Design of Digital Circuits(S2)

Chapter 1, Part 2

Modelling and Simulation

Section 1.3 Delay Time Tolerance

to 1.6 Sequential Circuits

Prof. G. Kemnitz

Institute of Informatics, Clausthal University of Technology

May 14, 2012

Delay Time Tolerance

Register

2.

Asynchronous input

3.1 Sampling a single bit
3.2 Switch de-bouncing
3.3 Asynch. initialization
3.4 Parallel interface
3.5 Exercises

Sequential circuit

4.1 Finite state machine
4.2 Automaton ⇒ VHDL
4.3 System crash
4.4 Combination lock
4.5 Operation graph
4.6 Quadrature encoder

4.7 Exercises

Delay Time Tolerance

Shortcomings of the hitherto Model

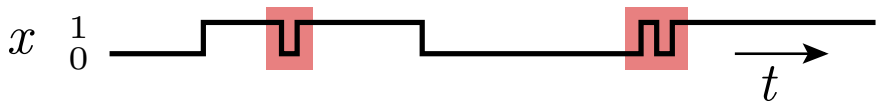
- signals change continuing and do not jump
- between two values they are for a short time invalid
- one input change may cause multiple output changes
- delay time is not known exactly / depends on temperature, variation in the production process, ageing, ...

A digital circuit is only reliable, if it's function does not depend on timing parameters as long as they are within there range of tolerance.

Glitches

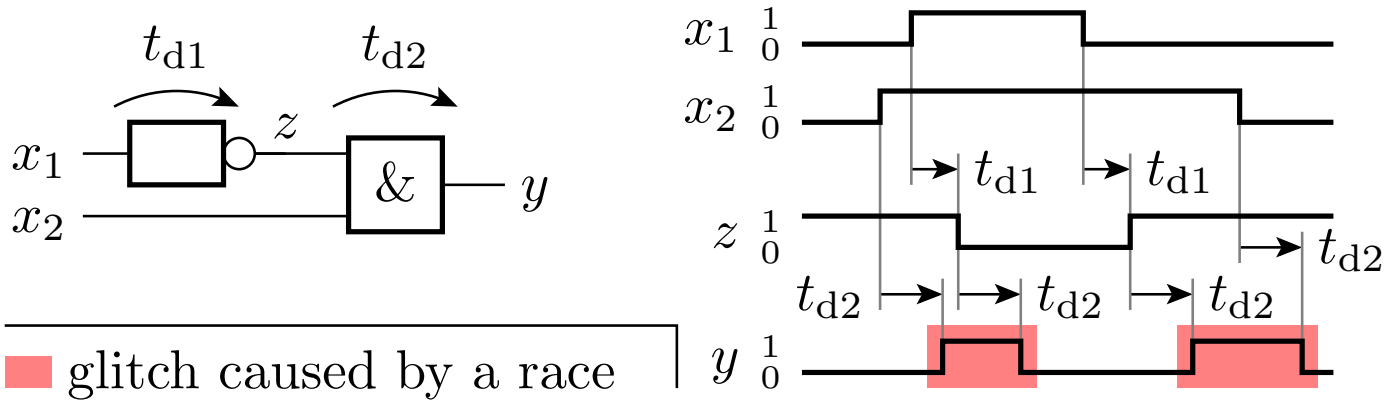
Glitches, Races

Glitches: short pulses, arising during signal processing and may cause malfunctions



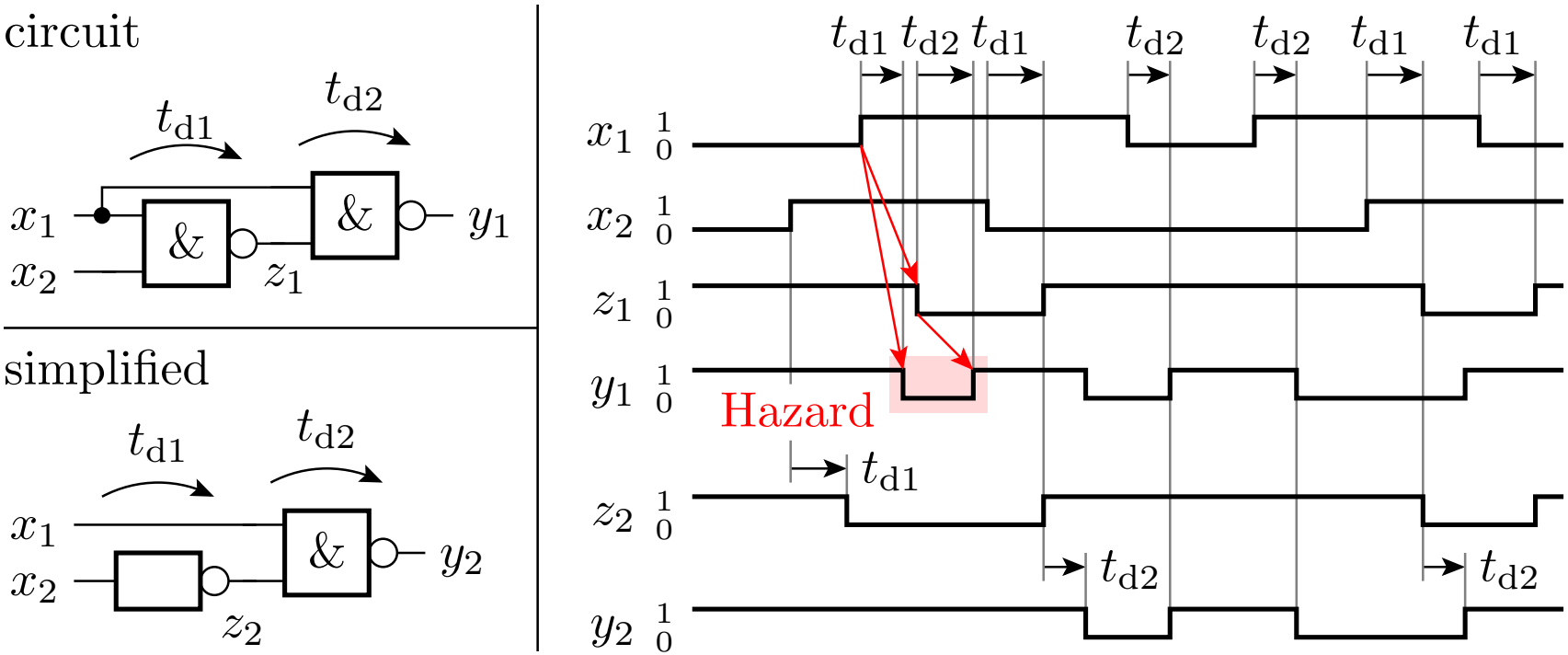
Possible Causes:

- Race: almost simultaneous changes at multiple inputs and different delay times.



