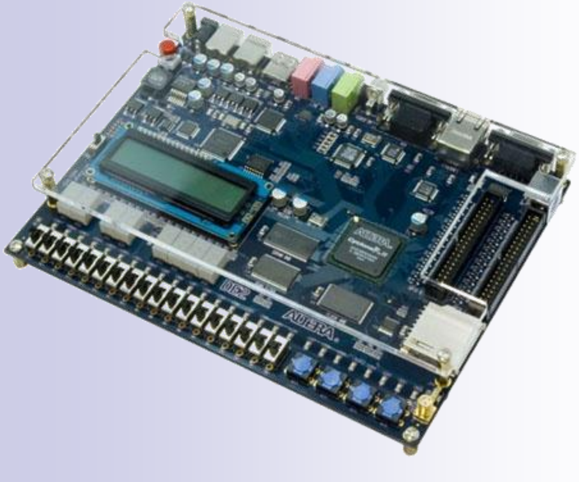
Logic Systems and Processors
cz: Logické systémy a procesory

Lecturer: Richard Šusta

richard@susta.cz, susta@fel.cvut.cz,
+420 2 2435 **7359**

Version V1.0



CTU-FEE in Prague, CR – subject BE5B35LSP

Finite State Machines

Terminology

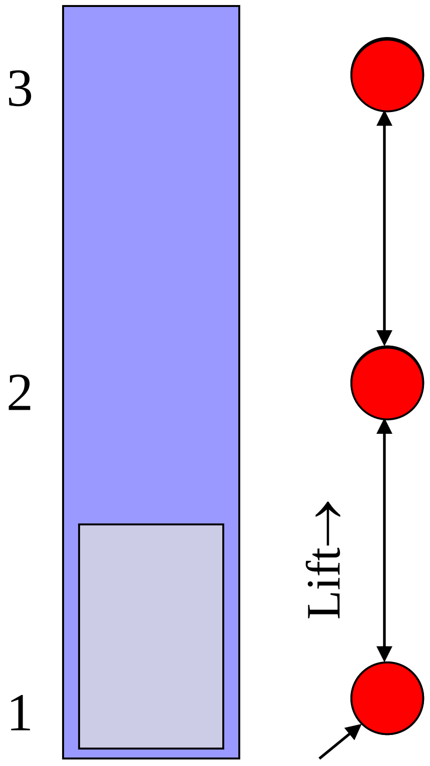
English: **Automaton,** plural **Automata**

– *technical English here derives the plural essentially from the Latin declension!*

Even though the word "automatons" exists, its usage in science papers is considered unknowing.

- „**Automaton**“ represents a general term.
- In technical English, **FSM "Finite State Machine**" means a deterministic automaton with binary inputs and outputs.
- The "**Finite State Automaton**", on the other hand, defines an automaton that can have abstract inputs and possibly also probabilistic or nondeterministic behavior, which will be discussed in later lectures.

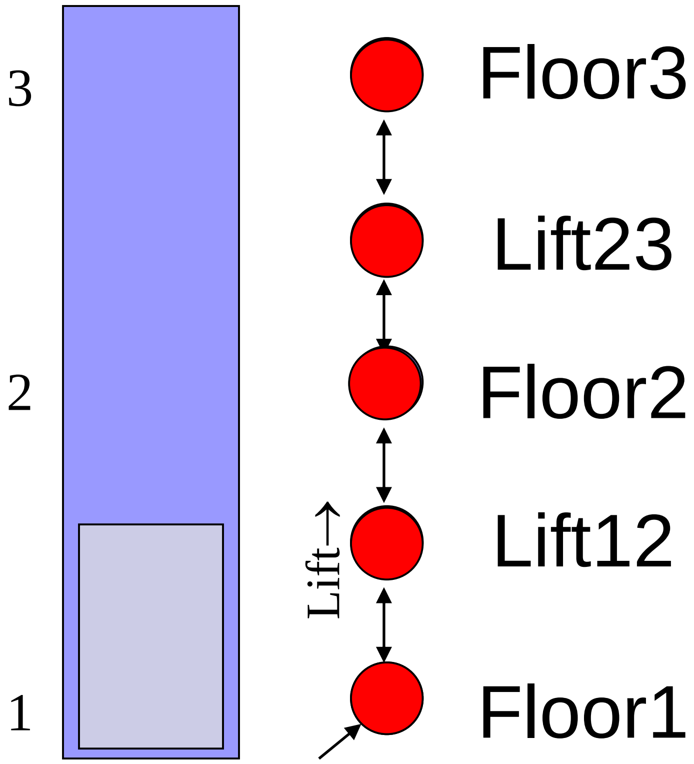
Example: Models of Lifts by FSM



FSM reduce a system to minimum states that are necessary for solved tasks.

If we are interested only in a position on the floors, then, one elevator can be described perhaps by this machine.

Example: Models of Lifts by FSM

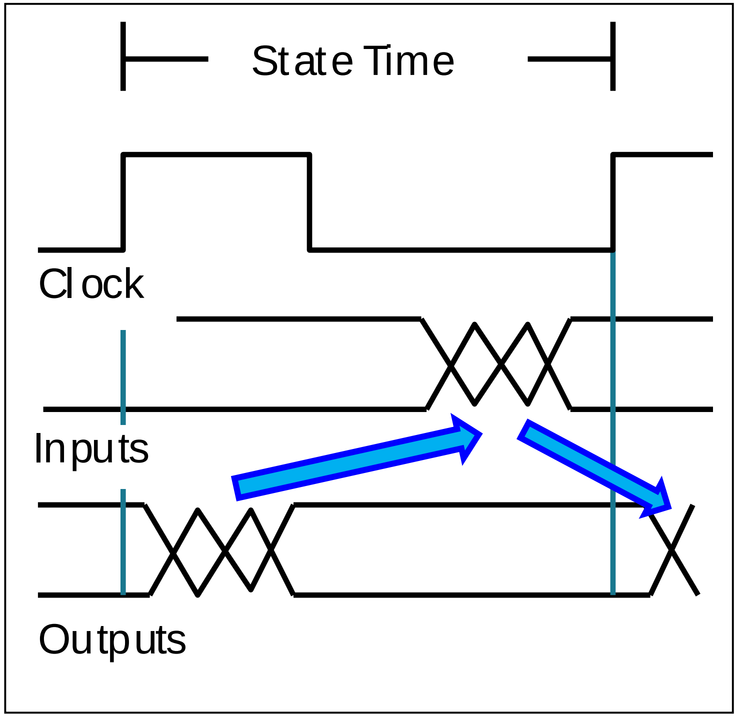


We choose states based on real events, that we need.
The model can include any detail ...

If we are interested not only in the floors, but in the lift ride, we add the necessary states.

Concept of State Machine

Example: **Positive Edge Triggered Synchronous System**



On rising edge, inputs sampled
outputs, next state computed

After propagation delay, outputs and
next state become stable

Immediate Outputs:
affect datapath immediately
could cause inputs from datapath to change

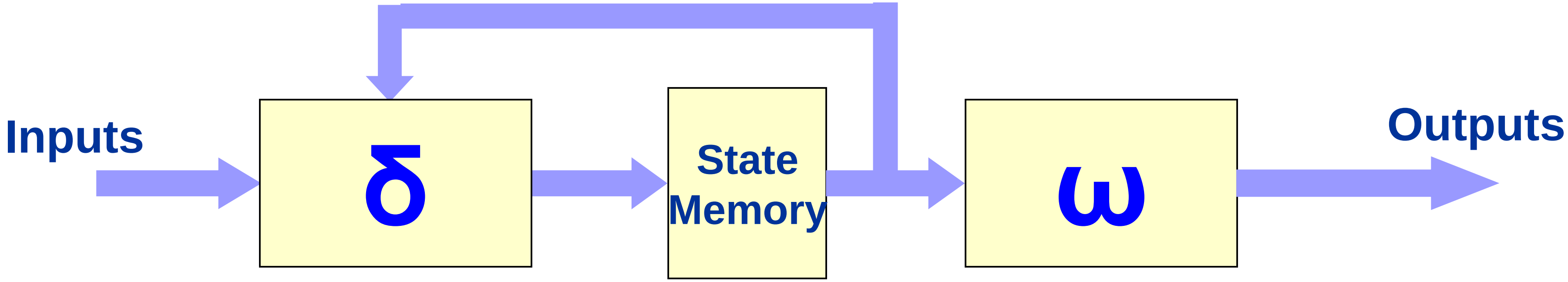
Delayed Outputs:
take effect on next clock edge
propagation delays must exceed hold times

Mealy and Moore Finite State Machine

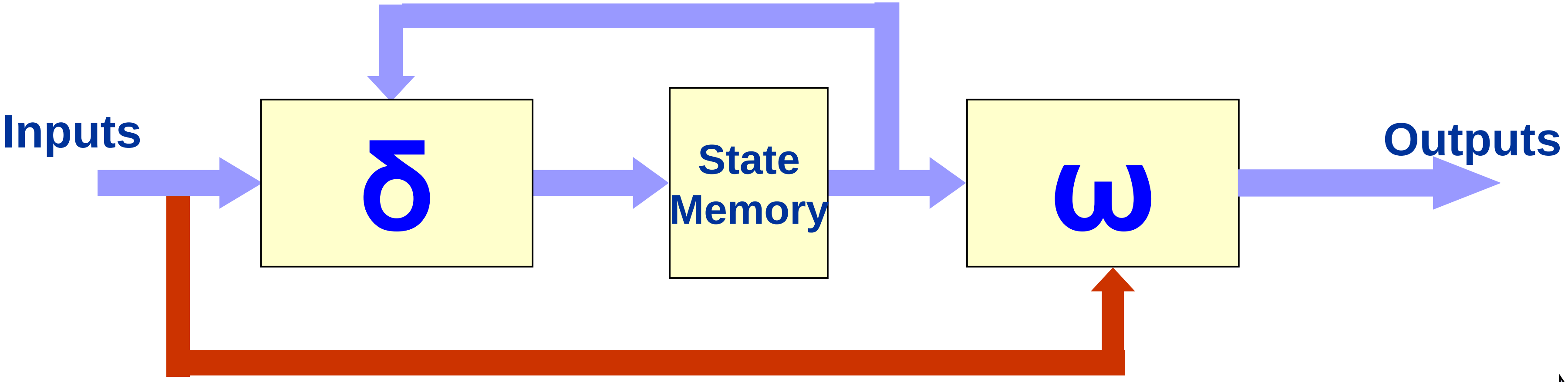
δ - next state combinational circuit

ω - output circuit, combinational/sequential

Moore's automaton



Mealy's automat



Mealy's machine is a year older :-)

- **G. H. Mealy**, "A Method for Synthesizing Sequential Circuits," *Bell Systems Tech. J.*, vol. 34, pp. 1045-1079, September 1955.
- **E. F. Moore**, "**Gedanken-Experiments** on Sequential Machines," *Annals of Mathematical Studies*, no. 34, pp. 129-153, 1956, Princeton Univ. Press, NJ.

Language note:

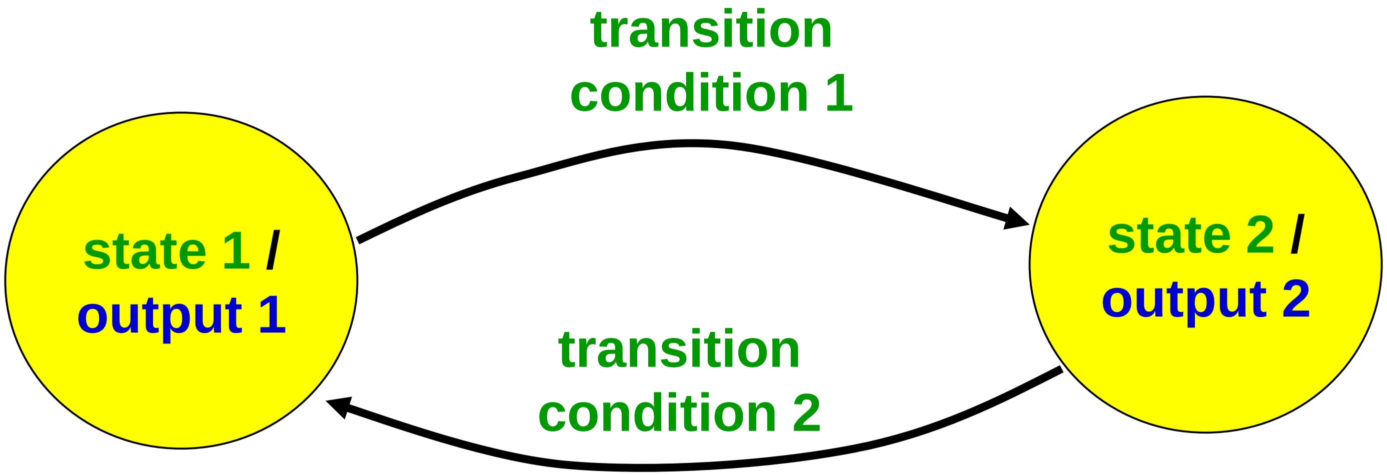
Gedanken *= plural noun in German*

Gedanke = idea, thought.

