  - Module-k + module-l → Module-k*l
- Design:
  - Increment the most significant counter only when the least significant counter reaches it last state.
  - Counter combination is easier with appropriate inputs and outputs:
    - End-of-count: detect when a counter is at the maximum value.
    - Enable: enable a counter easily.

Example

Example 4

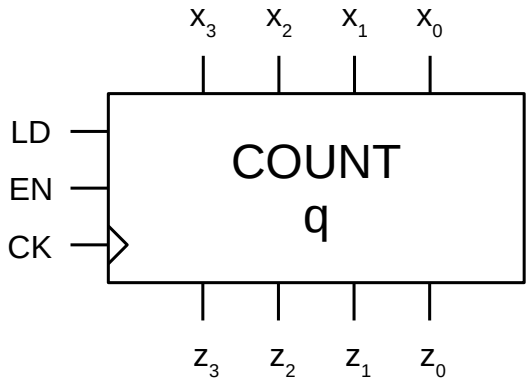
We have modulus-16 counters with synchronous clear (CL) and enable (EN) inputs, and end-of-count (carry) output (C). Using this counters and some logic gates we want to build:

a) A mudulus-256 counter with a synchronous clear input (CL).

b) A mudulus-256 counter with a synchronous clear (CL) and enable (EN) inputs and end-of-count (carry) output (C).

Clue: you do not need any gates to build counter (a), but you do for counter (b).

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



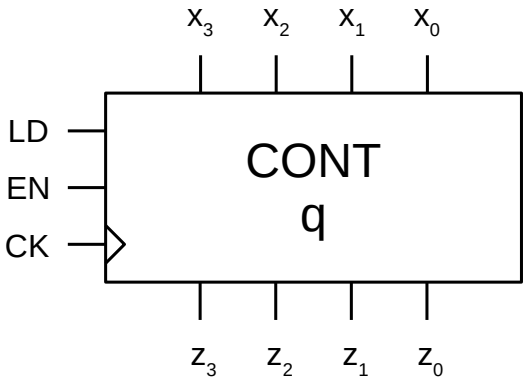
Operation table

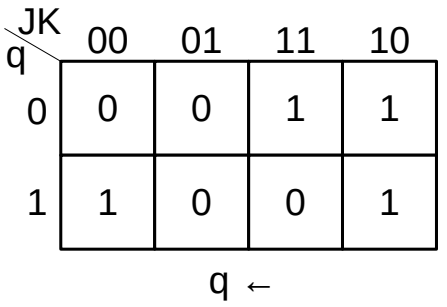
LD, EN	Operation	Type
1x	$q \leftarrow x$	sync.
01	$q \leftarrow (q+1) \bmod 2^n$	sync.
00	$q \leftarrow q$	sync.

Verilog code

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

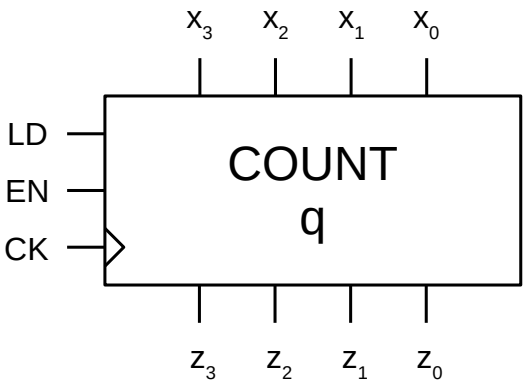
Binary modulus 2^n up counter with "load" and "enable" inputs





Description	Operation	Stage	St. excitation	St. excitation
Carga en paralelo	$q \leftarrow x$	$q_i \leftarrow x_i$	$J_i=1, K_i=0$ if $x=1$ $J_i=0, K_i=1$ if $x=0$	$J_i = x_i$ $K_i = \overline{x_i}$

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



LD,EN	Operation	Typ. stage	Stage 1	Stage 0
1x	$q \leftarrow x$	$J_i=x_i$, $K_i=\overline{x}_i$	$J_1=x_1$, $K_1=\overline{x}_1$	$J_0=x_0$, $K_0=\overline{x}_0$
01	$q \leftarrow (q+1) \bmod 16$	$J_i=K_i=q_{i-1} \ldots q_0$	$\mathbf{J_1=K_1=q_0}$	$\mathbf{J_0=K_0=1}$
00	$q \leftarrow q$	$J_i=K_i=0$	$J_1=K_1=0$	$J_0=K_0=0$