Hazard

- also if only the value at one input changes, it may cause multiple output changes
- Cause: convergent signal flow (branching flow reuniting later again)



races and Hazards depends

- less on the function, but more on the structure (slide before, left)
- the number and the length of the glitches depends on the delay times and so on temperature, variation in the production process, ageing, ...
- in specifications for synthesis glitches are neither foreseeable nor excluded

---

- the up to now run time model is very imprecise due to effects like glitches

Fact 1

Digital circuits should be designed so that possible glitches do not effect the function.

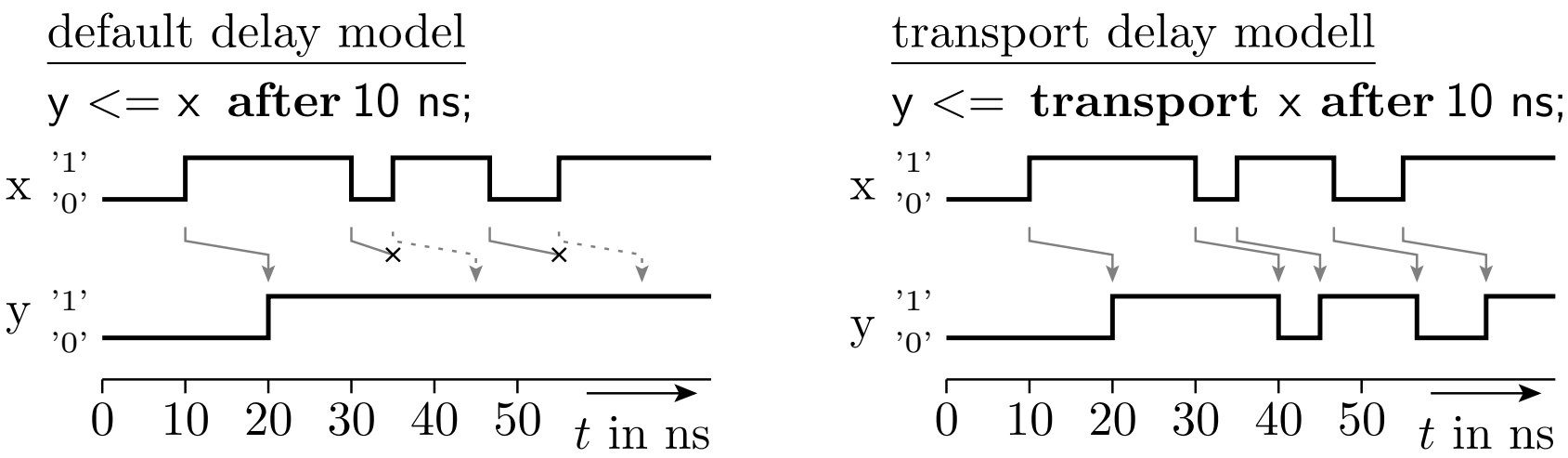
Delay model

Delay model of a signal assignment

- Glitches multiply the signal events
- Signal assignments eliminate short glitches; adjustable by the delay model

```
signal_name <= [DM] expression after td;
        DM ⇒ transport|[reject tr] inertial]
```

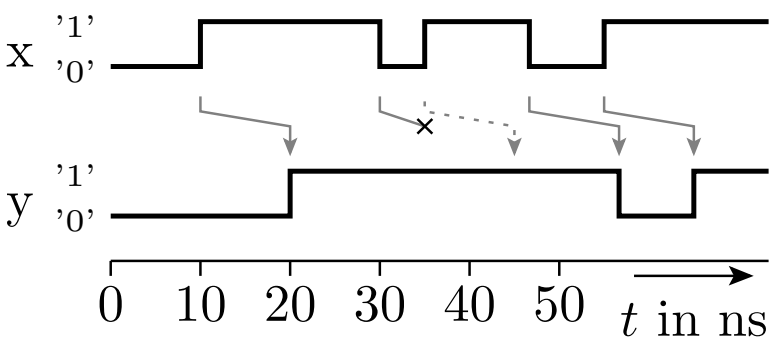
  - *DM* -- delay model (optional)
  - t_d -- delay time
  - t_r -- reject time



- default delay model: new assignment deletes all pending transactions
- transport delay model: new assignment deletes only pending transactions later than the new assigned transaction

delay model with reject time

y <= **reject** 5 ns **inertial** x **after** 10 ns;