In English, it is used with "gedanken-experiment" to describe an approach using conceptual rather than actual experiments. For example, Albert Einstein often referred to his ideas of hypothetical experiments as such.

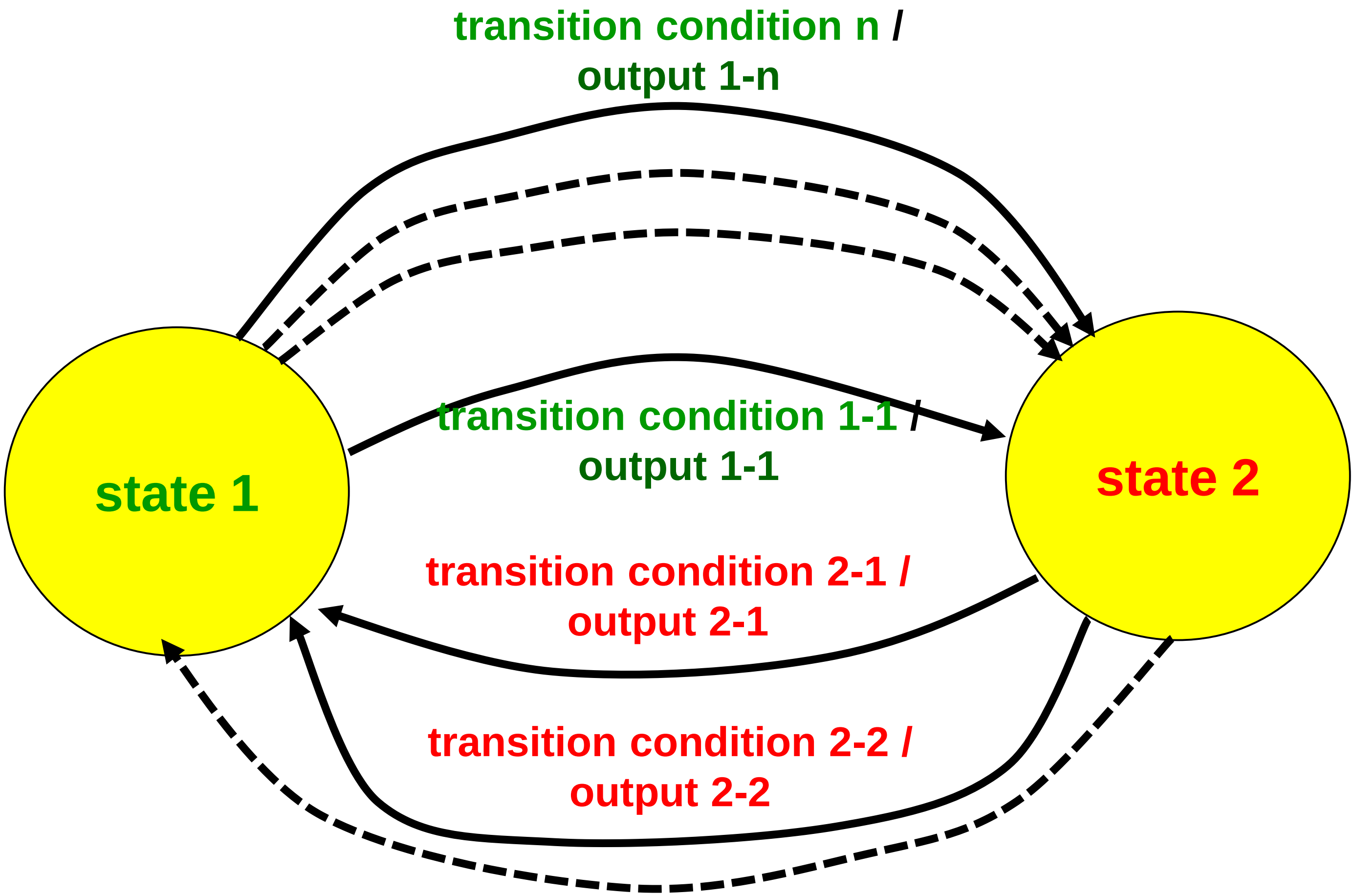
Moore Machine

- Outputs depend on states only



Mealy Machine

- Outputs depend on states and inputs


transition condition n /
output 1-n
transition condition 1-1 /
output 1-1
state 1
state 2
transition condition 2-1 /
output 2-1
transition condition 2-2 /
output 2-2

FSM Definitions

FSM - Finite State Machine

The ordered tuple

$$\mathbf{M = < X, S, Z,} \omega\mathbf{,} \delta\mathbf{, s_0 >}$$

- **X** - *finite set of all* input vectors
- **Z** - *finite set of all* output vectors
- **S** - *finite set of all* internal states
- δ - transient function is mapping δ**: X** x **S -> S**
- ω - output function is mapping ω**:**

  ω**: S -> Z (**Moore**)**

  ω**: X** x **S -> Z (**Mealy)
- $\mathbf{S_0}$ - initial state $S_0 \in S$

Moore and Mealy's machine

- Both machines have identical characteristics and only the outputs differ.
- **Moore's machine**:
  - □ The outputs do not depend on the inputs, i.e., the outputs are effectively generated from the state the FSM is currently in.

Mealy's machine:

- □ The outputs are generated from the current state and the immediate input, i.e., the outputs are created at the time of preparing the FSM transition from one state to another.
  It is used more often in programs - they take inputs from stable values in variables.

Moore / Mealy Theorem

Theorem:

- For any Moore automaton, there exists an equivalent Mealy automaton with the same or fewer states.
- Conversely, for every Mealy automaton that has N states and recognizes X possible combinations of inputs, there is an equivalent Moore automaton having N*X states.

The proof can be done by construction.

Systems with Internal States

Nonlinear system

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x},\mathbf{u})$$

$$\mathbf{y} = \mathbf{h}(\mathbf{x},\mathbf{u})$$

x - vector of n state variables
u - vector of m inputs
y - vector p of outputs

can be linearized in a working point

$$\frac{d\mathbf{x}}{dt} = \mathbf{Ax} + \mathbf{Bu}$$

$$\mathbf{y} = \mathbf{Cx} + \mathbf{Du}$$

Matrix A: $n \times n$, **B:** $n \times \mathrm{m}$
C: $p \times n$, **D:** $p \times \mathrm{m}$

or converted to a discrete form

$$\mathbf{x}[\tau+1] = \mathbf{f}(\mathbf{x}[\tau],\mathbf{u}[\tau])$$

$$\mathbf{y}[\tau] = \mathbf{h}(\mathbf{x}[\tau],\mathbf{u}[\tau])$$

The values of continuous variables are sampled with period T.
Its choice is critical!

FSM
Deterministic, binary inputs and outputs

$$\mathbf{s}[k+1] = \delta(\mathbf{s}[k],\mathbf{u}[k])$$

$$\mathbf{y}[k] = \omega(\mathbf{s}[k],\mathbf{u}[k])$$

Binary state, input and output at time k. The FSM can vary if great range

FSM

FSM uses the same concept as the description of continuous systems.

- δ - transient function calculates the change from the current state and inputs and represents the A and B matrices or function f(x,u).
- The ω - output function of the FSM generates an output based on the current state similar to the C and D matrices or h(x, u).

The D matrix will speed up the response but also increase the difficulty of driving. Natural systems often have D=0, and their output function simplifies to h(x), i.e., analogous to Moore's FSM.

Description of Moore's FSM

is based of graphs

Example Start-Stop

The circuit from the 1st LSP task

Image credit: Schneider Electric

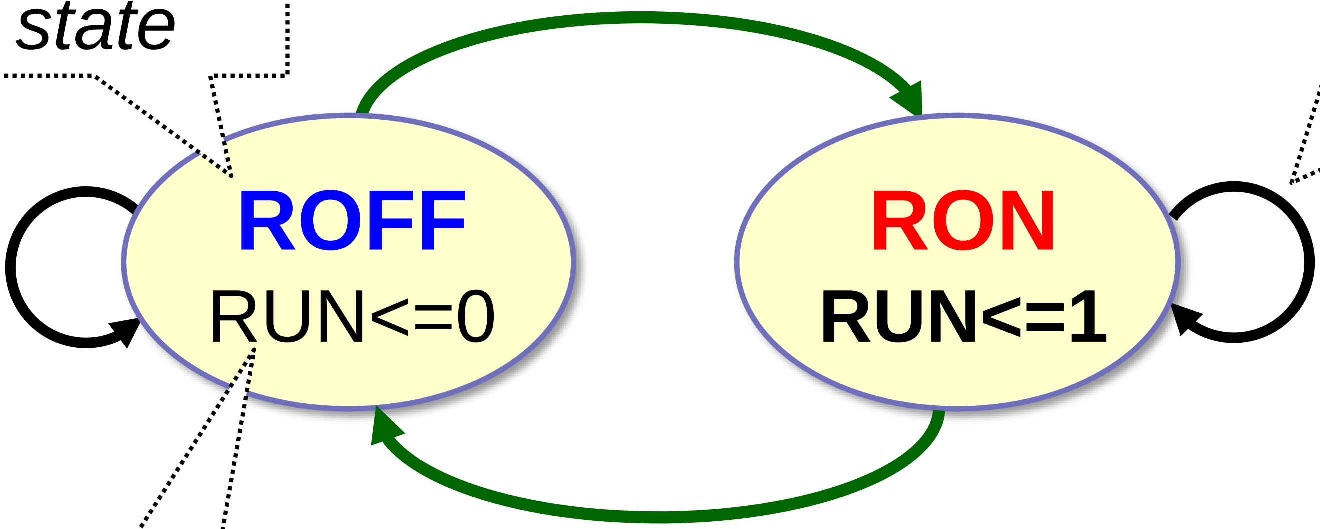
Start-Stop circuit from LSP task 1

Oriented chart
of its transitions

*logical condition for transition
from ROFF to **RON***

*Our name
of state*

Stop='0' and Start='1'



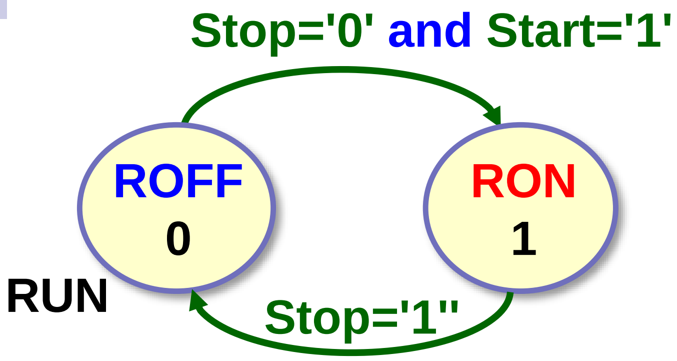
*Hint,
that FSM
remains
in the state
otherwise;
it is often
omitted*

Stop='1'

*generated
output*

*condition of transition
from **RON** to ROFF*

Start-Stop circuit from LSP task 1



Simplified graph of transitions

Transition graph

as **Transition table**

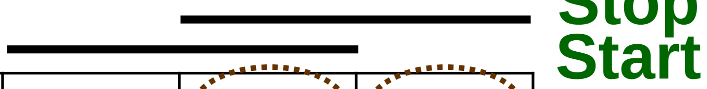
	ROFF	RON
ROFF	-	Stop=0 and Start=1
RON	Stop=1	-

Current Status

future state

Run
0
1

Table Outputs



ROFF	ROFF	RON	ROFF	ROFF
RON	ROFF	RON	ROFF	ROFF

mere highlighting the steady state

Run
0
1

State transition table

State Transition Table

The state transition table looks like an adjacency matrix utilized for describing graphs. It remains well-arranged even with a large number of conditions.

If p is the number of all states, then its dimension is p x p

rows - current states, from s1 to sp
columns - next states, from s1 to sp
intersection - conditions x transition from state sk to sj

$$M = \begin{array}{c||c|c|c||} & s1 & sj & sp \\ s1 & & & \\ \hline sk & & x & \\ \hline sp & & & \end{array}$$

<u>Terminological note:</u>
Do not confuse a
***state transition table** (finite state machines)*
with the term
***state-transition matrix** in control theory,*
which describes a similar phenomenon but
in a continuous area.

