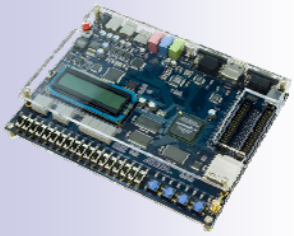
Struktury počítačových systémů

Přednáší: Richard Šusta

10. listopadu 2011 verze 1.0



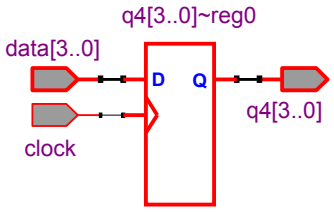
ČVUT-FEL v Praze – kód předmětu A0B35SPS

Answers

to 4bit register questions...

Primary program

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock : in std_logic;
        data : in std_logic_vector(3 downto 0);
        q4 : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock)
  begin
    if clock = '1' and clock'event then
       q4 <= data;
    end if;
   end process;
 end reg4_arch;
```

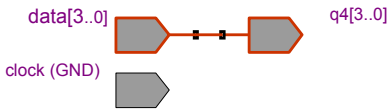


4bit synchronous register samples data inputs on rising edge of clock and stores them in q4 outputs.

Primary program without clock-'if'

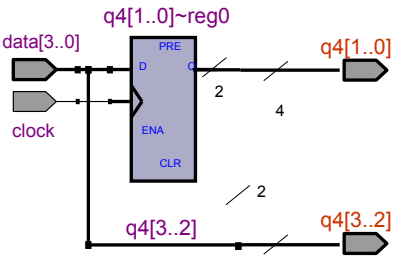
We obtain only direct connection of inputs to outputs after removing 'if'-clock statement.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock : in std_logic;
        data : in std_logic_vector(3 downto 0);
        q4 : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock)
  begin
   -- if clock = '1' and clock'event then
       q4 <= data;
   --  end if;
   end process;
 end reg4_arch;
```



Inside and outside clock-if

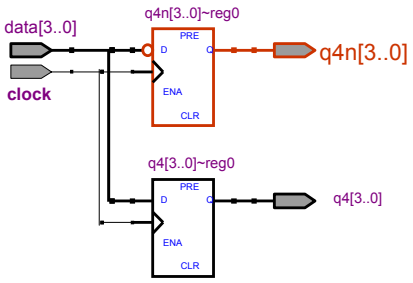
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock : in std_logic;
          data : in std_logic_vector(3 downto 0);
          q4 : out std_logic_vector(3 downto 0));
  end reg4;

architecture reg4_arch of reg4 is
begin process(clock)
  begin
    if clock = '1' and clock'event then
        q4(1 downto 0) <= data(1 downto 0);
    end if;
    q4(3 downto 2) <= data(3 downto 2);
   end process;
 end reg4_arch;
```



Primary program +additional negative output

Concurrent assigments <= inside 'if'-clock statement of the process are compiled to registered signals.

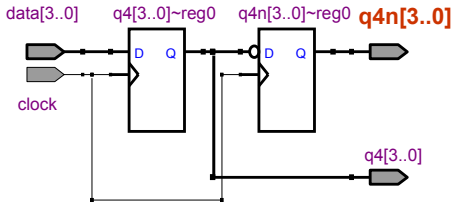
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock : in std_logic;
          data : in std_logic_vector(3 downto 0);
          q4 : out std_logic_vector(3 downto 0);
          q4n : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock)
  begin
    if clock = '1' and clock'event then
       q4 <= data;
      q4n <= not data;
    end if;
   end process;
 end reg4_arch;
```



Primary program +additional negative output

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock : in std_logic;
        data : in std_logic_vector(3 downto 0);
        q4 : buffer std_logic_vector(3 downto 0);
        q4n : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock)
  begin
    if clock = '1' and clock'event then
      q4 <= data;
      q4n <= not q4;
    end if;
  end process;
 end reg4_arch;
```

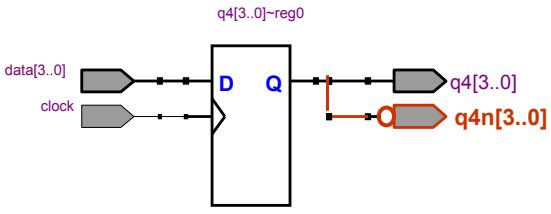
Concurrent assigments <= inside 'if'-clock statement of the process are compiled to registered signals. Buffer type of output signal allows reading.



Primary program +negative output

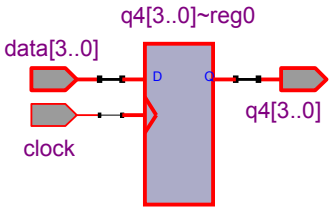
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock : in std_logic;
        data : in std_logic_vector(3 downto 0);
        q4 : buffer std_logic_vector(3 downto 0);
        q4n : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock)
  begin
   if clock = '1' and clock'event then
       q4 <= data;
   end if;
   q4n <= not q4;
 end process;
 end reg4_arch;
```

Concurrent assigments <= outside of 'if'-clock statement of the process are compiled by combinational logic.



We return to primary program

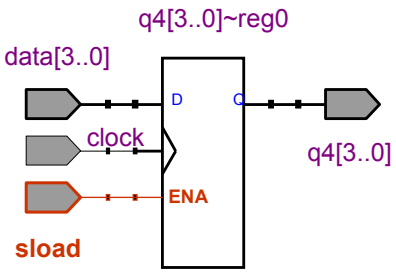
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock : in std_logic;
        data : in std_logic_vector(3 downto 0);
        q4 : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock)
  begin
    if clock = '1' and clock'event then
       q4 <= data;
    end if;
   end process;
 end reg4_arch;
```



Adding synchronous load (sload)

Synchronous load (sload) must be tested inside 'if'-clock statement and it is not in sensitivity list because it effects outputs only synchronously with the clock.

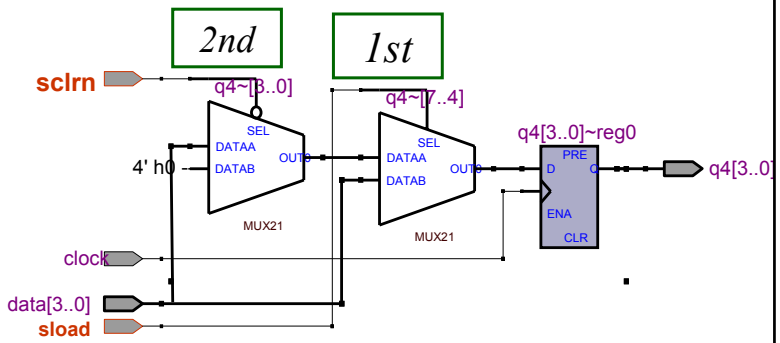
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock, sload : in std_logic;
        data : in std_logic_vector(3 downto 0);
        q4 : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock)
  begin
    if clock = '1' and clock'event then
       if sload = '1' then   q4 <= data;
       end if;
    end if;
   end process;
 end reg4_arch;
```



2 synchronous inputs – clear and load

```
port (clock, sload, sclrn : in std_logic;
*******
begin process(clock)
begin  if clock = '1' and clock'event then
            if sload = '1' then   q4 <= data;
            elsif sclrn = '0' then   q4 <= "0000"; end if;
        end if;
end process;
```

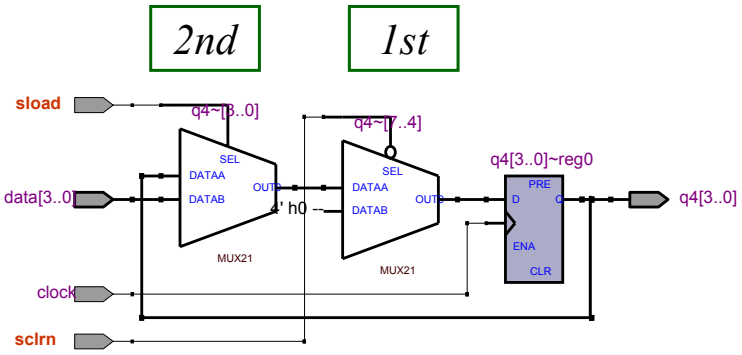
Synchronous clear (sclrn) is the second in 'if' conditions, thus sload has priority over sclrn.