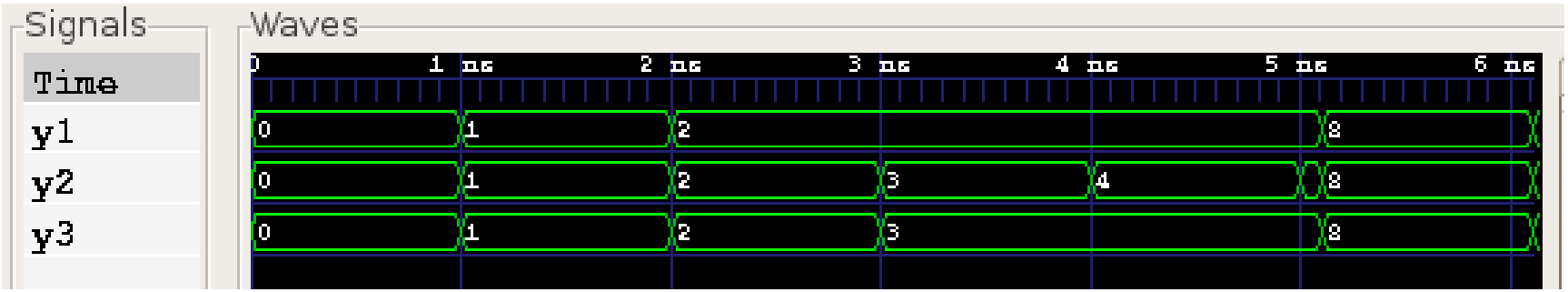
pending transactions, witch

- executed normaly
- will be deleted by a subsequent assignment
- are not scheduled because the value will stay the same

- ignores only pulses up to the width of the reject time t_r
- deletes all pending transactions for points of time later $t_\mathrm{d} - t_\mathrm{r}$ except pending transactions assigning the same value just before a new scheduled transaction

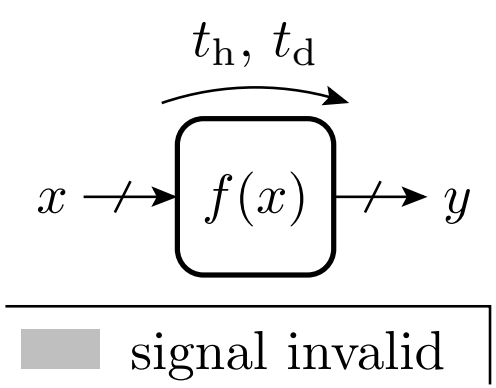
A simulation example

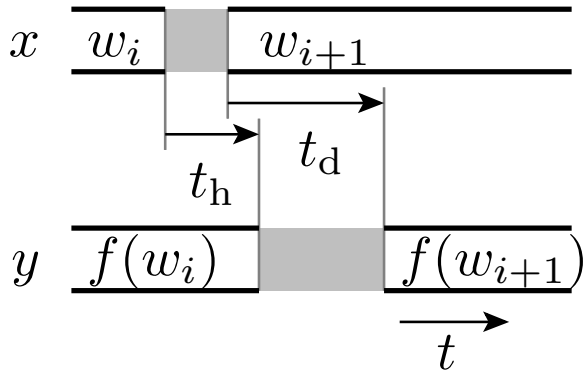
```
signal y1, y2, y3: NATURAL;
...
y1 <= 1 after 1 ns, 2 after 2 ns, 3 after 3 ns, 4 after 4 ns,
5 after 5 ns;
y2<=y1;  y3<=y2;   wait for 2.1 ns;
y1 <= 8 after 3 ns, 9 after 4 ns;
y2 <= transport  8 after 3 ns, 9 after 4 ns;
y1 <= reject 1.5 ns inertial 8 after 3 ns, 9 after 4 ns;
wait;
```



Delay tolerance region

Delay tolerance region





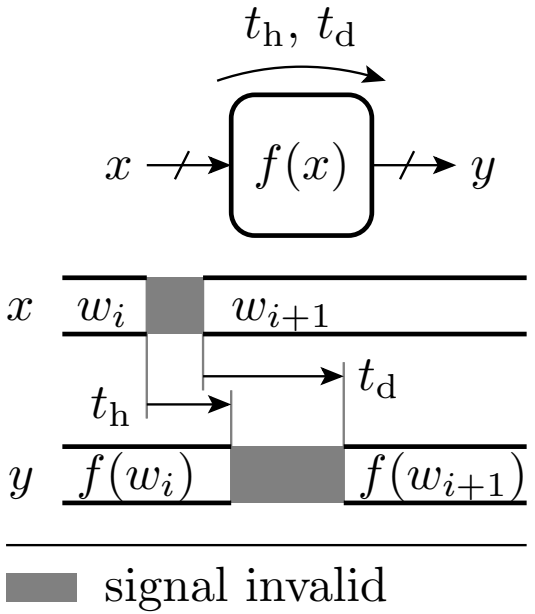
- t_h hold time, time the old output value stay valid after an input transaction
- t_d delay time, maximum time between an input transition to a valid value and the corresponding output transition

invalid:

- value in the forbidden range; point of transition time unknown; potential glitch

Description in VHDL

```
process(x)
begin
 if input_before_valid then
  y <= invalid after th;
 end if;
 if new_input_value_valid then
  y <= transport f(x) after td;(1)
 end if;
end process;
```



(1) transport model, so that the pending transaction to invalid will not be deleted