Transition table

2 parts - next state (δ) + outputs (ω)

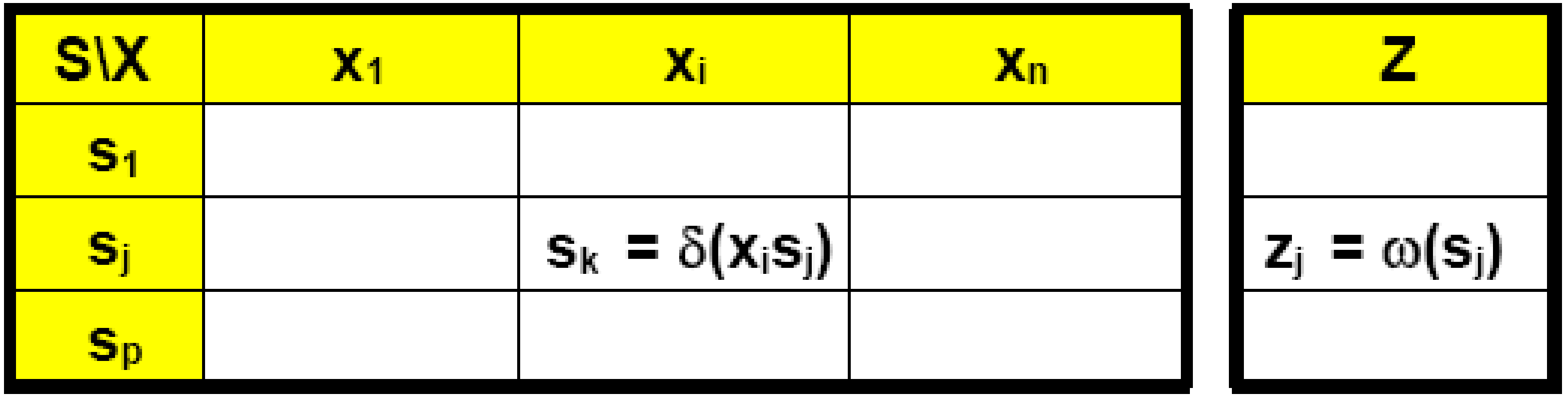
columns - all possible inputs x_i

rows - all possible internal states S_j + output z_m

intersection of row and column - subsequent state s_k

output table - output generated in the state

δ ω



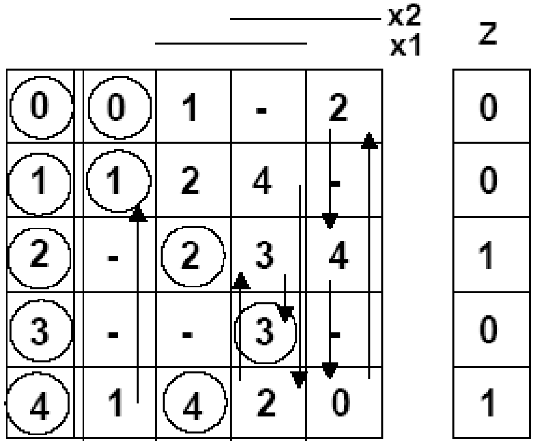
S\X	x_1	x_i	x_n
s_1			
s_j		$s_k = \delta(x_i s_j)$	
s_p			

Z
$z_j = \omega(s_j)$

stable state $s_j = \delta(x_i , s_j)$ - highlighted with a circle

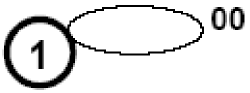
unstable state $s_k = \delta(x_i , s_j)$ $s_k \neq s_j$

4 basic transitions



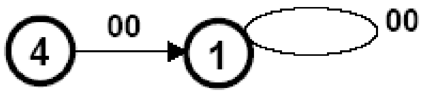
					x2 x1	z
0	0	1	-	2		0
1	1	2	4	-		0
2	-	2	3	4		1
3	-	-	3	-		0
4	1	4	2	0		1

1. Steady state



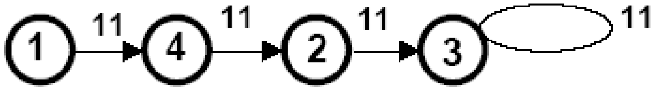
d(xi, sj) = sj

2. Simple transition



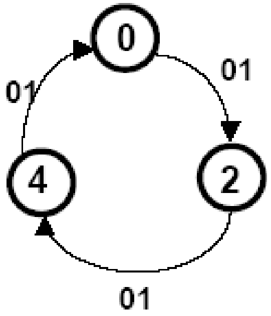
3. Cycle

- through unstable interstates,

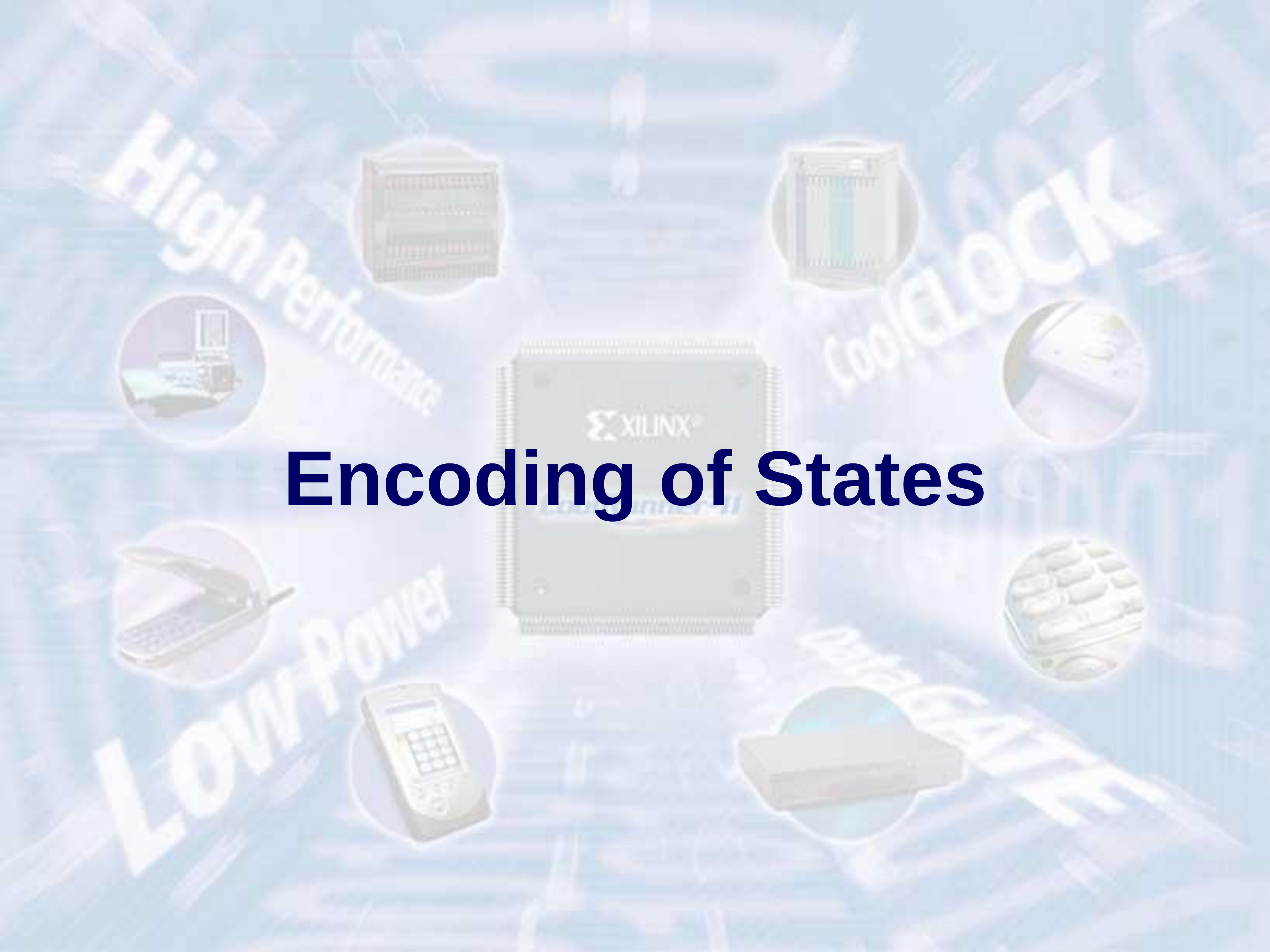


4. Oscillations

Note: counters work in oscillation, with each edge of the clock they go to the next state.



Encoding of States

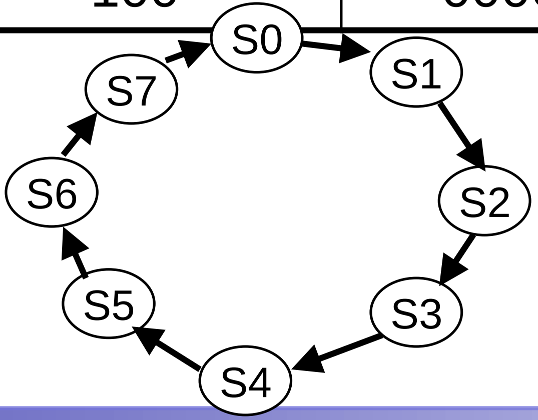


NP-complete state coding problem

- The encoding of the state can significantly affect the optimal implementation of the automaton.
  - □ Finding the optimal code means trying all possible variations. We can check them only for small FSMs.
  - □ In general, pseudo-optimal state codes are chosen, i.e., assigned by some heuristic in the hope that they will be close to the optimal ones...

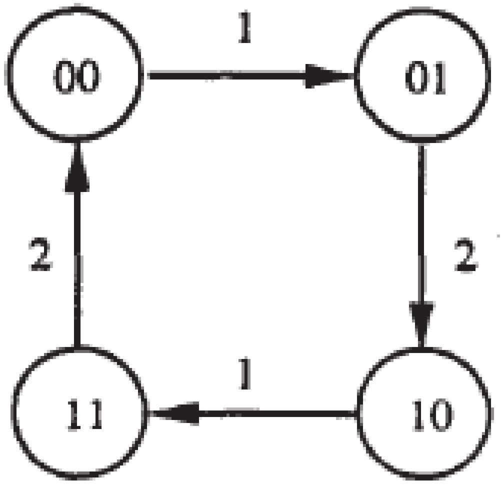
Examples of Coding Heuristics

State	Binary code	Gray's - minimal change	1 of N one hot	Enumeration - encoding is left to the compiler
S0	000	000	10000000	state_init
S1	001	001	01000000	state1
S2	010	011	00100000	state2
S3	011	010	00010000	state3
S4	100	110	00001000	state4
S5	101	111	00000100	state5
S6	110	101	00000010	state6
S7	111	100	00000001	state_last

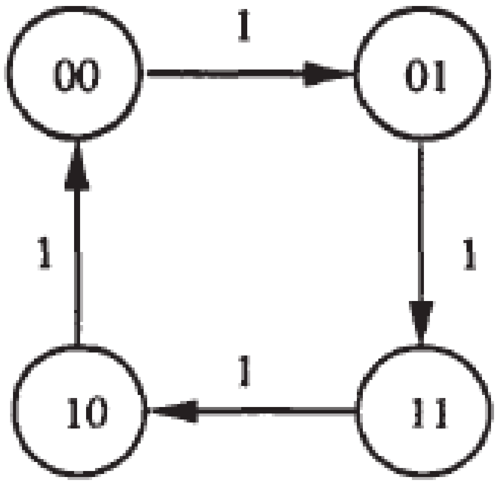


State coding heuristics 1/2

- **Binary** (sequential) - just assigning sequence numbers
  - □ A small number of state registers
  - □ State-dependent combinational circuits will grow, because we're constantly decoding the state number.
- **Minimal bit change:** assign codes to states in such an order that only the minimum number of bits change during transitions between them, usually according to Gray's code if possible. This is a variant of binary coding.



binary versus



minimum bit change

note: numbers on the edges indicate the number of bits changed during the transition

State coding heuristics 1/2

- **1 of N - One-Hot** - only one bit is in 1
  - □ - The state memory size is equal the number of states.
  - □ + 1 of N (One-hot) simplifies state-dependent combinational circuits because we test only 1 bit.
  - □ + We get fast transition functions.

- **Enumeration types**
  - □ - we need to introduce a new type.
  - □ + The compiler itself decides on the optimal code - it usually chooses something close to code 1 of N.

Examples of state encoding in VHDL

```
constant s0 : integer := 0;
constant s12 : integer := 1;
constant s3 : integer := 2;
signal state : integer range 0 to 2; -- We must specify range!!!
```

Safer coding subtypes

```
    SUBTYPE state_type is unsigned(1 downto 0);
    CONSTANT s0 : state_type := "00" ;
    CONSTANT s12 : state_type := "01" ;
    ...
```

Recommended encoding by enumeration type

```
    type state_t is (s0, s12, s3);

    signal state : state_t;
```

Enum type

Quartus II translates enumeration types using code 1 of N "one-hot"
[*Quartus reference manual*]

Quartus translates all 'enum' types as FSMs, so use 'enum' types only in them.