Priority synchronous clear

```
port (clock, sload, sclrn : in std_logic;
*******
begin process(clock)
begin  if clock = '1' and clock'event then
            if sclrn = '0' then   q4 <= "0000";
            elsif sload = '1' then   q4 <= data; end if;
        end if;
end process;
```

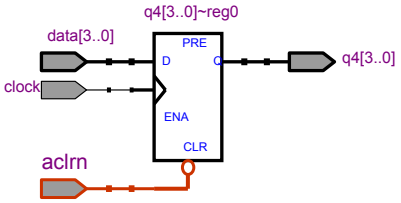
Synchronous clear (sclrn) is now before sload in 'if' , thus it has priority over sload



Adding asynchronous clear (aclrn)

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock, aclrn : in std_logic;
          data : in std_logic_vector(3 downto 0);
          q4 : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock, aclrn)
  begin
        if aclrn = '0' then   q4 <="0000";
        elsif clock = '1' and clock'event then
                q4 <= data;
        end if;
    end if;
   end process;
 end reg4_arch;
```

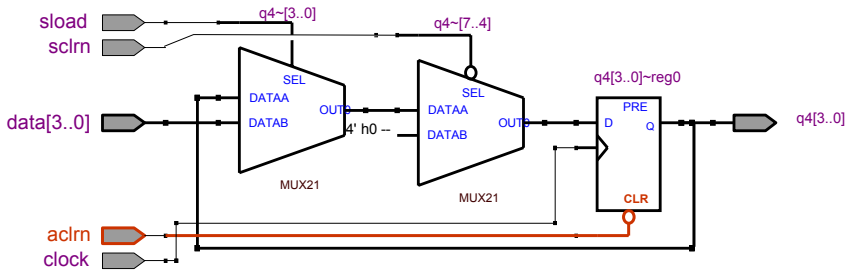
Asynchronous clear (aclrn) must be tested before 'if'-clock statement and it should be in sensitivity list. All asynchronous inputs effect asynchronous inputs of internal flip-flops.



Asynchronous and synchronous inputs

```
port (clock, sload, sclrn, aclrn : in std_logic;
*******
begin process(clock, aclrn )
begin if aclrn= '0' then q4 <= "0000";
       elsif clock = '1' and clock'event then
            if sclrn = '0' then   q4 <= "0000";
            elsif sload = '1' then   q4 <= data; end if;
        end if;
end process;
```

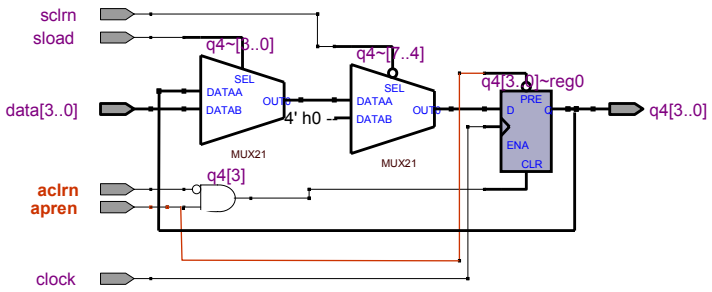
Asynchronous clear (aclrn) is before clock-'if' statement and in the sensitivity list



Adding asynchronous preset

```
port (clock, sload, sclrn, aclrn, apren : in std_logic;
*******
begin process(clock, aclrn, apren)
begin if apren = '0' then q4 <= "1111";
      elsif aclrn = '0' then q4 <= "0000";
      elsif clock = '1' and clock'event then
          if sclrn = '0' then   q4 <= "0000";
          elsif sload = '1' then   q4 <= data; end if;
      end if;
end process;
```

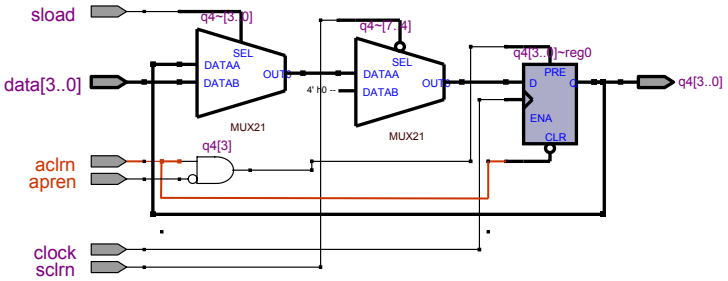
Asynchronous preset (apren) is before aclrn thus it has priority over aclrn



Adding asynchronous preset

```
port (clock, sload, sclrn, aclrn, apren : in std_logic;
*******
begin process(clock, aclrn, apren)
begin if aclrn= '0' then q4 <= "0000";
      elsif apren= '0' then q4 <= "1111";
      elsif clock = '1' and clock'event then
          if sclrn = '0' then   q4 <= "0000";
          elsif sload = '1' then   q4 <= data; end if;
       end if;
end process;
```

Asynchronous clear (aclrn), is the first in 'if' and it has now priority over apren



Wrong asynchronous input

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock, aclrn : in std_logic;
        data : in std_logic_vector(3 downto 0);
        q4 : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock, aclrn)
  begin
      if clock = '1' and clock'event then
            q4 <= data;
      elsif aclrn = '0' then   q4 <="0000";
      end if;
    end if;
   end process;
end reg4_arch;
```

Asynchronous clear (aclrn) must be tested before 'if'-clock statement of the process

Error (10818): Can't infer register for "q4[3..0]" at reg4.vhd(13) because it does not hold its value outside the clock edge
Error (10822): HDL error at reg4.vhd(13): couldn't implement registers for assignments on this clock edge