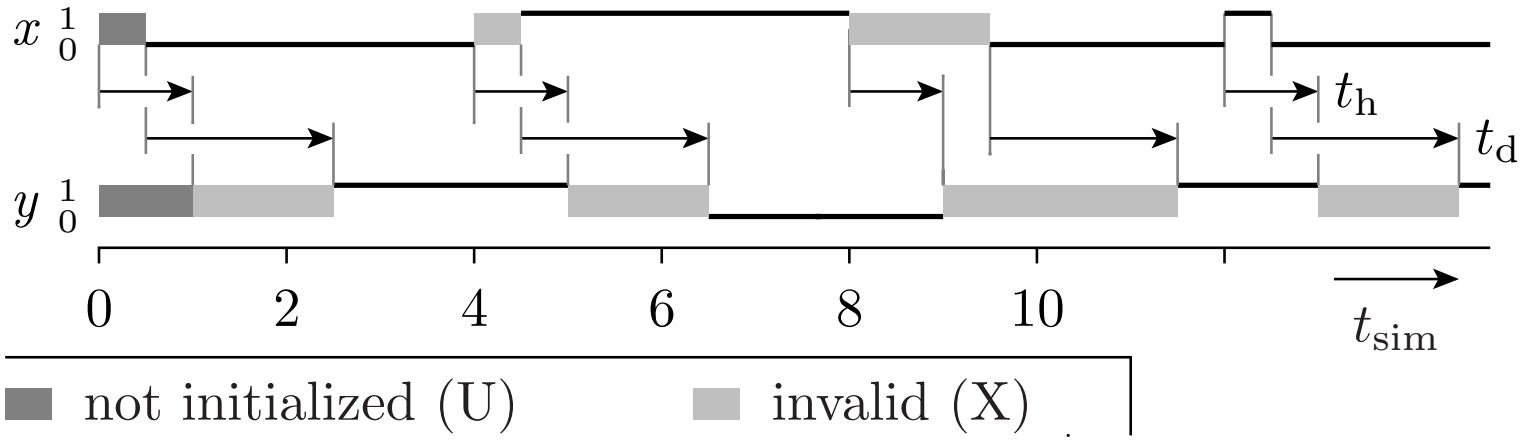
---

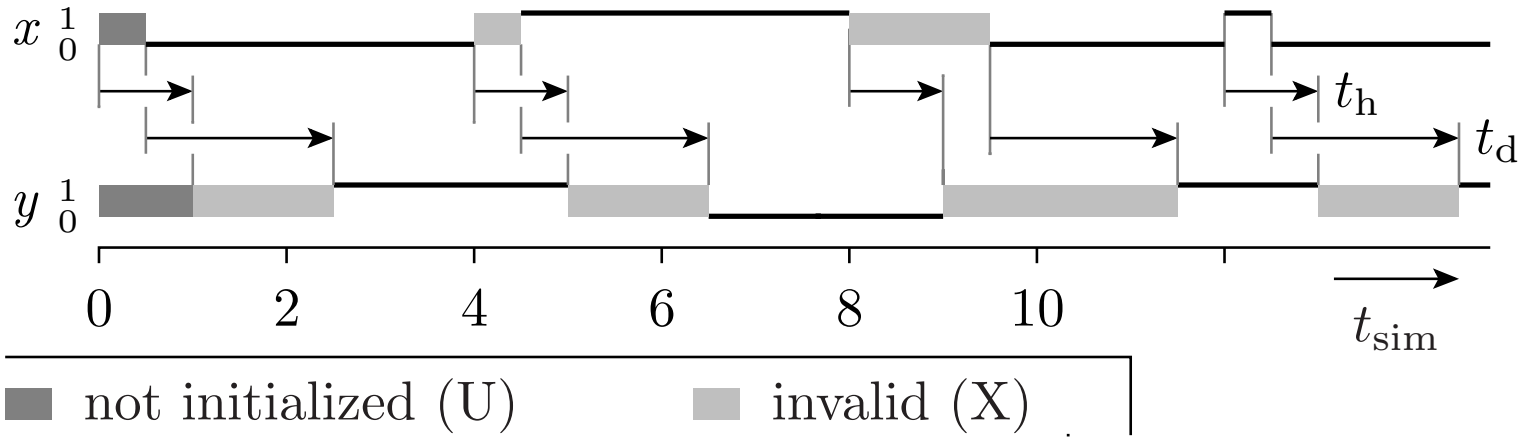
- abbreviation

```
y <= invalid after th, f(x) after td;
```

Simulation of an inverter

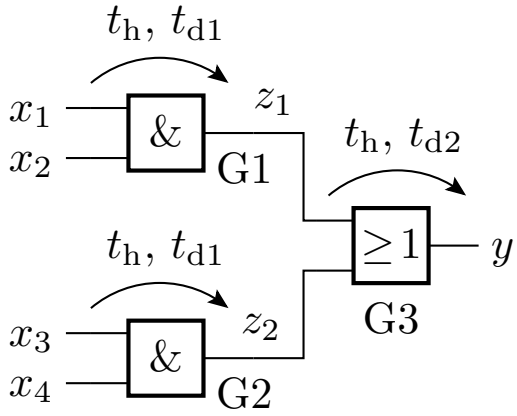
```
signal x, y: STD_LOGIC;
...
x <= '0' after 0.5 ns, 'X' after 4 ns, '1' after 4.5 ns,
     'X' after 8 ns, '0' after 9.5 ns, '1' after 12 ns,
     '0' after 12.5 ns;
y <= 'X' after 1 ns, not x after 2 ns;
```