A note for very advanced study:

The enum encoding is determined by "Setting (Ctrl+Shift+E) -> Analyse & Synthetis Settings -> [More Settings] -> State Machine Processing", which is Auto by default. It defines the default value of the sys_encoding *attribute.*

- *If we want to use some 'enum' types as just numeric constants, we have to add a special attribute* **enum_encoding**
  (Google search: "Quartus attribute enum_encoding*" will give a* link *)*
- *But the enum_encoding attribute will block the enum type from being recognized as FSM states. So do not use it for enum intended to encode states. These are intended for the aforementioned sys_encoding.*

For details, see Quartus handbook, pages 16-36 to 16-38, in pdf 741 to 743:
https://dcenet.felk.cvut.cz/edu/fpga/doc/quartusii_handbook_archive_131.pdf

Example Start-Stop

from the 1st LSP tasks

Start-Stop as a Circuit

```
library ieee;use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity StartStopFSM is port (
        START, STOP, CLK : in std_logic;
        CLRN : in std_logic; -- FSMs prefer synchronous clear
        RUN : out std_logic);
end entity;
architecture rtl1 of StartStopFSM is
begin -- architecture
ilog: process(CLK)
        begin
           -- the next slide
        end process;
end architecture;
```

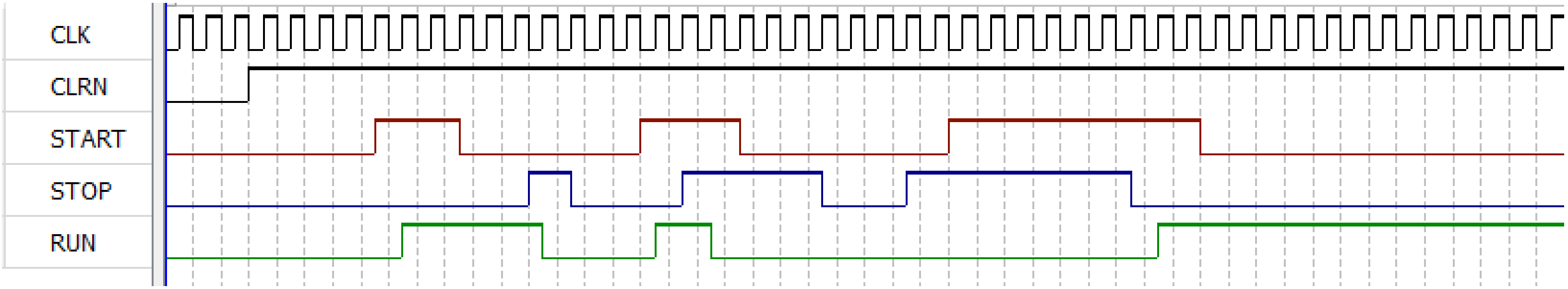
Start-Stop as a Circuit

```
architecture rtl1 of StartStopFSM is
begin -- architecture
iproc: process(CLK)
        begin
          if rising_edge(CLK) then
                    if (STOP or not CLRN)='1' then
                          RUN <='0';
                    elsif START ='1' then
                          RUN <='1';
                    end if;
          end if;
      end process;
end architecture;
```

Start-Stop - VHDL FSM

```
architecture rtlFSM of StartStopFSM is
begin
ifsm: process(CLK)
 type state_t is (ROFF, RON); -- enumerated types are reserved only for FSMs
 variable state: state_t:=ROFF;
 begin
   if rising_edge(CLK) then
      case state is
         when ROFF =>
δ          if (START and CLRN and not STOP)='1' then state:=RON; end if;
         when RON =>
           if (not CLRN or STOP)='1' then state:=ROFF; end if;
         end case;
      end if;
ω     if state=RON then RUN<='1'; else RUN<='0'; end if;
 end process; end architecture;
```

Simulation



- *The automatic description has not yet yielded a visible advantage for the simple circuit.*
- *However, it will make it easier to modify.*

Fuse: Start holds for over 2 and a half seconds

```
architecture rtlTiming of StartStopFSM is begin
ifsm: process(CLK) -- CLOCK_50 MHz
  type state_t is (ROFF, RTIME, RON);
  -- enumerated types are reserved only for FSMs
  variable state: state_t:=ROFF;
  variable cntr: unsigned(27 downto 0):=(others=>'0');
                      --2**27 / 50e6 = 2.68 seconds
  variable isTiming:boolean:=FALSE;
  begin


 end process; end architecture;
```

Fuse: Start holds for over 2 and a half seconds

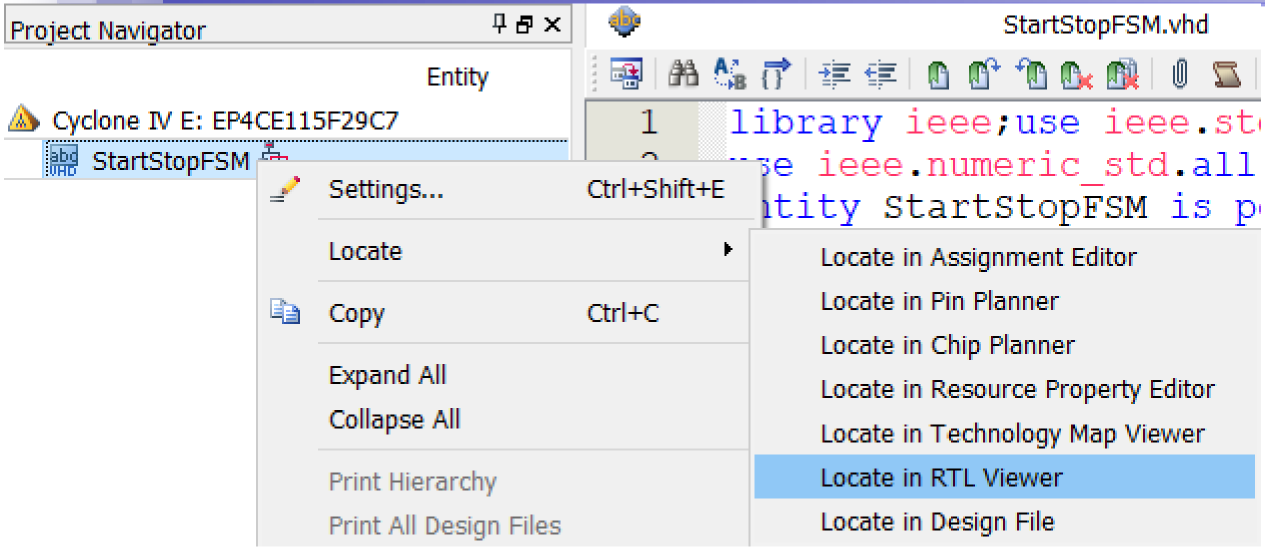
```
begin -- process
 if rising_edge(CLK) then
    isTiming:=FALSE; RUN<='0';                                    FSM
    if (CLRN='0' or STOP='1') then state:=ROFF;
    else case state is
             when ROFF => if START='1' then state:=RTIME; end if;
             when RTIME=> isTiming:=TRUE;
                          if START='0' then state:=ROFF;
                          elsif cntr(cntr'HIGH)='1' then state:=RON; end if;
              when RON => RUN<='1';
         end case;
   end if; -- if (CLRN='0' or STOP='1')
   if isTiming then cntr:=cntr+1; else cntr:=(others=>'0'); end if;
 end if; -- if rising_edge(CLK) then
end process;
end architecture;
```

Note: FSM should always be a boss in its circuits.
It does not count, i.e., work, only commands the time counter.

Whole Process

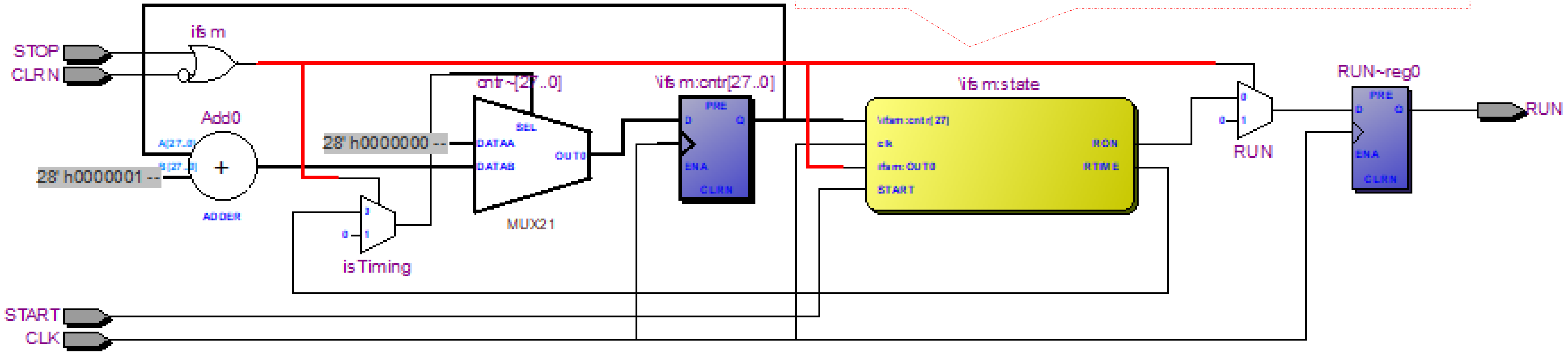
```
architecture rtlTiming of StartStopFSM is begin
ifsm: process(CLK) -- CLOCK_50 MHz
  type state_t is (ROFF, RTIME, RON); -- enumerated types are reserved only for FSMs
  variable state: state_t:=ROFF;
  variable cntr: unsigned(27 downto 0):=(others=>'0'); --2**27 / 50e6 = 2.68 seconds
  variable isTiming:boolean:=FALSE;
  begin if rising_edge(CLK) then
            isTiming:=FALSE; RUN<='0';
            if (CLRN='0' or STOP='1') then state:=ROFF;
            else case state is
                    when ROFF => if START='1' then state:=RTIME; end if;
                    when RTIME=> isTiming:=TRUE;
                                 if START='0' then state:=ROFF;
                                 elsif cntr(cntr'HIGH)='1' then state:=RON; end if;
                    when RON => RUN<='1';
                 end case;
           end if; -- if (CLRN='0' or STOP='1')
           if isTiming then cntr:=cntr+1; else cntr:=(others=>'0'); end if;
    end if; -- if rising_edge(CLK) then
  end process; end architecture;
```

RTL Viewer

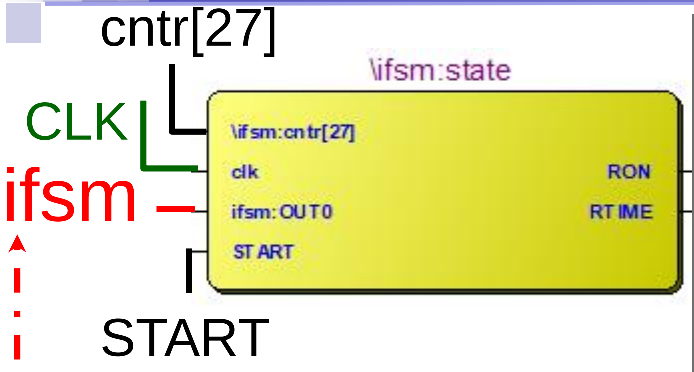


```
library ieee;use ieee.st
use ieee.numeric_std.all
ntity StartStopFSM is p
```