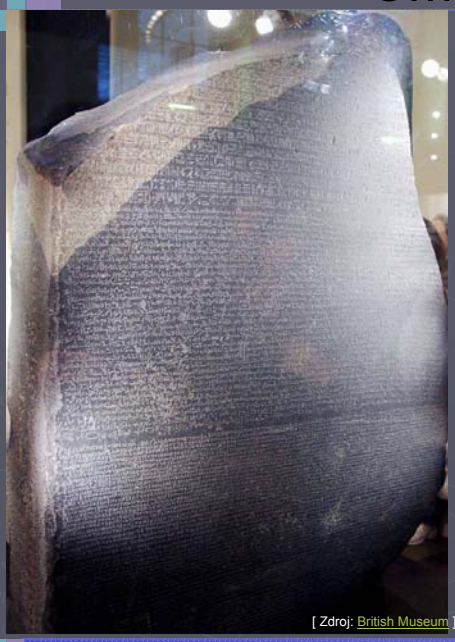
Wrong double clock dependency

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 entity reg4 is
  port (clock: in std_logic;
        data : in std_logic_vector(3 downto 0);
        q4 : out std_logic_vector(3 downto 0));
  end reg4;
architecture reg4_arch of reg4 is
begin process(clock)
  begin
      if clock = '1' and clock'event then
            q4 <= data;
      elsif clock = '0' and clock'event then
            q4 <= data;
      end if;
    end if;
   end process;
end reg4_arch;
```

Asynchronous clear (aclrn) must be tested before 'if'-clock statement of the process

Error (10819): Netlist error at reg4.vhd(13): can't infer register for q4[0] because it changes value on both rising and falling edges of the clock
Error (10822): HDL error at reg4.vhd(13): couldn't implement registers for assignments on this clock edge

Omluva



[Zdroj: British Museum]

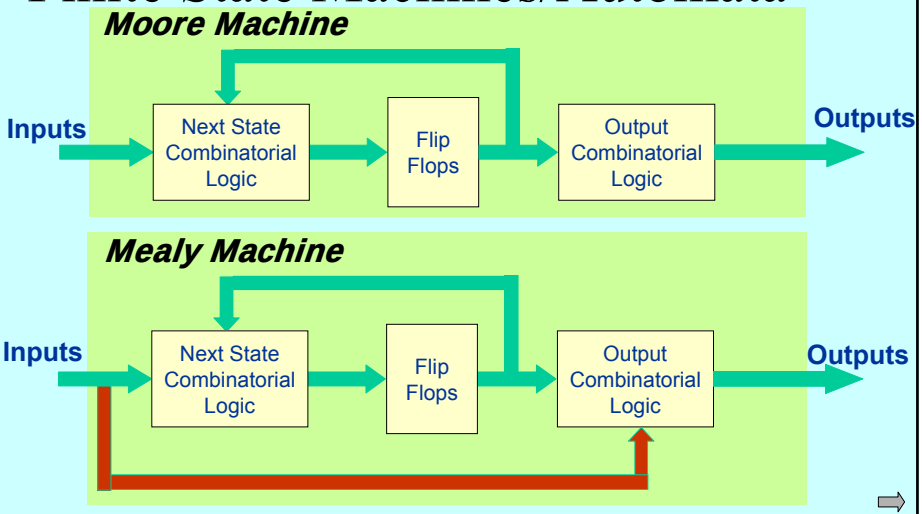
Následující část dnešní přednášky bude česko-anglická

** žel nemám vše přeložené do angličtiny pro zahraniční studenty, ale ani vše v češtině*

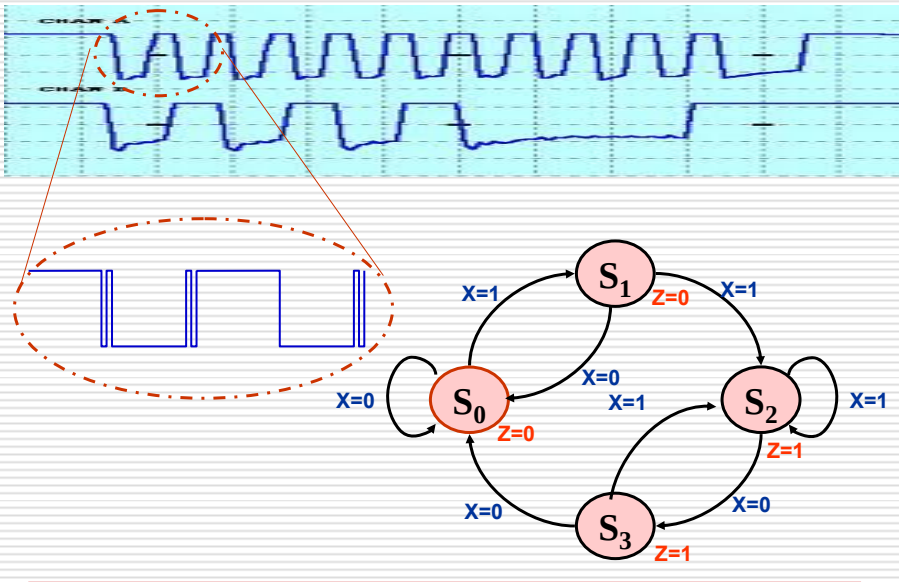
** nicméně jsem upřednostňoval češtinu, pokud byla volba.*

SPS

Mealy and Moore Finite State Machines/Automata



Example:Debouncer (cz:odrušení vstupu)



Definice konečného automatu

FSM – Finite State Machine

Uspořádaná šestice

$$M = < X, S, Z, \omega, \delta, s_0 \in S >$$

- **X** - *konečná množina všech* vstupních vektorů
- **Z** - *konečná množina všech* výstupních vektorů
- **S** - *konečná množina všech* vnitřních stavů
- δ - přechodová funkce - zobrazení δ: **X** x **S** -> **S**
- ω - výstupní funkce - zobrazení ω:
  - ω: **S** -> **Z** (Moore)
  - ω: **X** x **S** -> **Z** (Mealy)
- s_0 - počáteční stav $S_0 \in S$

Konečný automat - konečná množina vst., výst., a stavů

Úplně určený automat - zobrazení δ i ω def. pro každý (X x S)

Neúplně určený automat - nedefinovaná některá zobrazení δ, ω

Neiniciální automat - určen pěticí, chybí S_0

Deterministické automaty - δ i ω popsány deterministicky, nemění se

Stochastické automaty - δ i ω jsou pravděpodobnostní výrazy

Nedeterministické automaty - δ i ω popsány nedeterministicky

Nedeterministické chování není náhodné

- Deterministické:
  f(1) → 1 *vždy*
- Náhodné:
  f(1) → 1 v *50% případů,*
  f(1) → 2 *v ostatních situacích*
- Nedeterministické chování
  f(1) → 1 nebo f(1) → 2,
  ale nepovíme, kdy tomu tak bude.

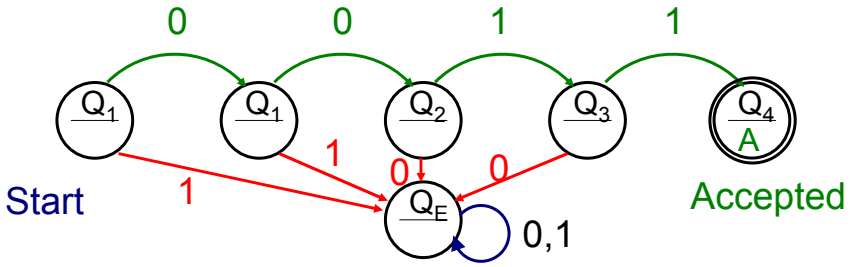
- Nedeterministické chování může vypadat deterministicky, náhodně, nebo i hůře

Example

from deterministic
to nondeterministic
automaton

Příklad: Synchronní kódový zámek

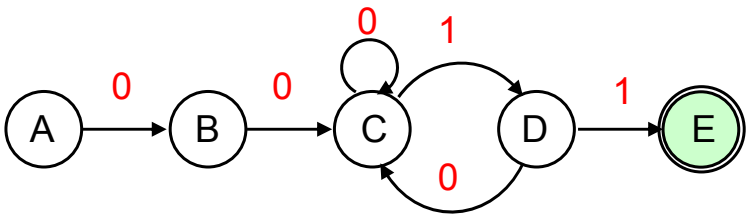
- Odemkne, pokud vstup začíná: 0011, jinak ne.
- Generuje výstup A (Accepted=přijato) ve stavu Q_5



Takový automat se někdy nazývá konečný akceptor, nebo rozpoznávací či klasifikační automat
(Finite State Acceptor or Acceptor Finite State Machine)

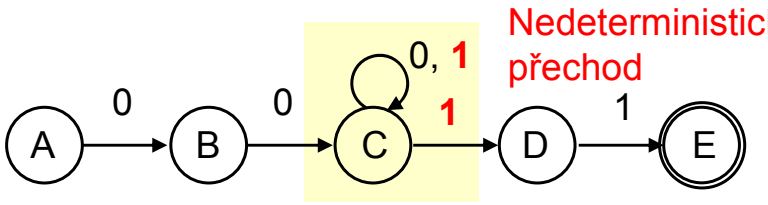
Příklad2: Synchronní kódový zámek

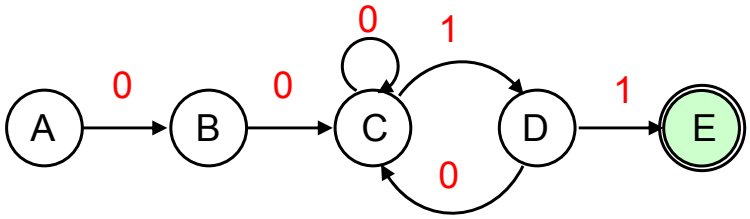
- Chceme nyní odemknout, pokud vstupní posloupnost začíná 00 a končí 11



Příklad: Synchronní kódový zámek

NFA – Nondeterministic Finite Automat/Acceptor

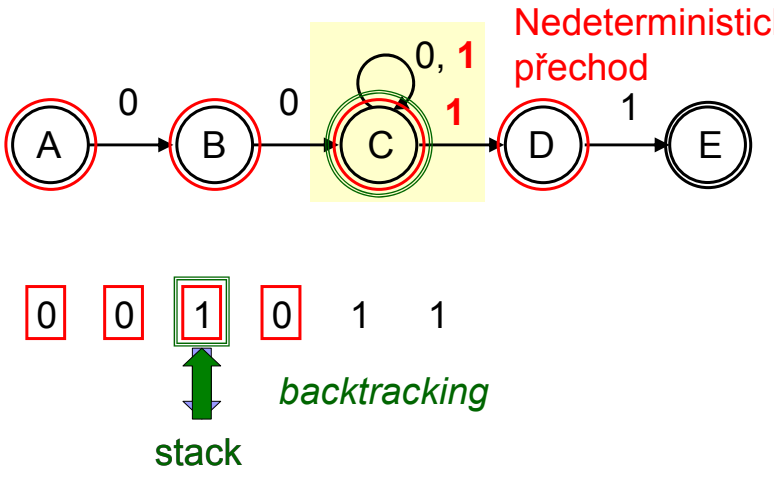




DFA – Deterministic Finite Automat/Acceptor

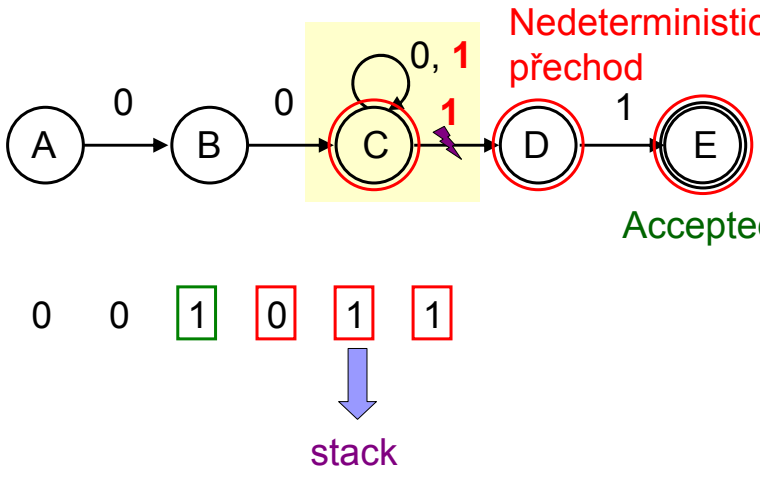