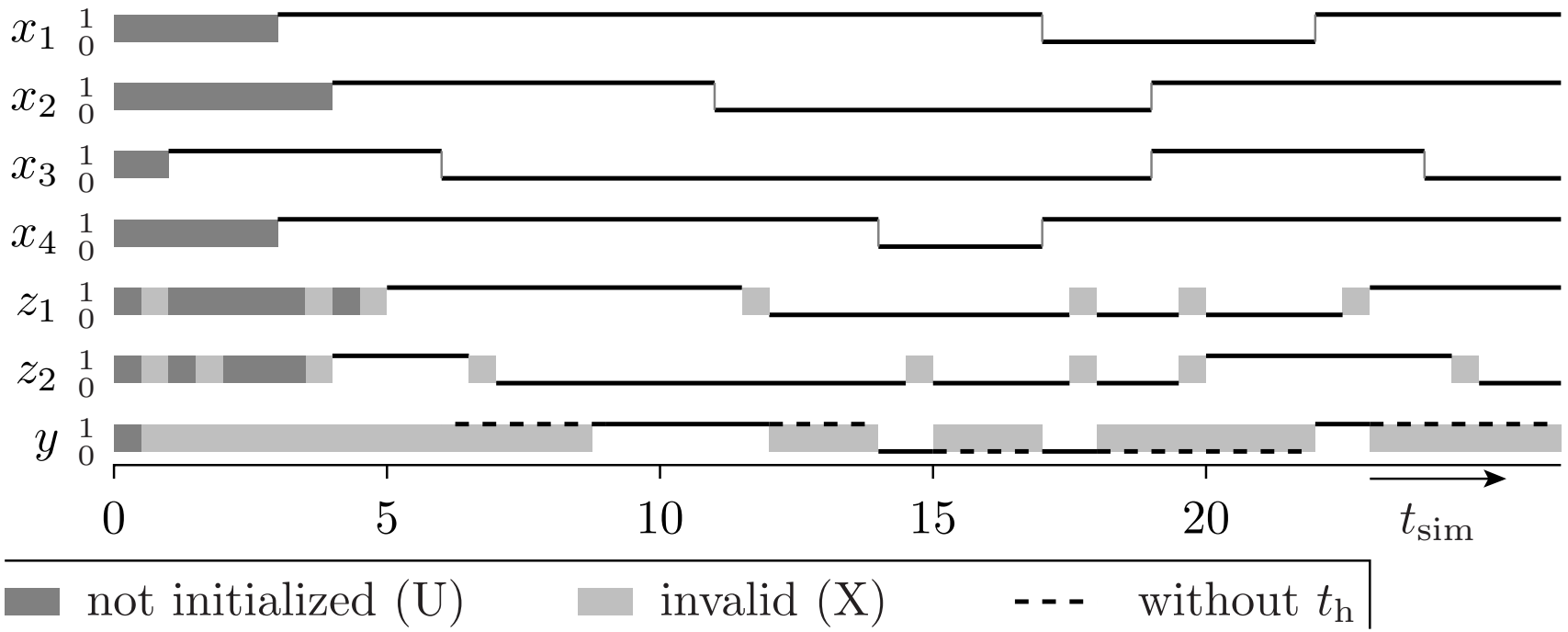
- process »x-assignment« will only be executed at the begin of the simulation; schedules multiple transactions
- Process »y-assignment« also restarts with each transaction of x ; assigns two transactions
- default delay model, each new assignment deletes all pending transaction
- in case of pending transaction to »X« the new assignment of »X« takes the transaction time of the pending transaction to »X«

Circuit with multiple gates




```
signal x1, x2, x3, x4,
    z1, z2, y: STD_LOGIC;
constant th: DELAY_LENGTH:= 500 ps;
constant td1: DELAY_LENGTH:= 1 ns;
constant td2: DELAY_LENGTH:= 2 ns;
...
G1:z1<= 'X' after th, x1 and x2 after td1;
G2:z2<= 'X' after th, x3 and x4 after td1;
G3: y<= 'X' after th, z1 or  z2 after td2;
input_process: process begin
 wait for 1 ns; x3<='1';
 wait for 2 ns; x1<='1'; x4<='1';
 wait for 4 ns; x2<='1'; wait for 3 ns; x4<='0';
 wait for 2 ns; x3<='0'; wait;
end process;
```

```
G1:z1<= 'X' after th, x1 and x2 after td1;
G2:z2<= 'X' after th, x3 and x4 after td1;
G3: y<= 'X' after th, z1 or  z2 after td2;
```



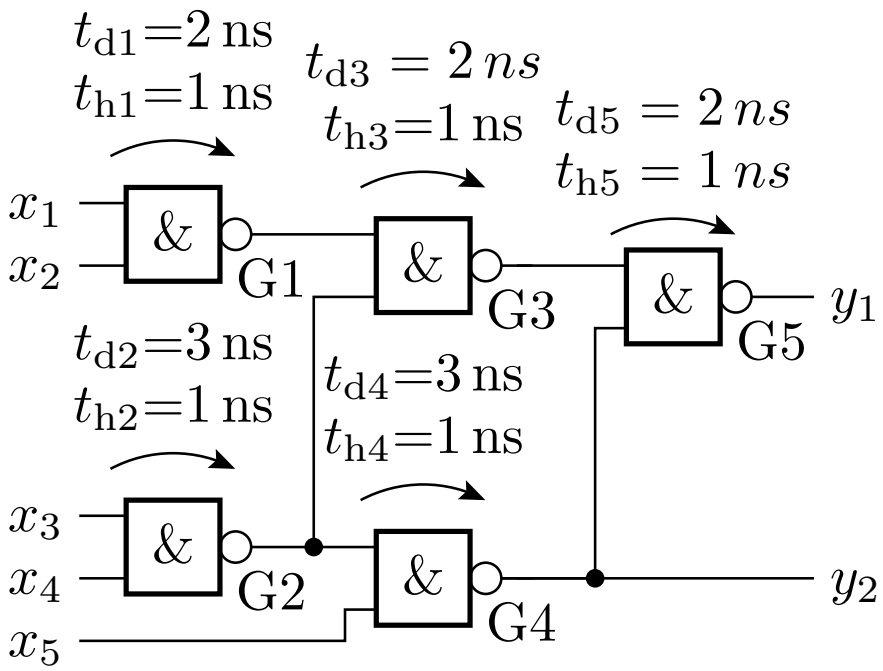
- the output signal is in the example most time invalid
- invalid means »not necessary correct«
- circuits processing invalid signal values may operate right, but reliable; difficult to debug

Run Time Analysis

Run Time Analysis

Calculation of the hold and delay time of the complete circuit from the timing parameters of the subcircuits

path	$\sum t_{\mathrm{h}.i}$	$\sum t_{\mathrm{d}.i}$
G1-G3-G5	3 ns	6 ns
G2-G3-G5	3 ns	7 ns
G2-G4-G5	3 ns	8 ns
G2-G4	2 ns	6 ns
G4-G5	2 ns	5 ns
G4	1 ns	3 ns