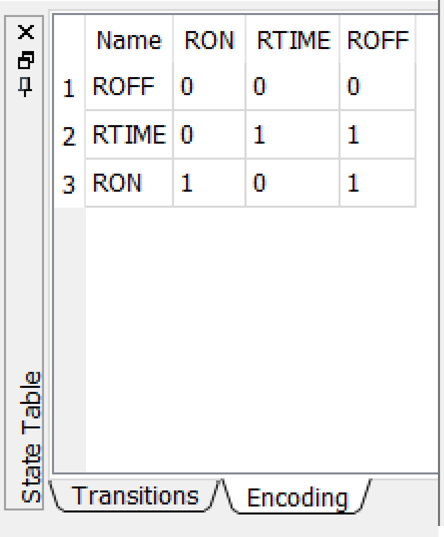
ifsm:compiler's signal



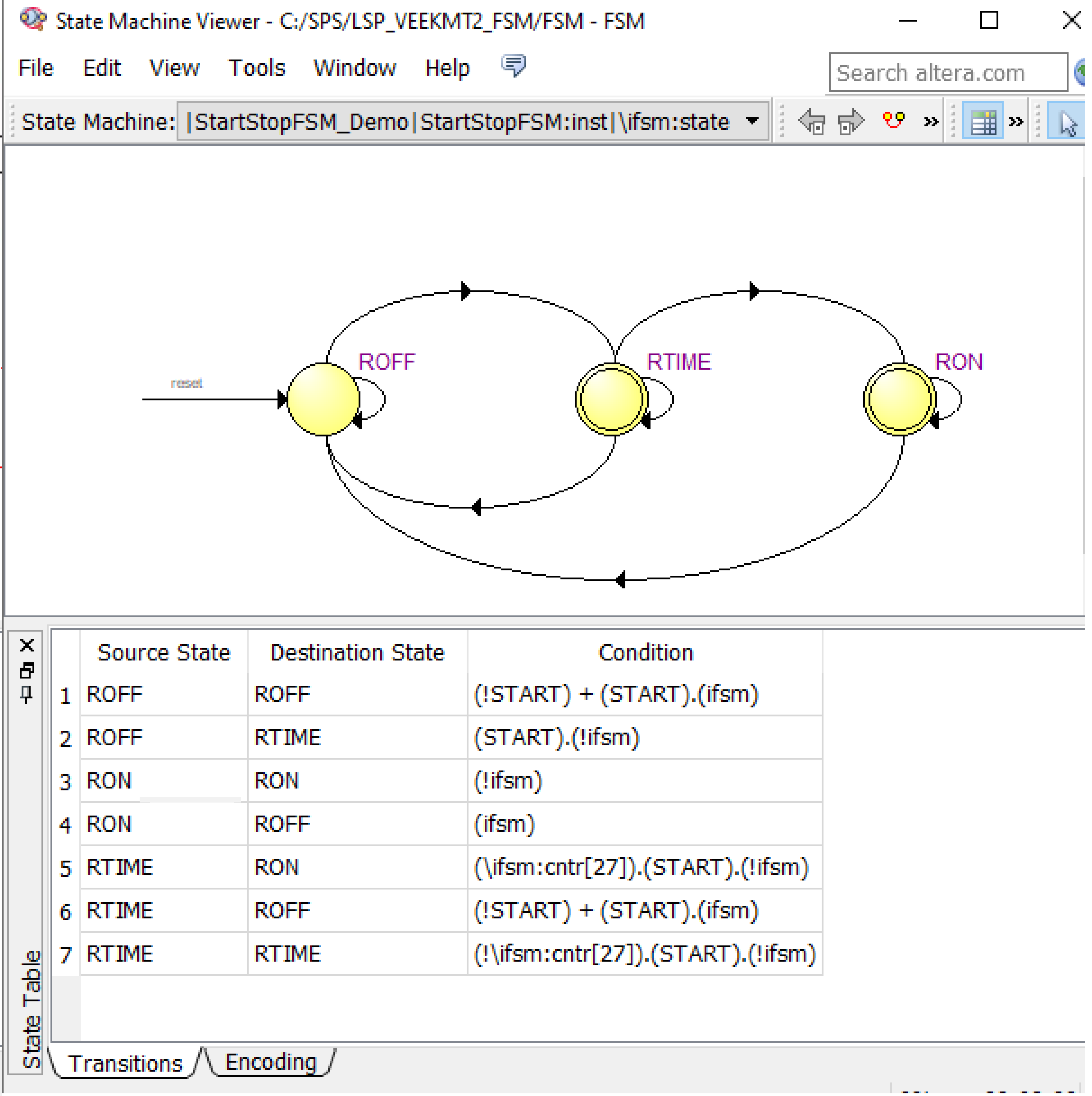
RTLViewer



(CLRN='0' or STOP='1')



	Name	RON	RTIME	ROFF
1	ROFF	0	0	0
2	RTIME	0	1	1
3	RON	1	0	1



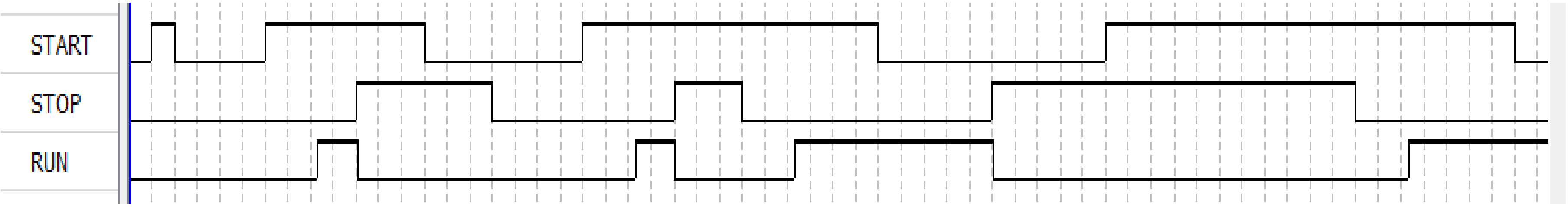
State Machine Viewer - C:/SPS/LSP_VEEKMT2_FSM/FSM - FSM

State Machine: |StartStopFSM_Demo|StartStopFSM:inst|\ifsm:state

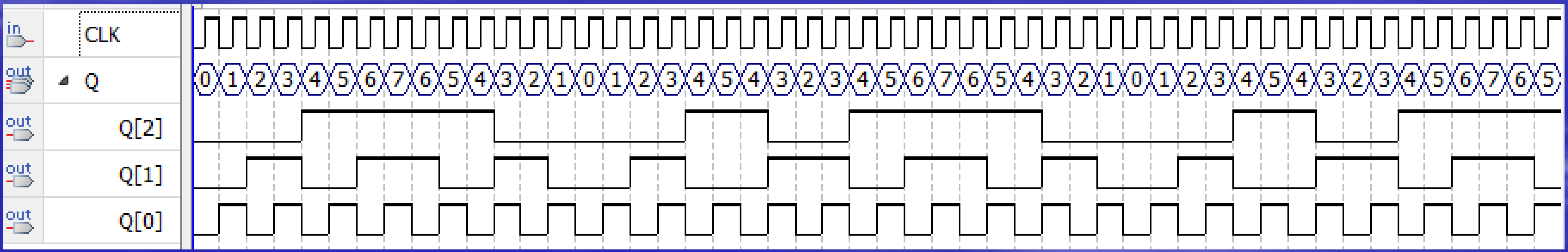
	Source State	Destination State	Condition
1	ROFF	ROFF	(!START) + (START).(ifsm)
2	ROFF	RTIME	(START).(!ifsm)
3	RON	RON	(!ifsm)
4	RON	ROFF	(ifsm)
5	RTIME	RON	(\ifsm:cntr[27]).(START).(!ifsm)
6	RTIME	ROFF	(!START) + (START).(ifsm)
7	RTIME	RTIME	(!\ifsm:cntr[27]).(START).(!ifsm)

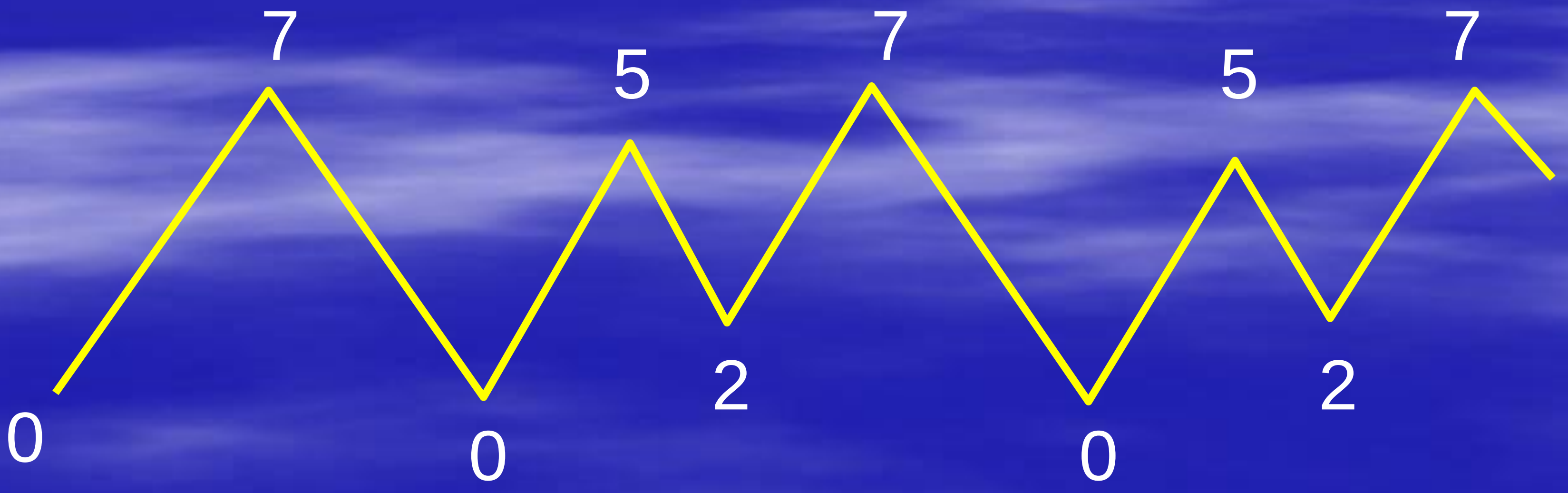
Demo

Simulation: `variable cntr: unsigned(4 downto 0):=(others=>'0');`



A more complex example - signal generator

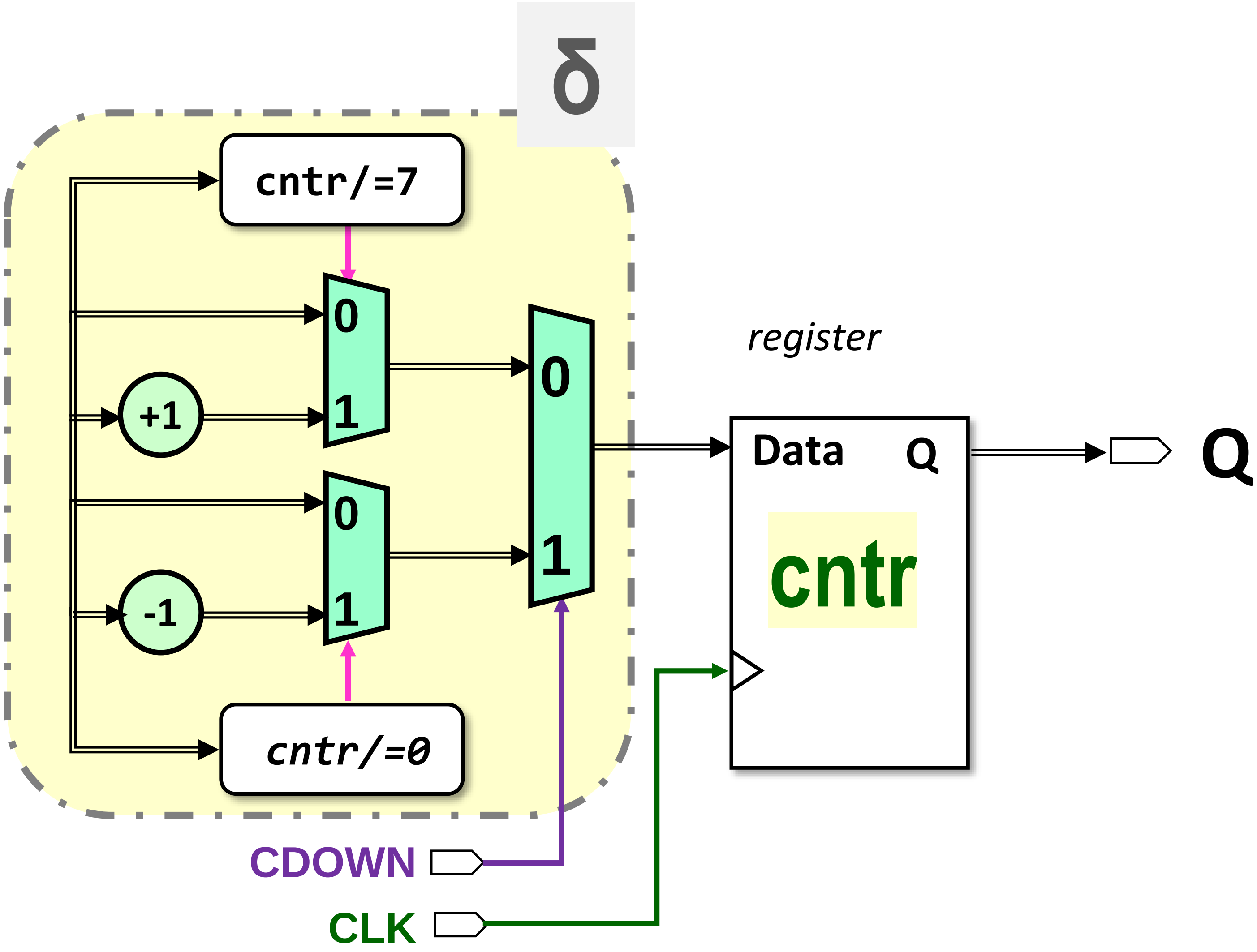




Designing the counter as an automaton ?

- It is a possible solution, but an unnecessarily complicated one.
- It is easier to describe the counter using an adder, either +1 or -1.
- If we use up and down counting at the same time, then the smallest number of LEs comes out when switching the multiplexer between the +1 and -1 adder output.
- The +1 and -1 adder has a simple structure, Quartus minimizes two of these better than one full adder with switched +1 / -1 input.