NFA- animace 1/2

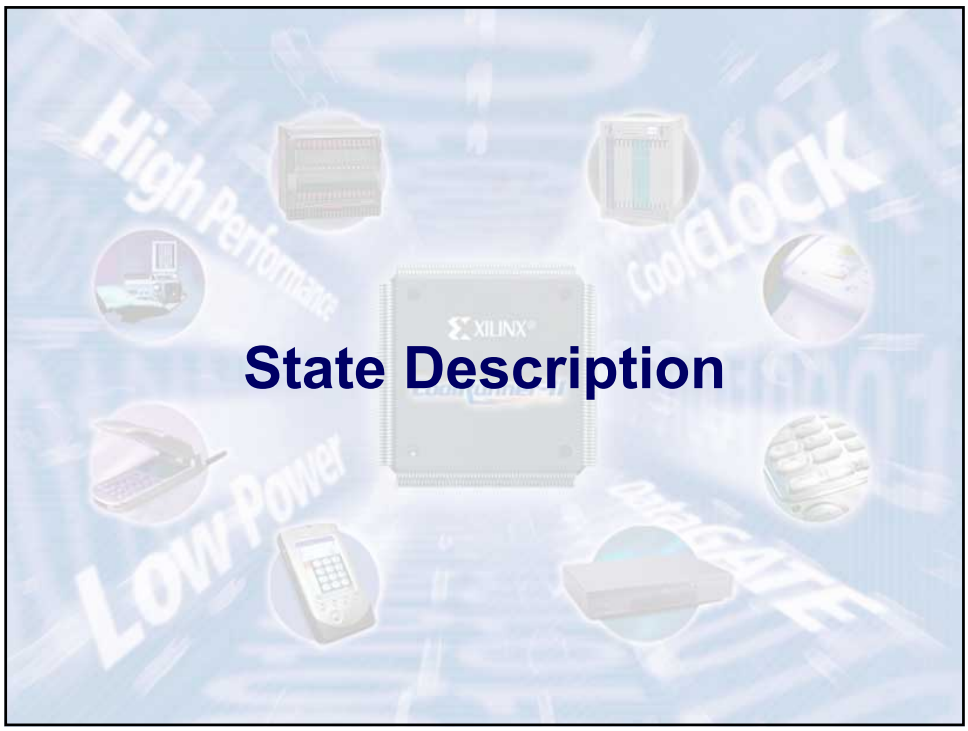
NFA – Nondeterministic Finite Automat/Acceptor



NFA – animace 2/2

NFA – Nondeterministic Finite Automat/Acceptor





State Description

Formy popisu chování automatu

1.Tabulka přechodů - 2 části - příští stav (δ), výstupy (ω)

sloupce - všechny vektory X - aktuální vstup x_i

řádky - všechny vnitřní stavy S - výchozí stav s_j

průsečík řádku a sloupce - následný stav s_k a souč. výstup z_m

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

δ

x_1	x_i	x_n
	$z_m = \omega(x_i s_j)$	

ω

MEALY

p.n -výst.v.

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

δ

Z
$z_j = \omega(s_j)$

ω

MOORE

p -výst.v.

z tab. lze zjistit : stabilní stav $s_j = \delta(x_i, s_j)$ - označují se kroužkem

nestabilní stav $s_k = \delta(x_i, s_j)$ $s_k \neq s_j$

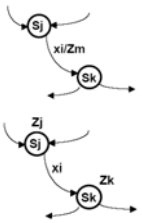
Graf přechodů

2. popis chování automatu orient. grafem (nahrazuje tab.přechodů)

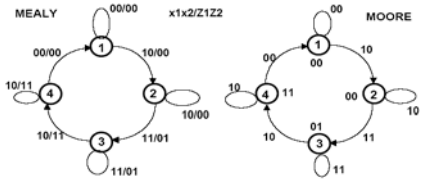
uzly - vnitřní stavy - označení s_j
orientované hrany - možné přechody - označení x_i / ω (x_i,s_j)

MEALY - automat ve stavu s_j má na vstupu vektor x_i, vysílá vektor z_m a přejde do stavu s_k

MOORE - automat ve stavu s_j (stab.stav) vysílá vektor z_j a po příchodu x_i přejde do stavu s_k.



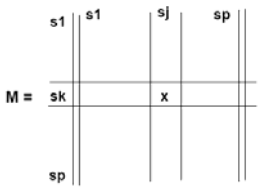
Př.:



Matice přechodů

3. - při velkém počtu stavů – přehlednost, při p – stavech matice p x p

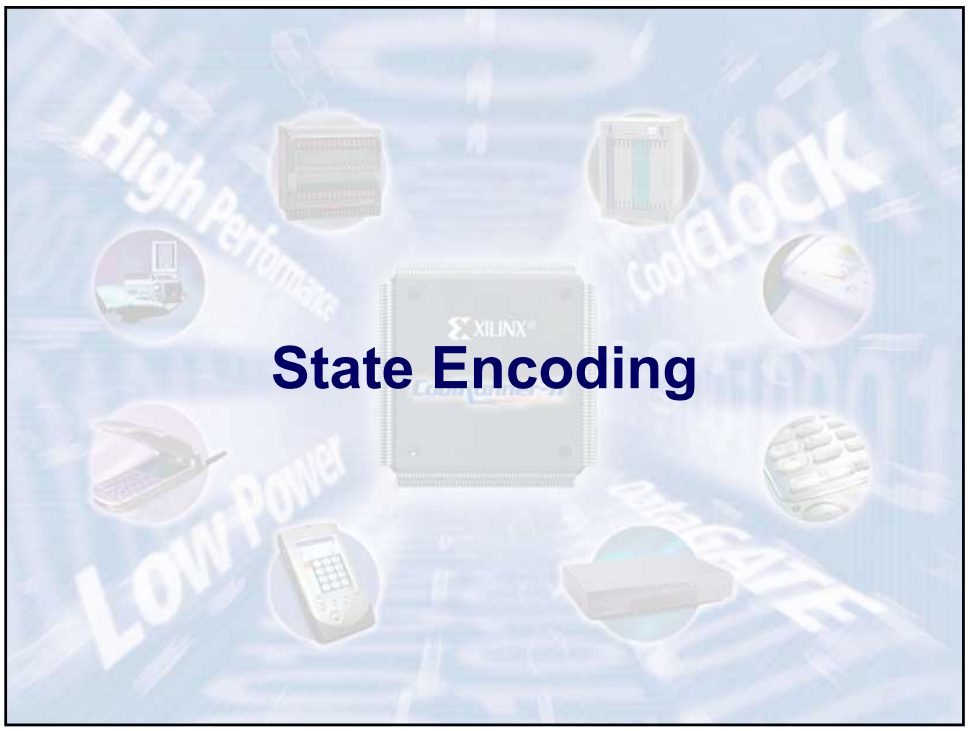
řádky - výchozí stav
sloupce - následný stav
průsečík - podmínky přechodu - x



Ms - matice přechodů (prvek - vektor x)
Mz - matice výstupů (prvek - vektor z)
Mx - matice vstupů (1/0 - způsobí přech.)

Př.: matice prechodu a matice výstupů

$$M_s = \begin{array}{c} \begin{matrix} & x_1 & x_2 \end{matrix} \\ \left\| \begin{matrix} 00 & 10 & -- & -- \\ -- & 10 & 11 & -- \\ -- & -- & 11 & 10 \\ 00 & -- & -- & 10 \end{matrix} \right\| \end{array} \qquad M_z = \begin{array}{c} \begin{matrix} & Z_1 & Z_2 \end{matrix} \\ \left\| \begin{matrix} 00 & 00 & -- & -- \\ -- & 00 & 01 & 01 \\ -- & -- & 01 & 01 \\ 00 & -- & -- & 11 \end{matrix} \right\| \end{array}$$



State Encoding

State Encoding Problem

- State Encoding Can Have a Big Influence on Optimality of the FSM Implementation
  - No methods other than checking all possible encodings are known to produce optimal circuit
  - Feasible for small circuits only
- Using Enumerated Types for States in VHDL Leaves Encoding Problem for Synthesis Tool

Types of State Encodings (1)

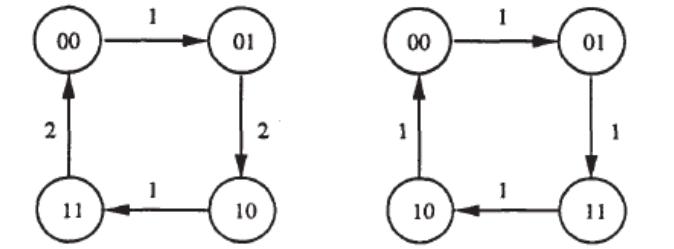
- Binary (Sequential) – States Encoded as Consecutive Binary Numbers
  - Small number of used flip-flops
  - Potentially complex transition functions leading to slow implementations
- One-Hot – Only One Bit Is Active
  - Number of used flip-flops as big as number of states
  - Simple and fast transition functions
  - Preferable coding technique in FPGAs

Types of State Encodings (2)

State	Binary Code	One-Hot Code
S0	000	10000000
S1	001	01000000
S2	010	00100000
S3	011	00010000
S4	100	00001000
S5	101	00000100
S6	110	00000010
S7	111	00000001

State Encoding

- Minimum-bit change: assigns codes to states so that the total number of bit changes for all state transitions is minimized.



(a) Straight–forward encoding (b) Minimum–bit–change encoding

Encoding of states by constants