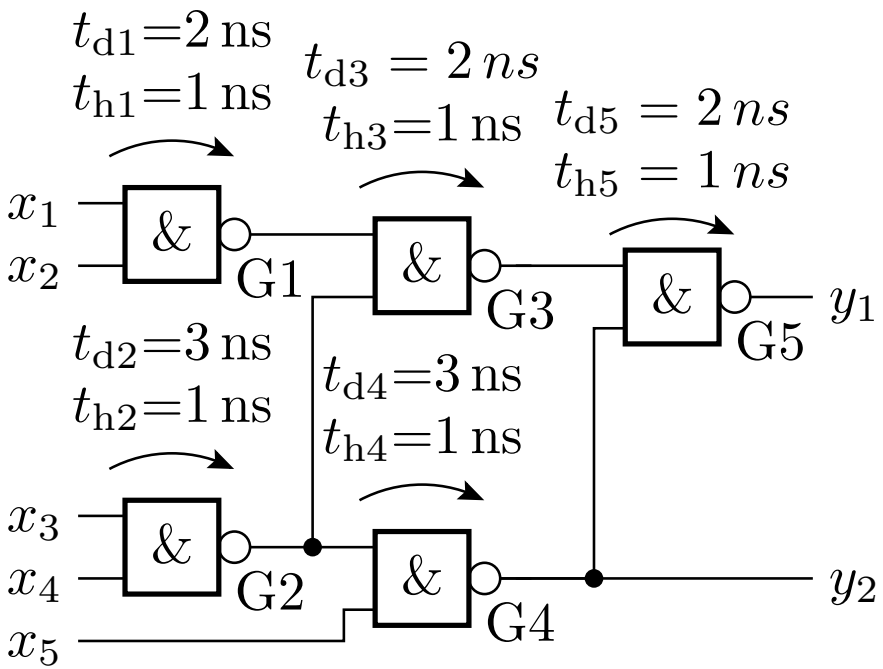


- summing up the hold and delay times along the path's
- the hold time of the complete circuit is that of the shortest path
- the delay time of the complete circuit is that of the path with the longest delay

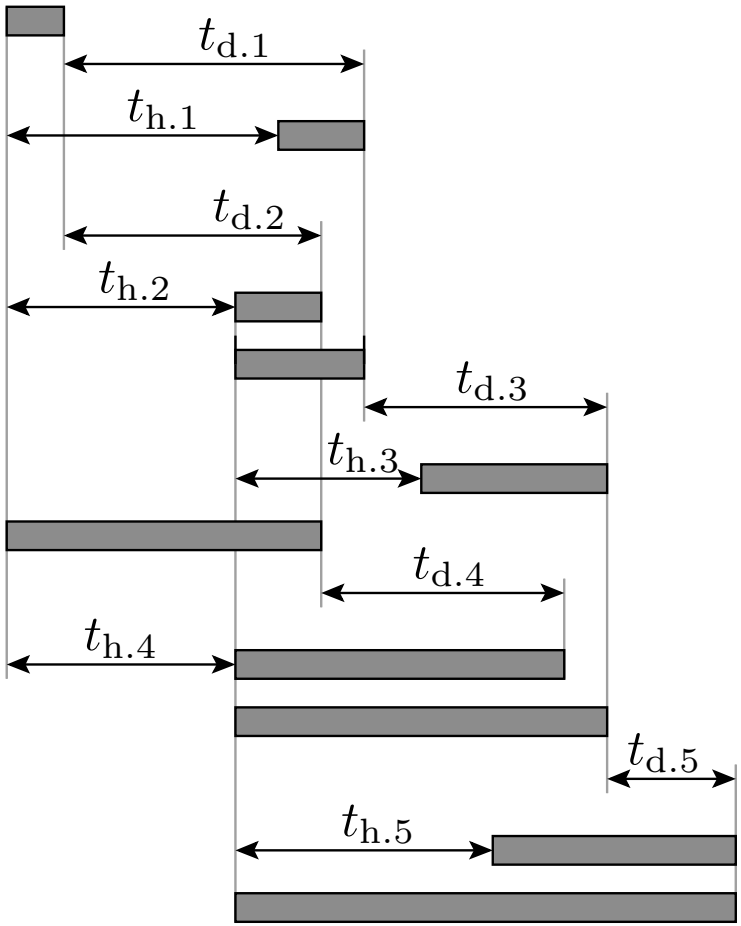
Algorithm for Large Circuits

the number of path's grows exponentially; better algorithm with linear order:

- repeat for each signal group starting from the inputs
  - calculate hold and delay time to it



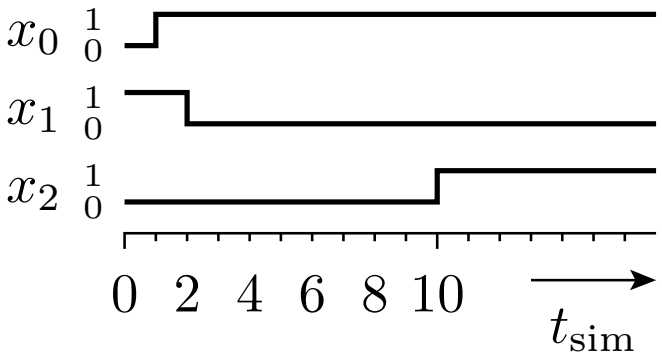
signal group		$t_{\mathrm{hS}.i}$	$t_{\mathrm{dS}.i}$
0		0 (Definition)	0 (Definition)
1		$t_{\mathrm{hS}.0} + t_{\mathrm{h}.1}$	$t_{\mathrm{dS}.0} + t_{\mathrm{d}.1}$
2		$t_{\mathrm{hS}.0} + t_{\mathrm{h}.2}$	$t_{\mathrm{dS}.0} + t_{\mathrm{d}.2}$
3		$\min(t_{\mathrm{hS}.1}, t_{\mathrm{h}.2})$	$\max(t_{\mathrm{dS}.1}, t_{\mathrm{dS}.2})$
5		$t_{\mathrm{hS}.3} + t_{\mathrm{h}.3}$	$t_{\mathrm{dS}.3} + t_{\mathrm{d}.3}$
4		$t_{\mathrm{hS}.0}$	$t_{\mathrm{d}.2}$
6		$t_{\mathrm{hS}.0} + t_{\mathrm{h}.4}$	$t_{\mathrm{dS}.2} + t_{\mathrm{d}.4}$
7		$\min(t_{\mathrm{hS}.5}, t_{\mathrm{h}.6})$	$\max(t_{\mathrm{dS}.5}, T_{\mathrm{d}.6})$
8		$t_{\mathrm{hS}.7} + t_{\mathrm{h}.5}$	$t_{\mathrm{dS}.7} + t_{\mathrm{d}.5}$
9		$\min(t_{\mathrm{hS}.6}, t_{\mathrm{hS}.9})$	$\max(t_{\mathrm{dS}.6}, t_{\mathrm{dS}.9})$