Step 1: Counter up/down 1/4



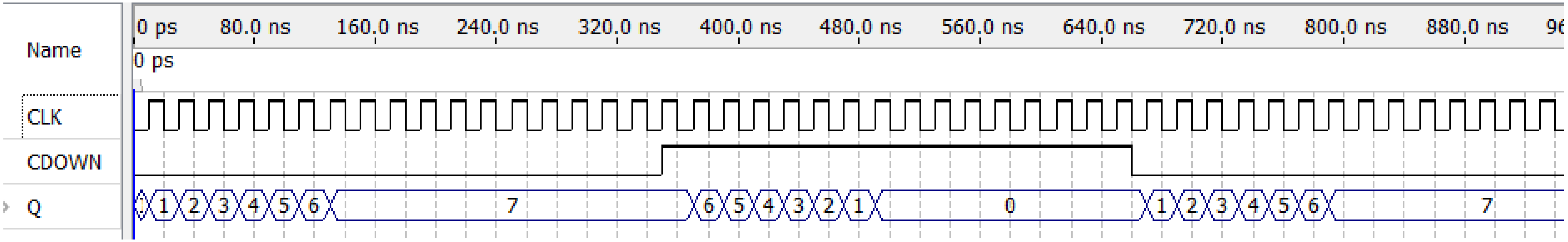
Step 1: Counter up/down 2/4

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity CounterUpDown is
port(CLK, CDOWN : in std_logic;Q : out std_logic_vector(2 downto 0));
end entity;
architecture rtl of CounterUpDown is
begin
  icount: process(CLK)
  variable cntr : integer range 0 to 7:=0;
  begin
   -- next page
  end process;
end architecture;
```

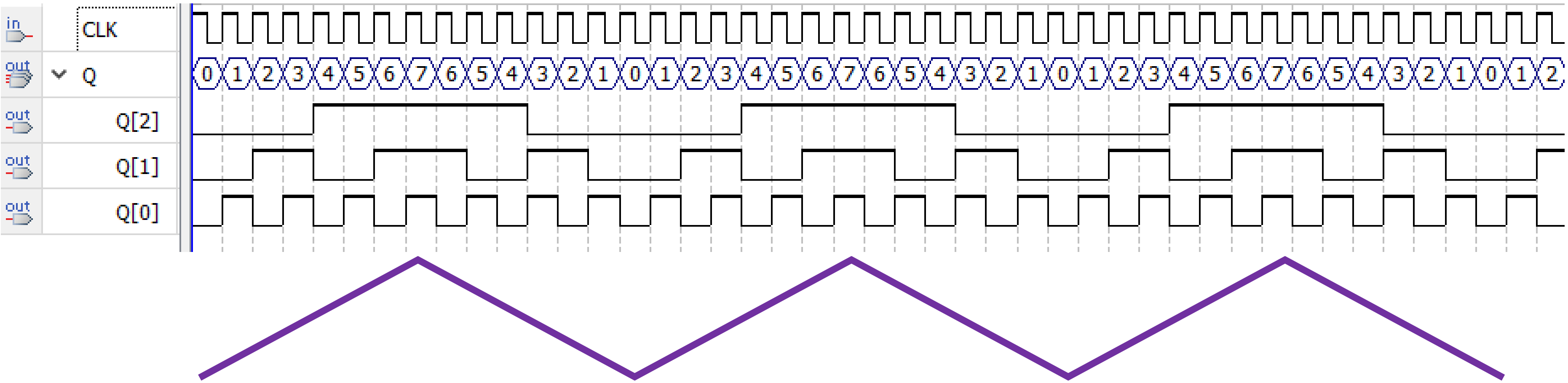
Step 1: Counter up/down 3/4

```
icount: process(CLK)
 variable cntr : integer range 0 to 7:=0; -- range important
 begin
   if rising_edge(CLK) then
      if CDOWN='1' then
         if cntr/=0 then cntr:=cntr-1; end if;
     else
        if cntr/=7 then cntr:= cntr+1; end if;
     end if;
   end if;
   Q<=std_logic_vector(to_unsigned(cntr, Q'LENGTH));
 end process;
```

Counter up/down 4/4



Easy modification to a triangular signal generator



Modification to triangle signal generator

δ - CounterUpDown

CDOWN is an input signal

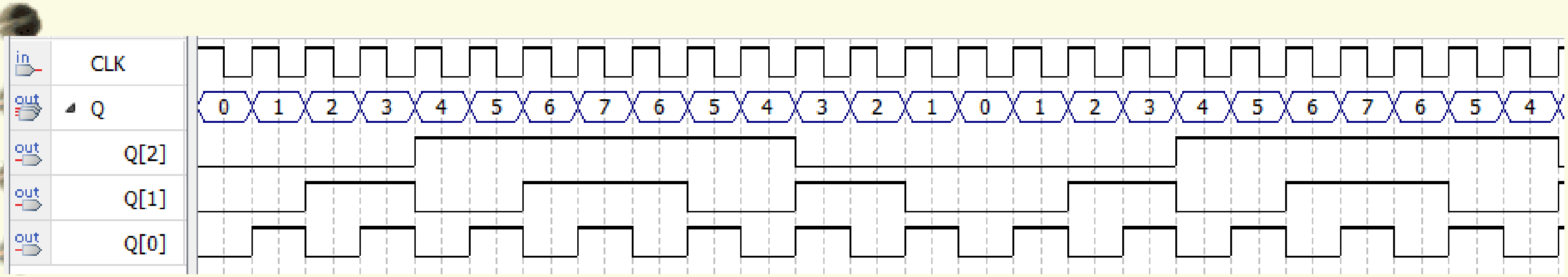
```
 if CDOWN='1' then
      if cntr/=0 then cntr:=cntr-1; end if;
else
     if cntr/=7 then cntr:= cntr+1; end if;
end if;
```

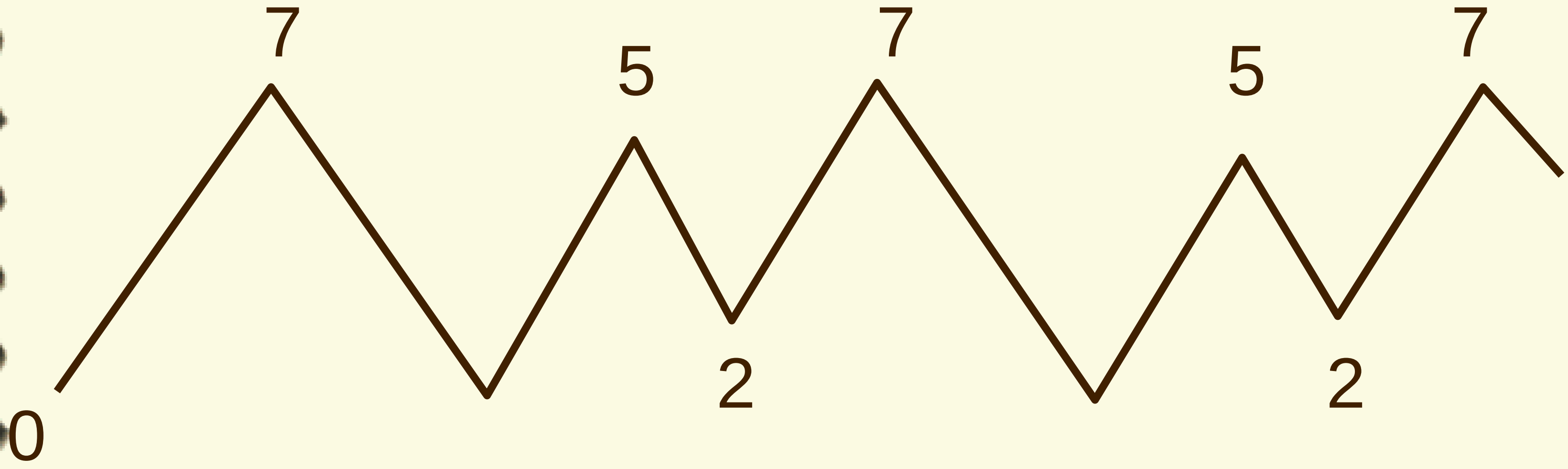
δ - TriangleCounter

***cdown** is a variable*

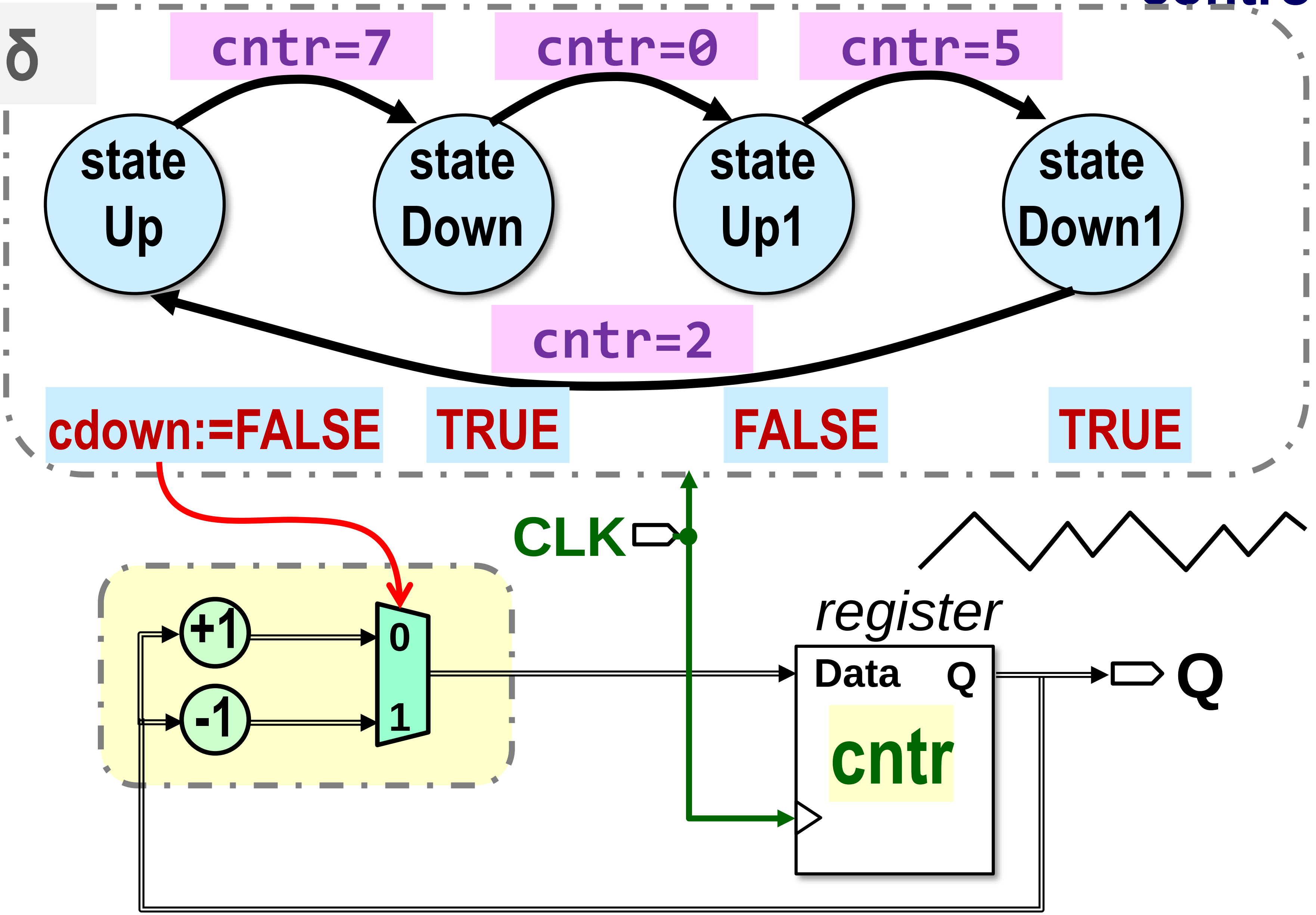
```
if cdown then if cntr=0 then cdown:=false; end if;
else if cntr=7 then cdown:=true; end if;
end if;
if cdown then cntr:=cntr-1; else cntr:=cntr+1; end if;
```

Complex triangular signal counter



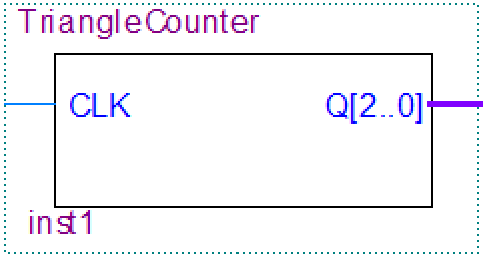


Inappropriate direct up and down counter control



VHDL entities

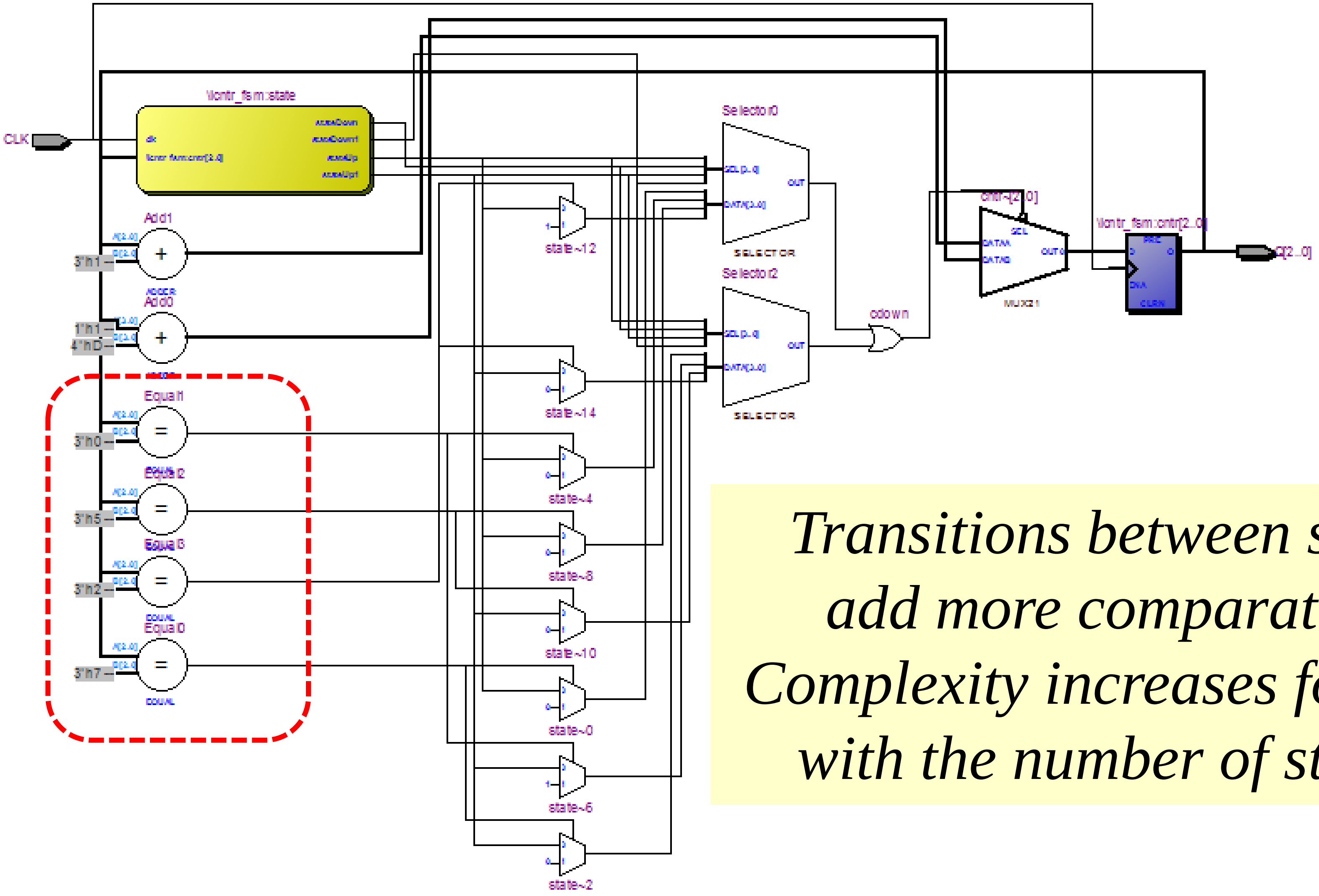
```
library ieee;use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity TriangleCounter is
port
( CLK : in std_logic;
   Q : out std_logic_vector(2 downto 0));
end entity;
```



Architecture

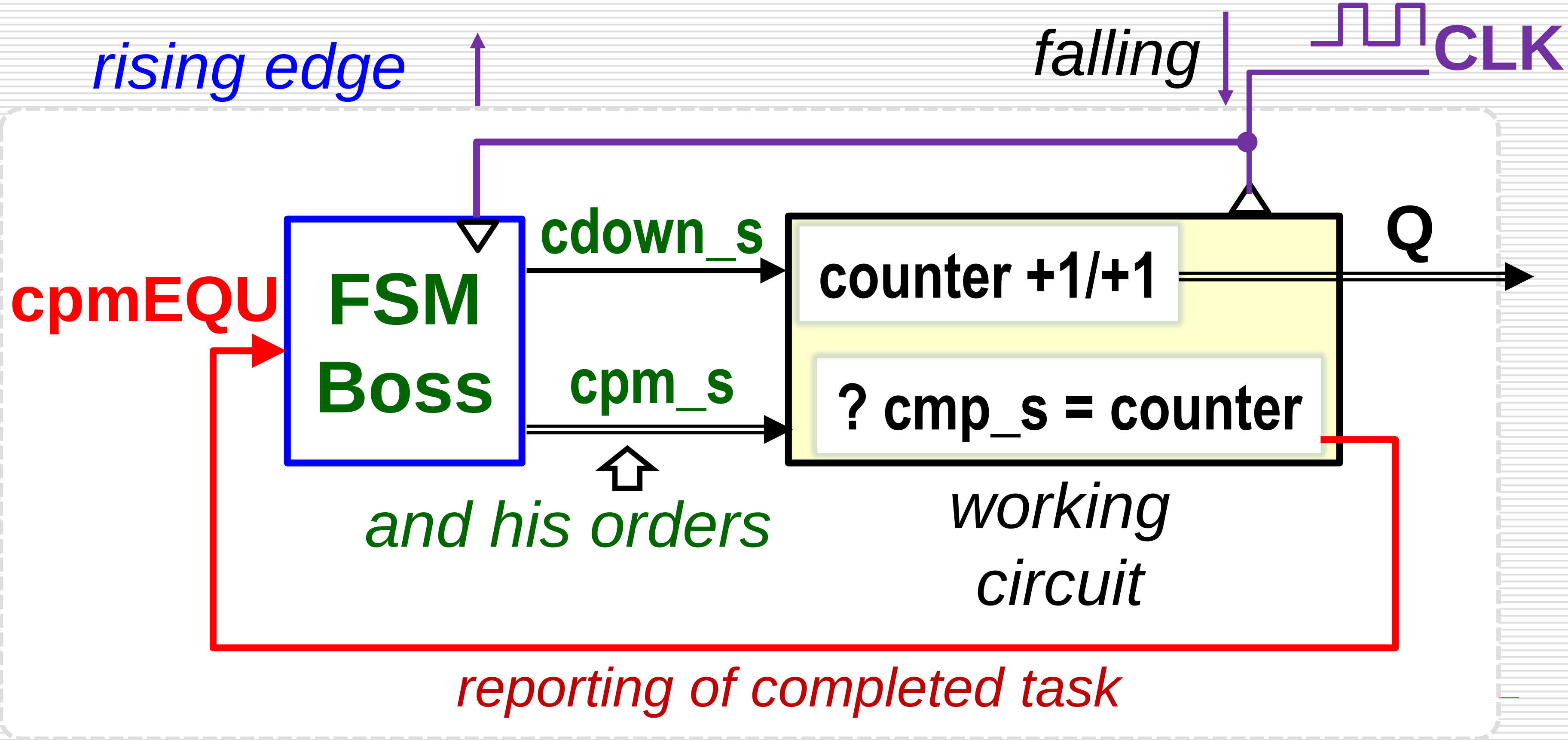
```
architecture rtl2wrong of TriangleCounter is
begin
icntr_fsm: process(CLK) -- explosive increasing of LE consumption with states
  variable cntr : integer range 0 to 7:=0;
  type state_t is (stateUp, stateDown, stateUp1, stateDown1);
  variable state: state_t:=stateUp;
  variable cdown : boolean:=FALSE;
  begin if rising_edge(CLK) then
      case state is -- delta and omega function of FSM
        when stateUp => cdown:=FALSE;      if cntr=7 then state:=stateDown; end if;
        when stateDown => cdown:=TRUE;     if cntr=0 then state:=stateUp1; end if;
        when stateUp1 => cdown:=FALSE;     if cntr=5 then state:=stateDown1; end if;
        when stateDown1 => cdown:=TRUE;    if cntr=2 then state:=stateUp; end if;
      end case;
     if cdown then cntr:=cntr-1; else cntr:=cntr+1; end if;
  end if;
  Q<=std_logic_vector(to_unsigned(cntr, Q'LENGTH));
 end process;
end architecture;
```

Each additional condition adds LEs



Transitions between states add more comparators. Complexity increases fourfold with the number of states.

Again, we'd promote the FSM to a boss who only manages others.



How do we create a great FSM?

- We divide the circuit into two processes connected by signals.
- The main (boss) FSM only sends the commands.
- The counter only informs that the count is done, in this case, achieving equality with the specified value.
- We will use a two-phase clock, which avoids entering future values into the conditions:
  - ❑ the counter will work on the descending edge
  - ❑ the boss's FSM on the waterfront.

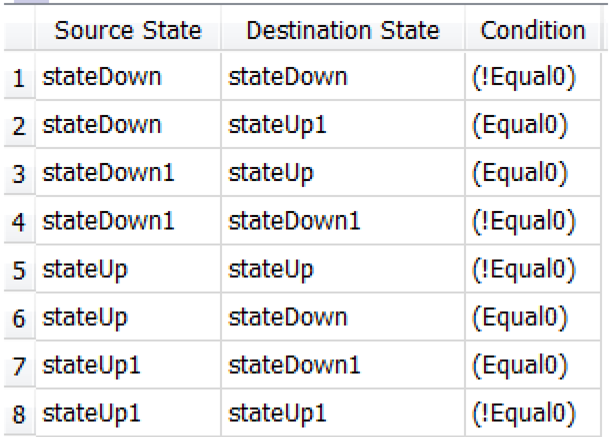
TriangleCounter V3 - 1/3 counter