```
constant s0 : integer := 0;
constant s12 : integer := 1;
constant s3 : integer := 2;
signal state : integer range 2 downto 0;
```

```
...
case state is
        when s0 =>
            if A='1' then output <= '1';
            else output <= '0';
                end if;
-- when s12 =>  -- case must contain all cases!!!
  when others =>output<='0';
end case;
```

Enconding of states

Encoding by subtype

```
SUBTYPE  state_type is STD_LOGIC_VECTOR(1 DOWNTO 0);
CONSTANT s0        : state_type := "01" ;
CONSTANT s1        : state_type := "11" ;
```

Encoding by enum

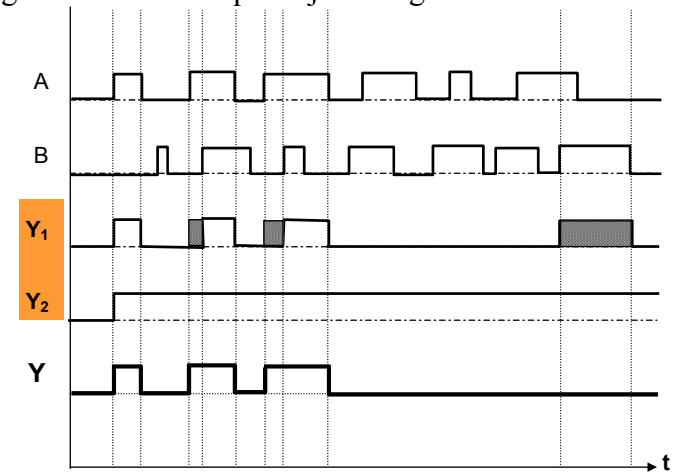
```
type state_type is (s0, s1);
signal state   : state_type;
```

+defined enum values

Examples

Ukázka návrhu automatu

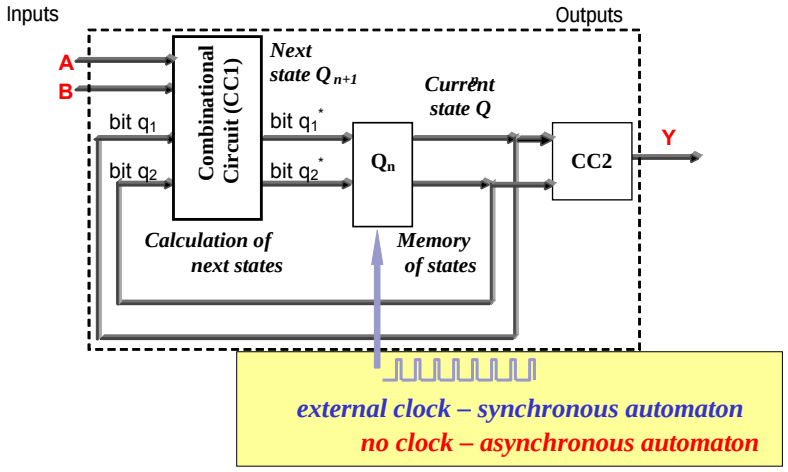
Signál A nešel do 1 později než signál B



Not correct

Návrh začínáme vždy podrobnou analýzou zadání. Jaké možné varianty připouští slovní formulace? Co zadavatel vlastně požaduje? Které možné průběhy mohou nastat?

Asynchronous/Synchronous Automaton/FSM – finite state machine



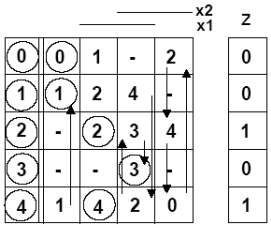
external clock – synchronous automaton
no clock – asynchronous automaton

Univerziální postup syntézy

V praktické syntéze vycházíme z koncepce MOORE automatu a v průběhu syntézy je možná úprava na MEALY automat.

1. Ze zadání časový diagram základní sekvence
2. Tabulka přechodů, graf přechodů
3. Minimalizace množiny vnitřních stavů
4. Volba vnitřního kódu
5. Mapy vnitřní fce (δ zobr.) a výstupní fce (ω zobr.)

4 základní přechody



Řád fundamentálního automatu - počet nestab. stavů v průběhu nejdelšího přechodu

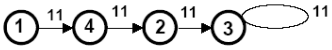
1. Stabilní stav

d(xi,sj) = sj

2. Jednoduchý přechod

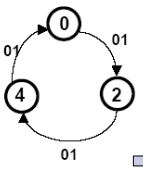
3. Cyklus

- přes nestabilní mezistavy

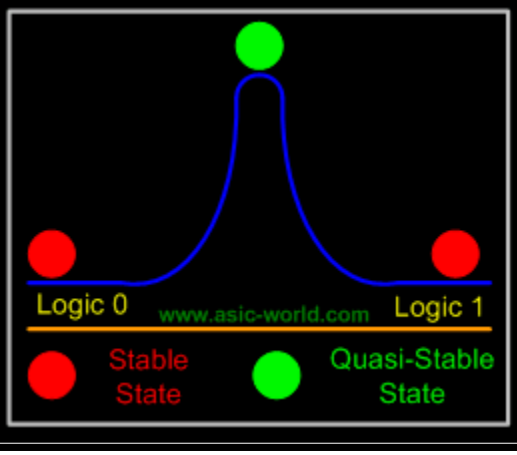


4. Oscilace

(není fundament. režim)



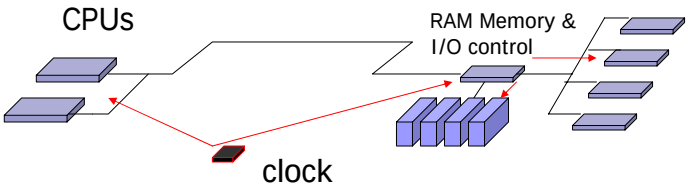
Metastability in FSM



[image source: http://www.asic-world.com/]

SPS 47

Clocks

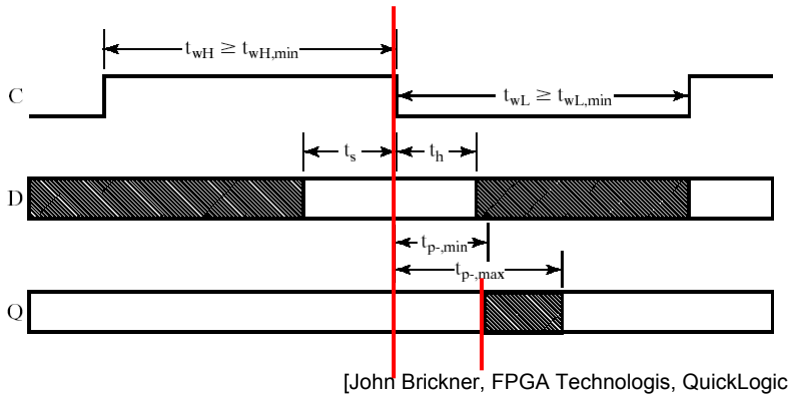


In a system design, clocks may be generated by external chips

- The idea assumption is that the clock edge are synchronized at each device
- A digital clock signal is ideally a 50% duty cycle square wave
- A "***clock domain"*** is comprised of the a set of signals that are referenced to the same idea clock signal.

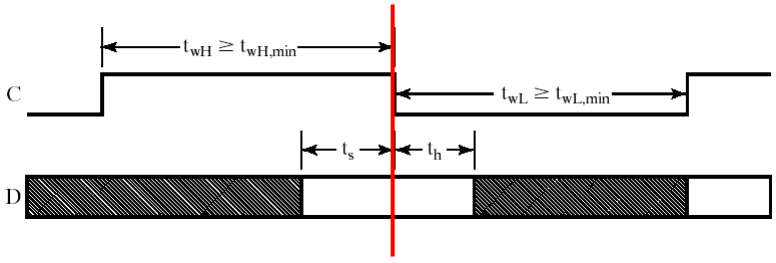
Propagation Delay

- Propagation delay – time after edge when output is available



[John Brickner, FPGA Technologis, QuickLogic 2005]

Flip-Flop Timing



- ***Setup time (t_s)– D must be stable before clock edge***
- ***Hold time (t_h)– D must be stable after clock edge***

[John Brickner, FPGA Technologis, QuickLogic 2005]

Setup nad Hold Times Dependency

Setup and hold times mainly **depend on**

- *technology of flip-flops*
- *routing of signals, i.e. delays on wires clock distributions to flip-flops*
- *combinational logic*

Setup and hold times **do not depend on**

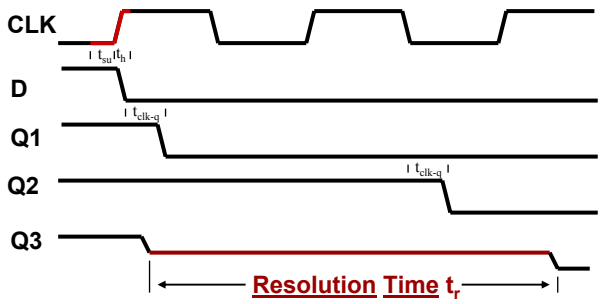
- *frequency*

Metastability: When bad things happen to good synchronizers