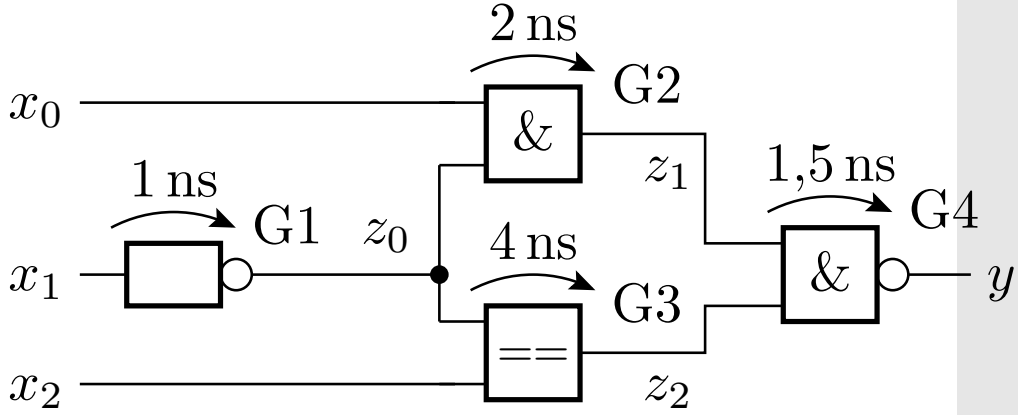


- linear order ⇒ also for large circuits

Exercises

Aufgabe 1.9: Hazard





- VHDL description of the circuit with concurrent signal assignments
- VHDL description of the input assignments
- calculate the signal forms of z_1, z_2 and y
- What signal transactions causes a glitch?

Aufgabe 1.10: VHDL delay models

```
signal y: STD_LOGIC;
...
--- nebenläufige Zuweisungen
y <= '0', 'X' after 3 ns, '1' after 7 ns,  'X' after 8 ns,
     '1' after 10 ns,  '0' after 11 ns, '1' after 13 ns,
     '1' after 15 ns, 'X' after 18 ns, '0' after 20 ns;
wait for 5 ns;
y <= '1' after 12 ns;
--- y <= transport '1' after 12 ns;
--- y <= reject 8 ns inertial '1' after 12 ns;
```

With pending transactions are deleted, if the second assignment uses

1. the default delay model
2. the transport model
3. the inertial model with 8 ns reject time?

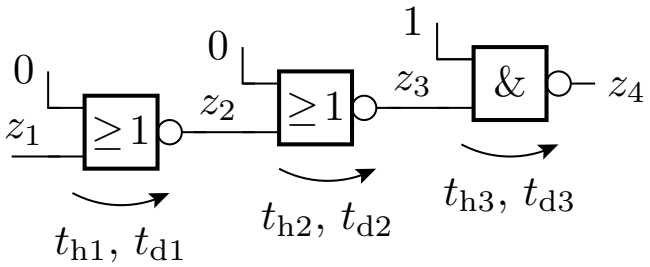
Aufgabe 1.11: Run time analysis

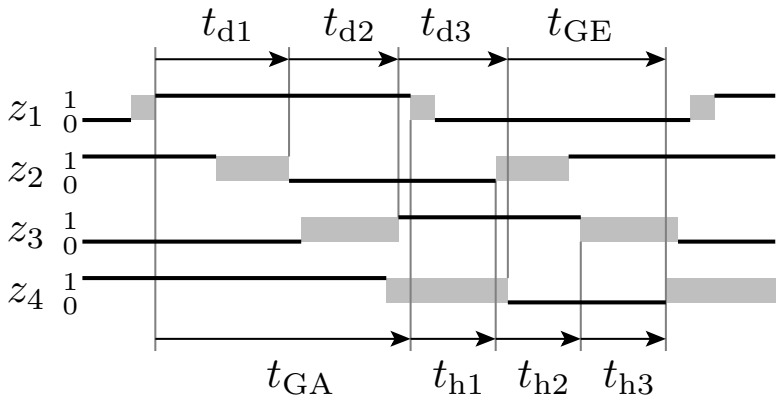
```
--- Vereinbarungsteil der Entwurfseinheit
  signal x: STD_LOGIC_VECTOR(4 downto 0):="01011";
  signal z: STD_LOGIC_VECTOR(4 downto 0);
  signal y: STD_LOGIC;
--- Anweisungsteil der Entwurfseinheit
G1: z(0) <= 'X' after 4 ns, x(0) and x(1) after 8
ns;
G2: z(1) <= 'X' after 5 ns, x(2)  or  x(3) after 6 ns;
G3: z(2) <= 'X' after 3 ns, z(0)  or  z(1) after 6 ns;
G4: z(3) <= 'X' after 4 ns, z(1) and x(3) after 7
ns;
G5: z(4) <= 'X' after 3 ns, z(3)  or  x(4) after 5 ns;
G6: y    <= 'X' after 5 ns, z(2)  or z(4) after 7 ns;
```

1. Draw the signal flow plan with hold and delay times.
2. Calculate the hold and the delay time of the complete circuit.