```
architecture rtl3 of TriangleCounter is
signal cmp_s : integer range 0 to 7:=0;
signal cmpEQ, cdown_s : boolean:=FALSE; -- connection to FSM
begin
icntr : process(CLK, cmp_s) -- stand alone counter
  variable cntr : integer range 0 to 7:=0;
  begin
     if falling_edge(CLK) then
          if cdown_s then cntr:=cntr-1; else cntr:=cntr+1; end if;
     end if;
     Q<=std_logic_vector(to_unsigned(cntr, Q'LENGTH));
     cmpEQ <= (cntr=cmp_s);
   end process;
```

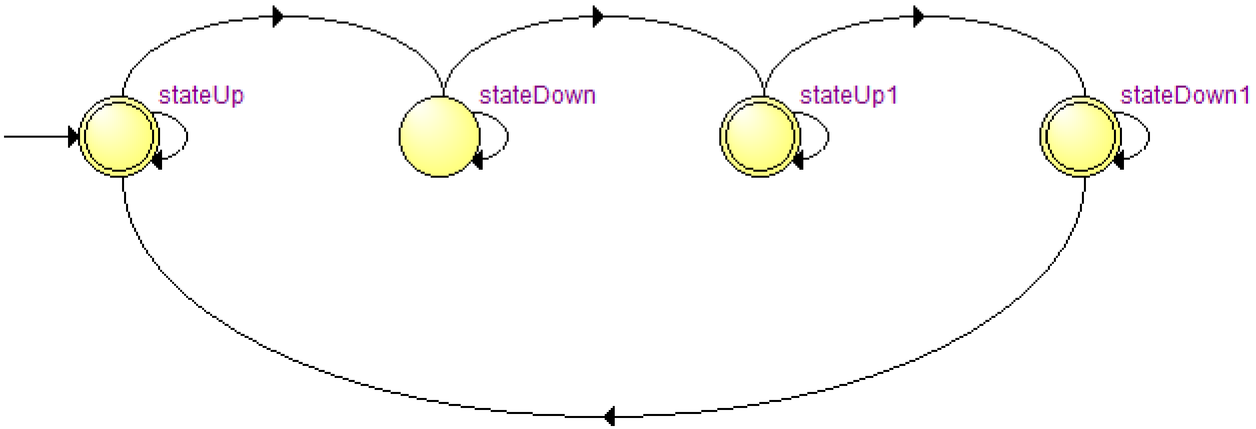
TriangleCounter V3 - FSM 2/3

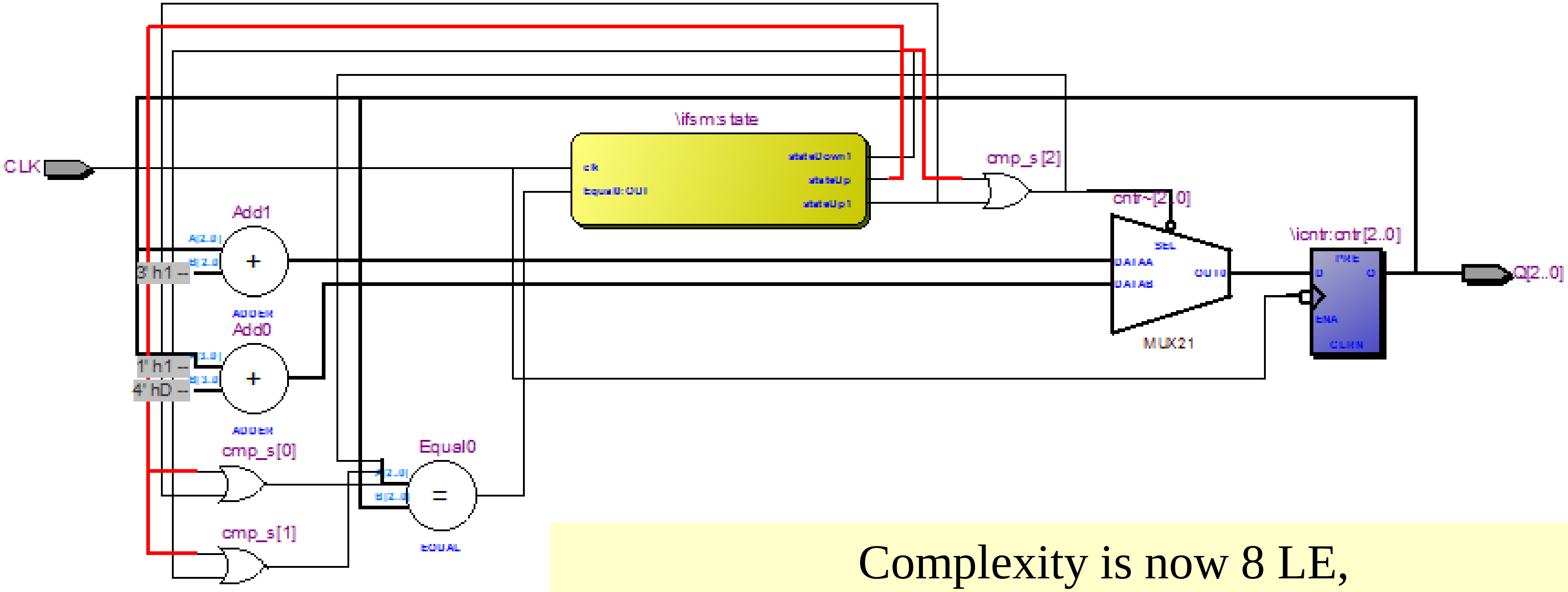
```
ifsm : process(CLK)
  type state_t is (stateUp, stateDown, stateUp1, stateDown1);
  variable state: state_t:=stateUp;
  begin if rising_edge(CLK) then
            if cmpEQ then -- delta function of FSM
              case state is
                  when stateUp => state:=stateDown;
                  when stateDown => state:=stateUp1;
                  when stateUp1 => state:=stateDown1;
                  when stateDown1 => state:=stateUp;
             end case;
           end if;
       end if; -- rising_edge
       case state is -- omega function of FSM
           when stateUp => cdown_s<=FALSE; cmp_s<=7;
           when stateDown => cdown_s<=TRUE; cmp_s<=0;
           when stateUp1 => cdown_s<=FALSE; cmp_s<=5;
           when stateDown1 => cdown_s<=TRUE; cmp_s<=2;
      end case;
end process; end architecture;
```

TriangleCounter V3 3/3 - Quartus RTL Viewer



	Source State	Destination State	Condition
1	stateDown	stateDown	(!Equal0)
2	stateDown	stateUp1	(Equal0)
3	stateDown1	stateUp	(Equal0)
4	stateDown1	stateDown1	(!Equal0)
5	stateUp	stateUp	(!Equal0)
6	stateUp	stateDown	(Equal0)
7	stateUp1	stateDown1	(Equal0)
8	stateUp1	stateUp1	(!Equal0)





Complexity is now 8 LE,
2 times the number of states → 2 times the complexity.

Testbench 1/3