Q: What happens when t_{su} / t_h constraints violated?

A: It depends, but there are three scenarios

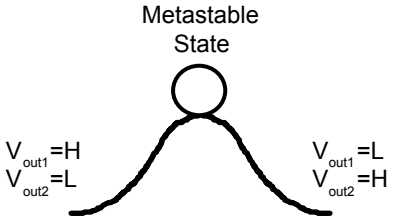
1. Circuit correctly records new D value
1. Circuit retains old D value for an extra cycle
3. Metastability - “stuck” between legal 0 and 1 until it “resolves”



[John A. Nestor: VLSI Design]

Metastability - “Ball on the Hill” Analogy

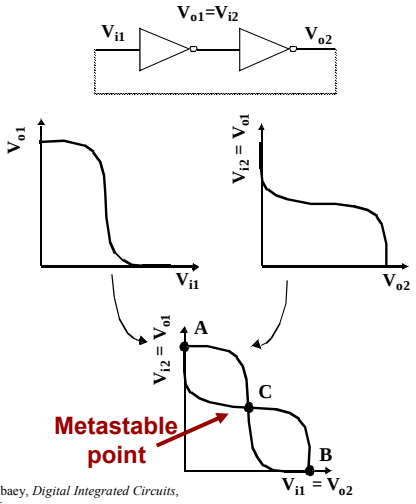
- Sides of hill = stable states
- Top of hill = metastable state
- Any small “push” (e.g., noise) will move the ball off the hill and into a stable state



[John A. Nestor: VLSI Design]

Metastability

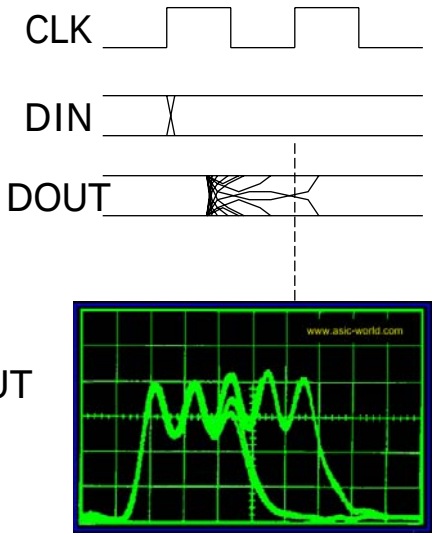
- Two stable states
  - V_{o1}=L, V_{o2}=H
  - V_{o1}=H, V_{o2}=L
- One metastable state
  - V_{o1} = V_{o2}
- Ugly characteristic: unbounded recovery time t_r

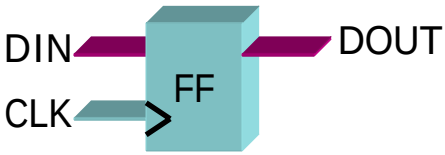


Graphic source: J. Rabaey, *Digital Integrated Circuits*, © Prentice-Hall, 1996

Metastability

- Limits I/O clock rates
- New designers must be informed





[John Brickner, FPGA Technologis, QuickLogic 2005]

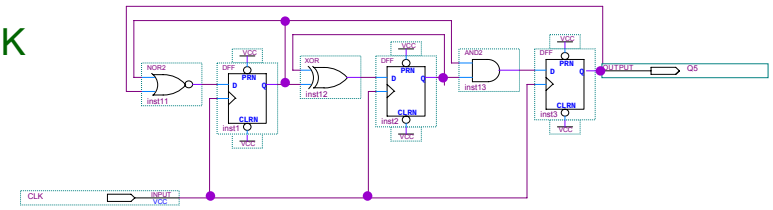
Metastability - Bad News / Good News

- Bad news
  - Metastability is unavoidable
  - Recovery time is theoretically unbounded
- Good news
  - Can empirically measure recovery times
  - Can use statistics from recovery times to make failure probability arbitrarily small

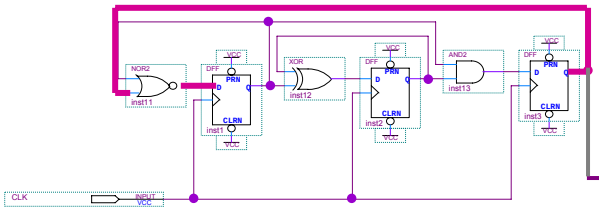
[John A. Nestor: VLSI Design]

Example: Metastability Caused by Adding Circuit

OK

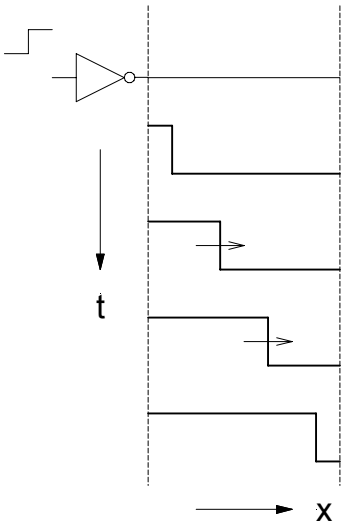


Metastable due to hold time violation

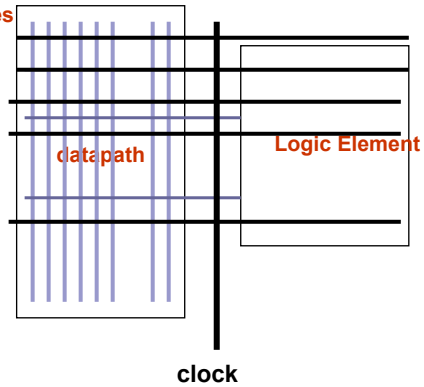


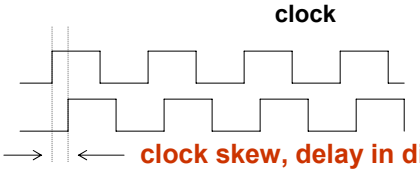
External Circuit

Delay - wires + logic



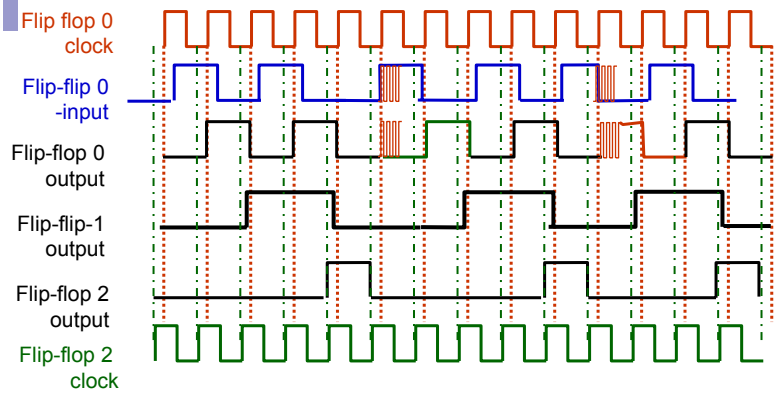
Data busses





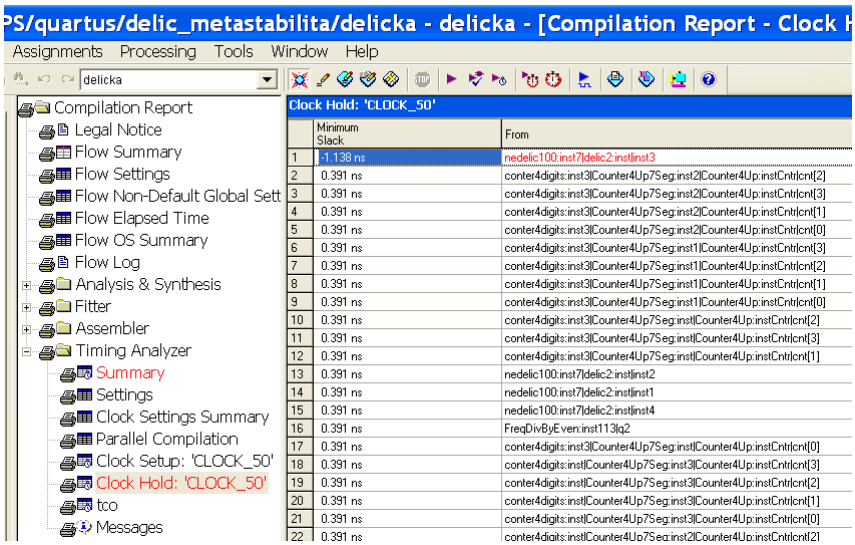
clock skew, delay in distribution

Metastability Caused by Clock Skew



- Flip-flop2 clock signal is accidently ahead of flip-flop 0 clock signal in FPGA circuit.
- Hold time violation occurs at flip-flop 0 data input for the edge that depends on flip-flop 2 output.
- Flip-flop 0 metastability can result in wrong final state of flip-flop 0 output.
- Divider can accidently dived by 4 instead of 5.

Quartus Compiler Report



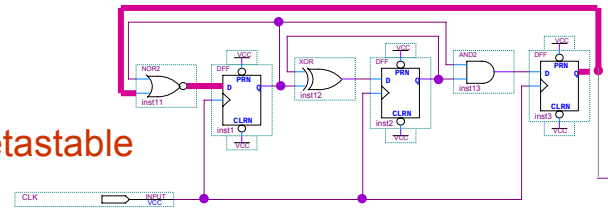
Removing Metastability

Removing metastability means removing setup and hold time violations in circuit. Many methods can be tested, e.g.

1. Rearranging circuits
2. Increasing delay by adding LCELL
3. Synchronizers
4. Negative/Positive edges phases
5. Redesigning circuit

- and others.....

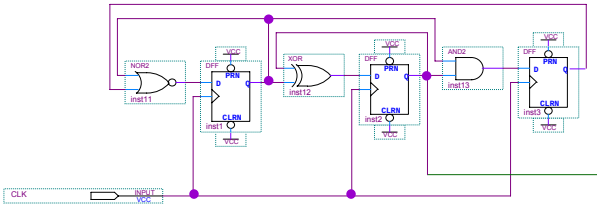
1. Rearranging Circuit



Metastable

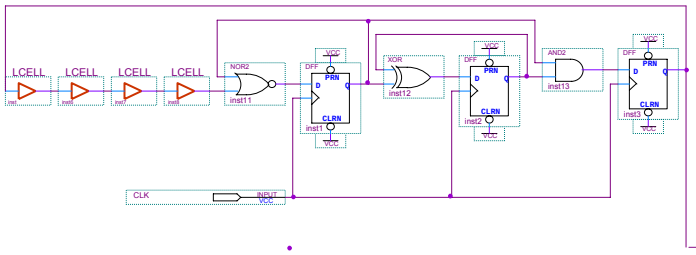
External Circuit

Hold time violation was removed, no metastability



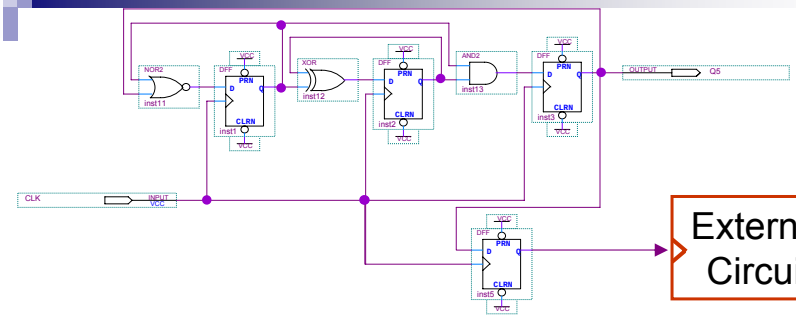
External Circuit

2. Increasing Delay by LCELLs



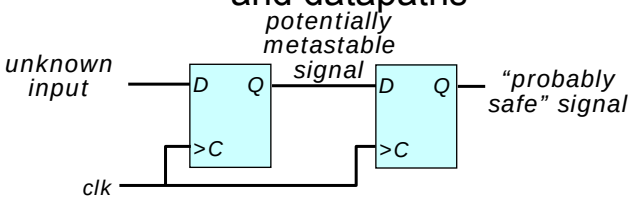
External Circuit

3. Synchronizer

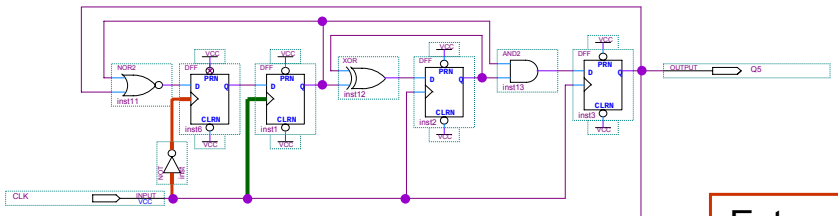


External Circuit

Synchronizers isolate metastable signals and datapaths



4. Negative/Positive Edge Phases



External Circuit

5. Redesigning circuit

```
library ieee; use ieee.std_logic_1164.all; use ieee.std_logic_unsigned.all;
entity div5_10_cntr is
   port(CLK  : in std_logic;
        q5   : out std_logic);
end entity;
architecture behav of div5_cntr is
begin
  process (CLK)
  variable   cnt : integer range 0 to 4:=0;
  begin
     if (CLK'event and CLK='1') then
        if cnt>3 then cnt:=0; q5<='1';
        else cnt := cnt + 1; q5<='0';
        end if;
     end if;
  end process;
end behav;
```

The Ten Commandments for Successful Design

1. All state machine outputs shall always be registered
2. Thou shalt use registers, never latches
3. Thy state machine inputs, including resets, shall be synchronous
4. Beware fast paths lest they bite thine ankles
5. Minimize skew of thine clocks
6. Cross clock domains with the greatest of caution. Synchronize thy signals!
7. Have no dead states in thy state machines
8. Have no logic with unbroken asynchronous feedback lest the fleas of myriad Test Engineers infest thee
9. All decode logic must be crafted carefully – eschew asynchronicity
10. Trust not thy simulator – it may beguile thee when thy design is junk

Peter Chambers, Peter's Provocative Pontifications, 97

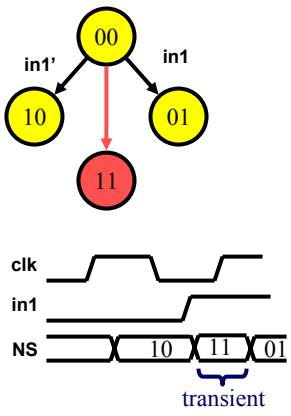
Old English

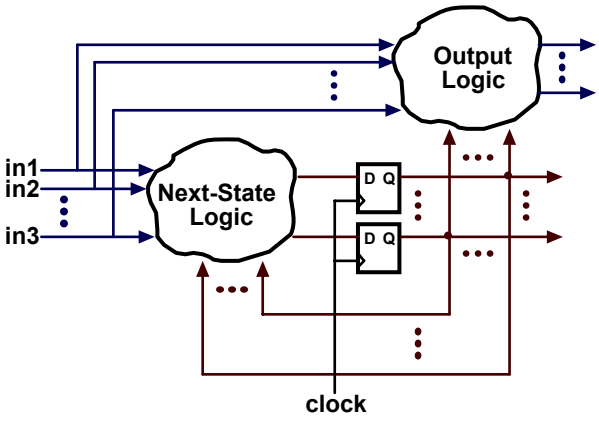
	Subjective	Objective	Possessive
1st Pers.	I	me	my/mine
2nd Pers.	thou	thee	thy/thine
3rd Pers.	he/she /it	him/her /it	his/her/ its
1st Pers. Pl	we	us	our
2nd Pers. Pl	ye/you[2]	you	your
3rd Pers. Pl.	they	them	their

to be - Present tense	to have - Present tense
I am	I have
thou art	thou hast
he/she/it is	he/she/it hath
we are	we have
ye are	ye have
they are	they have

Synchronizer Review

- Key idea: make sure inputs don't change at a "bad time" in sequential circuits

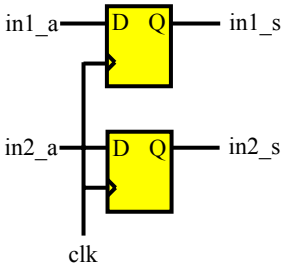




[John A. Nestor: VLSI Design]

Adding Synchronizers

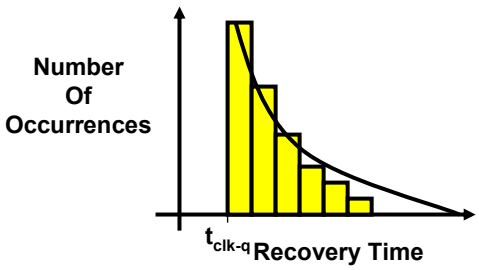
- Add a D Flip-Flop on each asynchronous input
- Synchronize each input only once
- Don't use dynamic flip-flops (we'll discuss why)
- Q: What happens when set up & hold time violated?



[John A. Nestor: VLSI Design]

Measuring Metastability Characteristics

- Intentionally cause metastability many times
- Measure recovery for each occurrence
- Fit recovery times to exponential function



[John A. Nestor: VLSI Design]

Designing with Metastability

- A synchronizer design at a given clock period provides a fixed amount of resolution time tr
- Definition: a synchronization failure occurs when actual recovery time $t_{r\text{-actual}} > t_r$
- For a given flip-flop, the mean time between failure (MTBF) is given by the formula

$$MTBF(t_r) = \frac{e^{(t_r/\tau)}}{T_o \times f_{clk} \times a}$$