```
library ieee; use ieee.std_logic_1164.all; use ieee.numeric_std.all;
entity tb_TriangleCounter is
end entity;

architecture rtl of tb_TriangleCounter is
-- definitions
    signal CLK : std_logic:='0';
    signal STOP : boolean := FALSE;
    signal Q1, Q2, Q3: std_logic_vector(2 downto 0);
    component TriangleCounter
      port(CLK : in std_logic; Q : out std_logic_vector(2 downto 0));
    end component TriangleCounter;
begin -- architecture

end architecture;
```

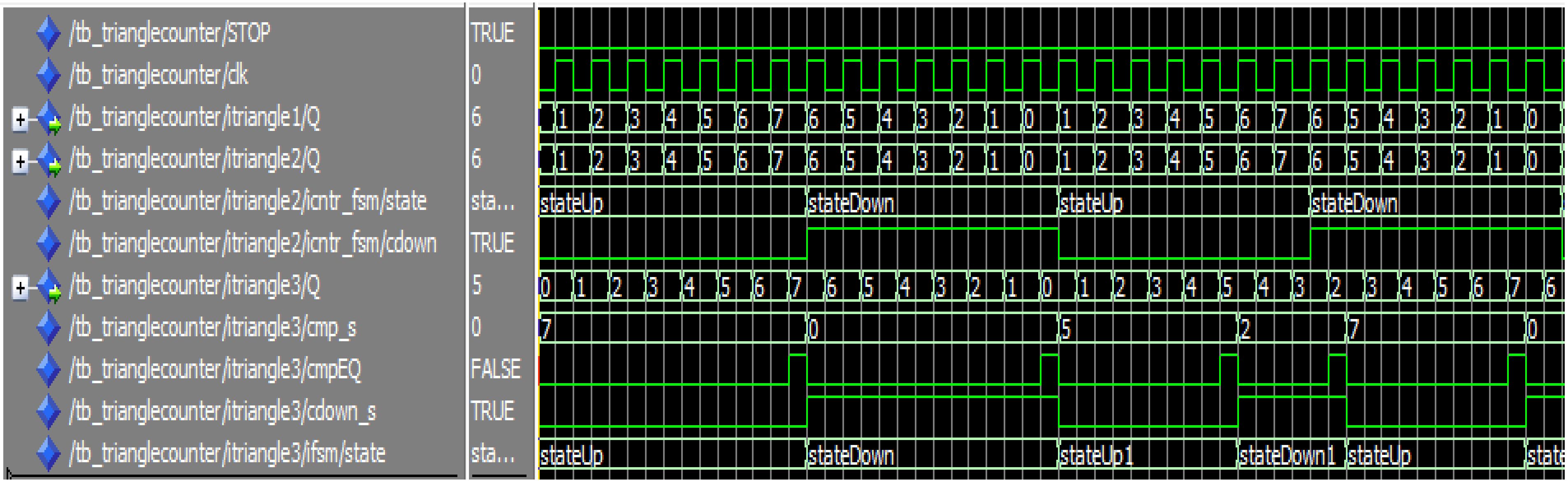
Testbench 2/3

```
begin -- architecture
    CLK<= not CLK after 10 ns;
    STOP<= FALSE, TRUE after 1 us;

-- the insertion of the instance map with selection of architecture
    itriangle1 : entity work.TriangleCounter(rtl1) port map(CLK, Q1);
    itriangle2 : entity work.TriangleCounter(rtl2) port map(CLK, Q2);
    itriangle3 : entity work.TriangleCounter(rtl3) port map(CLK, Q3);

    assert not STOP report "Done OK" severity failure;

end architecture;
```



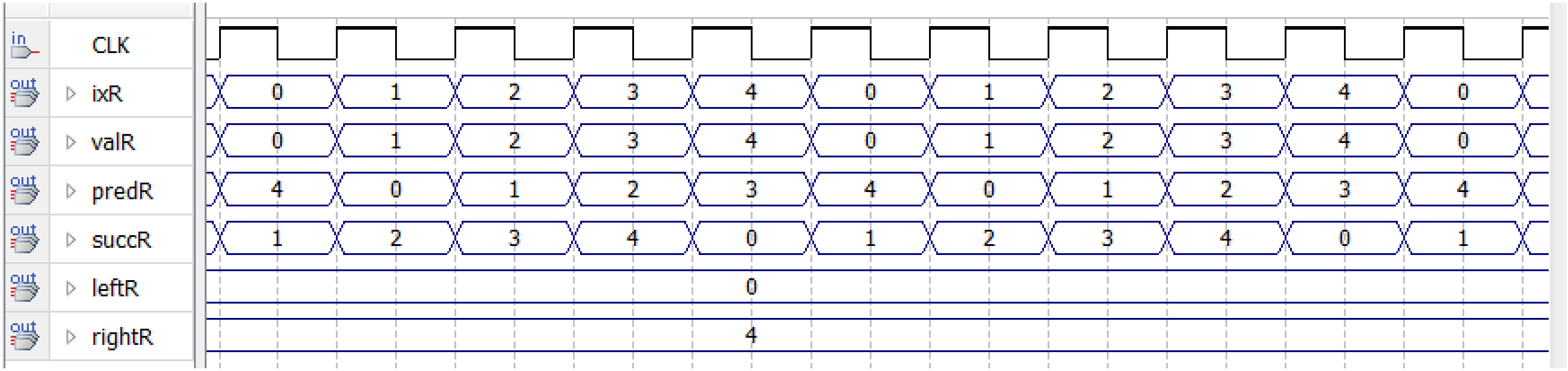
```
/tb_trianglecounter/STOP                      TRUE
/tb_trianglecounter/clk                       0
/tb_trianglecounter/itriangle1/Q              6
/tb_trianglecounter/itriangle2/Q              6
/tb_trianglecounter/itriangle2/icntr_fsm/state  sta...
/tb_trianglecounter/itriangle2/icntr_fsm/cdown  TRUE
/tb_trianglecounter/itriangle3/Q              5
/tb_trianglecounter/itriangle3/cmp_s          0
/tb_trianglecounter/itriangle3/cmpEQ          FALSE
/tb_trianglecounter/itriangle3/cdown_s        TRUE
/tb_trianglecounter/itriangle3/ifsm/state     sta...
```

State attributes 1/2

```
library ieee; use ieee.std_logic_1164.all;use ieee.numeric_std.all;
entity AttrOfStateType is port ( CLK : in std_logic; ixR,valR,predR,succR,leftR,rightR:out unsigned(2 downto 0));
end;
architecture rtl of AttrOfStateType is
begin
 process(CLK)
    type state_t is (s0, s1, s2, s3, s4);
    variable state: state_t:=s0;
    function State2uint(st:state_t) return unsigned is
    begin return to_unsigned(state_t'POS(st),3); -- order number of state
    end function;
    variable ix:unsigned(ixR'RANGE);
 begin
      if rising_edge(CLK) then
           ix:=State2uint(state); ixR<=ix; -- the order number
           valR<=State2uint(state_t'VAL(to_integer(ix))); -- state from the order
           predR<=State2uint(state_t'PRED(state)); -- the previous state
           state:=state_t'SUCC(state); succR<=State2uint(state); -- next state
      end if;
leftR<=State2uint(state_t'LEFT); -- leftmost state in the definition
rightR<=State2uint(state_t'RIGHT); -- rightmost state in the definition
 end process; end architecture;
```

State attributes 2/2

```
type state_t is (s0, s1, s2, s3, s4);
```


in CLK
out ixR 0 1 2 3 4 0 1 2 3 4 0
out valR 0 1 2 3 4 0 1 2 3 4 0
out predR 4 0 1 2 3 4 0 1 2 3 4
out succR 1 2 3 4 0 1 2 3 4 0 1
out leftR 0
out rightR 4

```
if rising_edge(CLK) then
    ix:=State2uint(state); ixR<=ix; -- the order number state_t'POS(state)
    valR<=State2uint(state_t'VAL(to_integer(ix))); -- state from the order
    predR<=State2uint(state_t'PRED(state)); -- the previous state
    state:=state_t'SUCC(state); succR<=State2uint(state); -- the next state
end if;
leftR<=State2uint(state_t'LEFT); -- the leftmost state in the definition
rightR<=State2uint(state_t'RIGHT); -- the rightmost state in the definition
```

Homework

for the next lecture

What does the FSM do?

```
library ieee; use ieee.std_logic_1164.all;use ieee.numeric_std.all;
entity DetectorABC is port ( CLK, swa, swb, swc, ACLRN : in std_logic; ixDebug: out unsigned(2 downto 0); q : out std_logic);
end;
architecture rtl of DetectorABC is
begin
   process (CLK, ACLRN) -- state register
   type state_t is (s0, s1, s2, s3, s4);
   variable state: state_t:=s0;
   variable x : integer range 0 to 7;
   function GoS0(stnow:state_t; x, cmp:integer) return state_t is
   begin if x=cmp then return stnow; else return s0; end if;
   end function;
begin
  if ACLRN='0' then state:=s0;
  elsif rising_edge(CLK) then
    x:=to_integer(unsigned(std_logic_vector'(swc&swb&swa)));
    if x=0 then state:= s1;
    else case state is
                when s1 => if x=1 then state:=s2; else state:=s0; end if;
                when s2 => if x=3 then state:=s3; else state:=GoS0(state, x, 1); end if;
                when s3 => if x=7 then state:=s4; else state:=GoS0(state, x, 3); end if;
                when s4 => state:=GoS0(state, x, 7);
                when others =>
        end case;
    end if;
  end if;
  if state=s4 then q<='1'; else q<='0'; end if;
  ixDebug <= to_unsigned(state_t'POS(state),ixDebug'LENGTH); -- t'POS(state) returns integer order number of state
end process; end architecture;
```

Instructions: the RTL scheme is clear this time, it is an FSM,
so make a transition table or diagram of it.