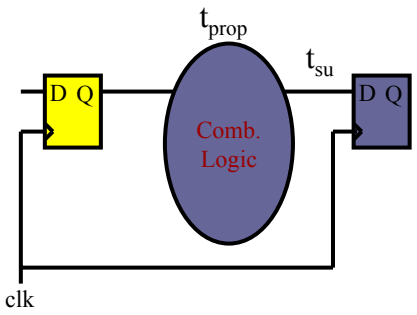
f_{clk} - System clock freq.
a - asynchronous input rate of change.
τ - empirically derived constant
T_o - empirically derived constant
t_r - time available for resolution

[John A. Nestor: VLSI Design]

Determining Resolution Time t_r

- Must leave time for system to respond properly after resolution

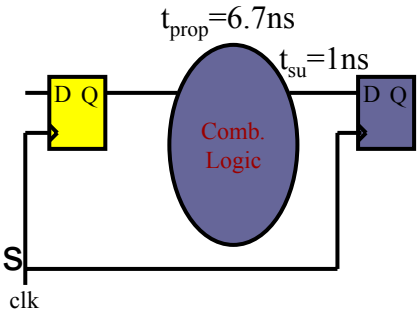
$$t_r = t_{clk} - t_{su} - t_{prop}$$



[John A. Nestor: VLSI Design]

Resolution Time Example

- Suppose that
  - f_{clk} = 100MHz (t_{clk} = 10ns)
  - a = 1MHz
  - t_{prop} = 6.7ns
  - t_{su} = 1ns
- Calculate t_r:
  - $t_r = t_{clk} - t_{su} - t_{prop}$
  - t_r = 10ns - 6.7ns -1ns
    = 2.3ns



[John A. Nestor: VLSI Design]

MTBF Calculation Example

- "Typical" values for a 0.25μm ASIC library flip-flop
  - τ = 0.31ns
  - T_o = 9.6as "a" = 10^{-18}
  - t_r = 2.3ns
- MTBF = 20.1 days - unacceptable!

$$MTBF(t_r) = \frac{e^{(t_r/\tau)}}{T_o \times f_{clk} \times a}$$

[John A. Nestor: VLSI Design]

What happens if we halve f_{clk}?

- Suppose that
  - f_{clk} = 50MHz (t_{clk} = 20ns)
  - a = 1MHz
  - t_{prop} = 6.7ns
  - t_{su} = 1ns
- Calculate t_r and MTBF:

$$MTBF(t_r) = \frac{e^{(t_r/\tau)}}{T_o \times f_{clk} \times a}$$

  - $t_r = t_{clk} - t_{su} - t_{prop}$
  - t_r = 20ns - 6.7ns -1ns = 12.3ns
  - MTBF = 5.7 X 10^{28} seconds = 1.8 X 10^{21} years

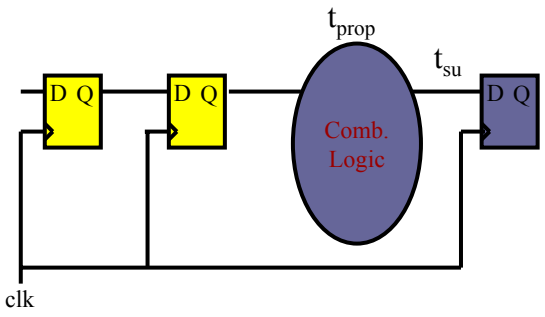
[John A. Nestor: VLSI Design]

Alternative: Dual-Stage Synchronizer

- Increased value for t_r:

$t_r = t_{clk} - t_{su} - t_{pr}$

t_r = 10ns - 1ns = 9ns



[John A. Nestor: VLSI Design]

Dual-Stage MTBF Calculation

- "Typical" values for a 0.25µm ASIC library flip-flop
  - τ = 0.31ns
  - T_o = 9.6as "a" = 10^{-18}
  - t_r = 9ns
- MTBF = ?

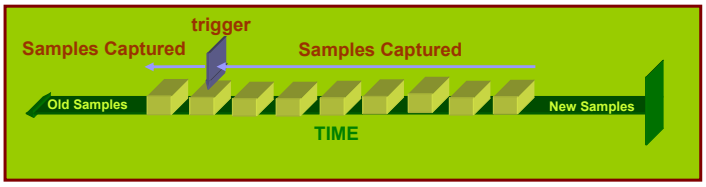
$$MTBF(t_r) = \frac{e^{(t_r/\tau)}}{T_o \times f_{clk} \times a}$$

[John A. Nestor: VLSI Design]

How can we look inside FPGA?

by SignalTAP Logic Analyzer

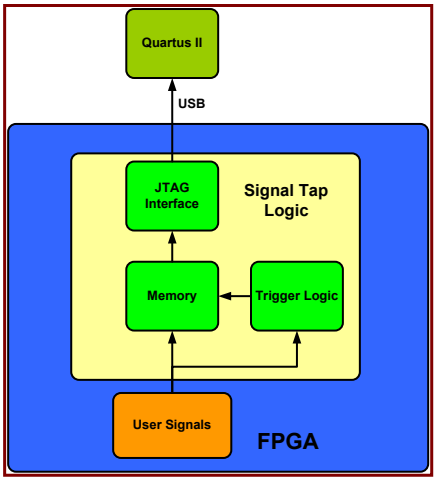
Logic Analyzer - How it works



- Trigger Options
  - Pre Trigger
  - Center Trigger
  - Post Trigger

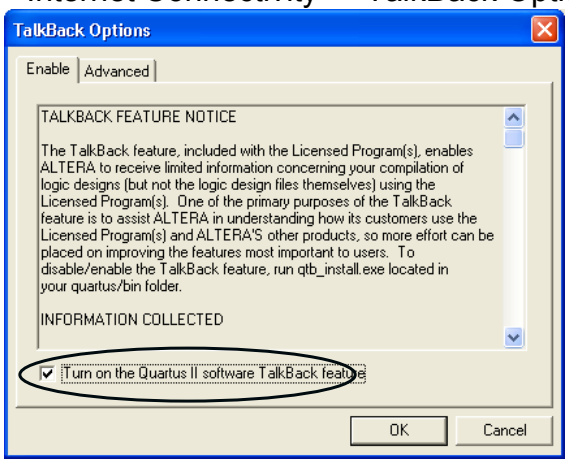
Signal TAP - How it Works

- Signal Tap “Wastes” FPGA Logic



Activating STP in Web Eddition

- Options -> Internet Connectivity -> TalkBack Options

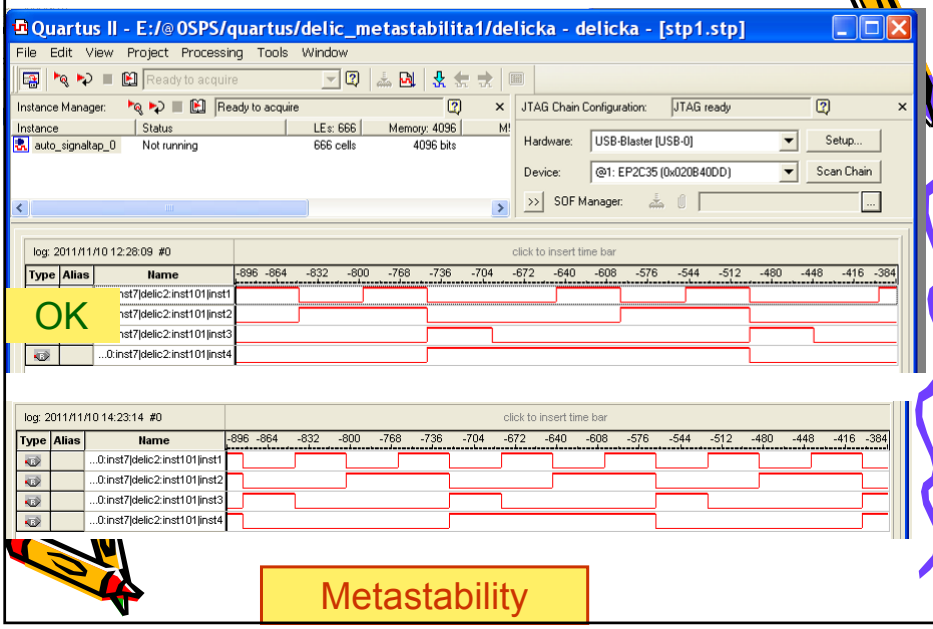


See: http://www.altera.com/support/kdb/solutions/rd06202008_168.html

Signal TAP Usage

- Create .STP File
  - Assign Sample Clock
  - Specify Sample Depth
  - Assign Signals to STP File
  - Specify Triggering
  - Setup JTAG
- Save .STP File & Compile with Design
- Program Device
- Acquire Data

Results of SignalTap



Metastability

Dodatek

- jak si nastavit NIOS...