Register

Period of validity and sampling





t_{GA}	period of validity at the begin of the path
t_{GE}	periode of validity at the end of the path

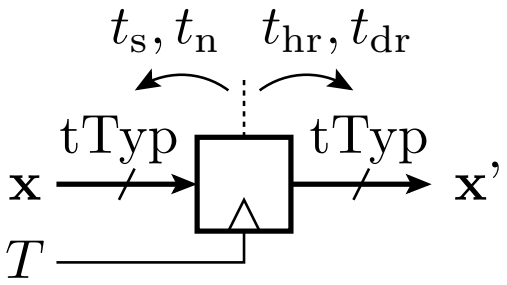
- period of validity a the output:

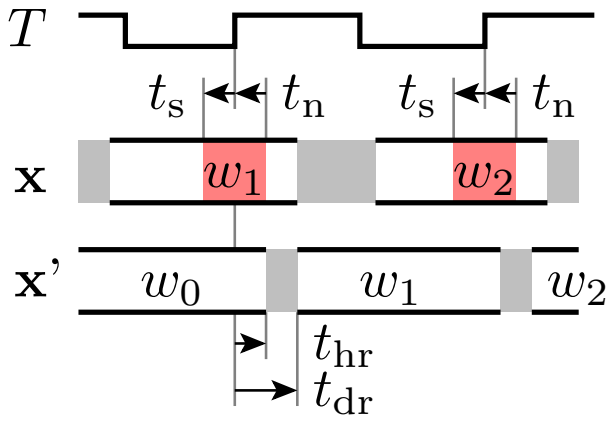
$$t_{GA} = t_{GE} + \sum_{i=1}^{3} t_{hi} - \sum_{i=1}^{3} t_{di}$$

- decreases with the length of the processing chain
- broadening of the period of validity ⇒ sampling (register)

Sampling with Registers

Sampling with Registers

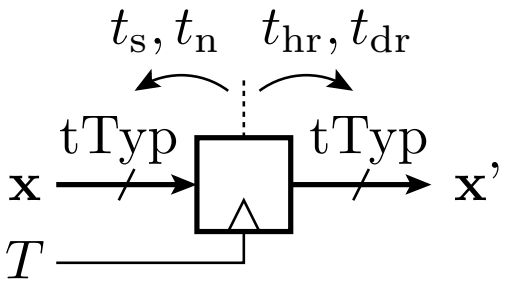


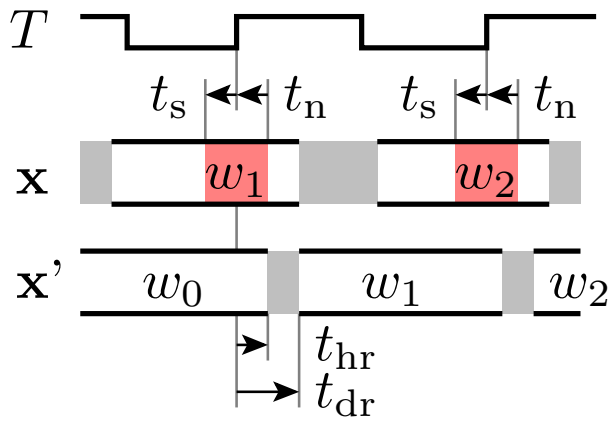


- register: samples input data with the active clock edge[1], else store
- t_s set-up time, time input data must be stable before the active clock edge
- t_n input hold time, time, the input data mus be stable after the active clock edge; will be often neglected
- t_{hr}, t_{dr} hold and delay time of the register
- tTyp: data type, »STD_LOGIC«, vector type of is, ...

[1] rising, falling edge or both

Clock Signal





Clocks are special signals, switching periodically between »0« and »1« and are used for time adjustment of data signal transitions

- modelled as ideal binary signals (without »X«)
- the delay tolerance region is described by the register parameters
- clock signals must be exactly in time and glitch free; require special circuitry

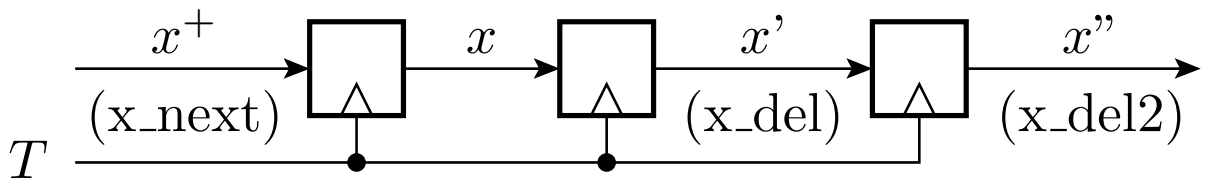
---

- active clock edge: rising, falling, both

Notation for sampled signals

Sampling is a basic function of digital circuits; definition of a notation:

- the signal sampled by a signal x will be named x^+ (following state) (in VHDL x_next)
- the sampled signal of x will be called x', the sampled signal of x' will be called x" etc. (in VHDL x_del, x_del2 etc.)



VHDL sampling process

VHDL sampling process

- behaviour of a sampling process

```
sampling_process: process(T)
 begin
  if active_clock_edge then
   output_signal <= invalid after th,
                    input_value after td;
   check_input_setup_time
   check_input_hold_time
 end if;
end process;
```

- additional required description means:
  - case distinction
  - check for clock edges (signal attribute)
  - instructions to control input setup and hold conditions

signal attribute, description means to check additional signal features:

```
signal s: tTyp;
constant t: DELAY_LENGTH;
```

s'EVENT → BOOLEAN	True, if the process has been waked-up by a transition of s
s'STABLE(t) → BOOLEAN	True, true if there was no transition since time t
s'LAST_EVENT → DELAY_LENGTH	time since the last transition
s'LAST_VALUE → tTyp	value before the last transition of s
s'DELAYED(t) → tTyp	the by t delayed signal of s
...	

- functions to check, whether a process was awakened be an active edge (Package IEEE.STD_LOGIC_1164):

```
function RISING_EDGE(signal T: STD_LOGIC) return BOOLEAN is
begin
 return T'EVENT and T='1';
end function;

function FALLING_EDGE(signal T: STD_LOGIC) return BOOLEAN is
begin
 return T'EVENT and T='0';
end function;
```

- control input setup and hold conditions:

```
s'STABLE(t); s'LAST_EVENT
```

- binary case distinction for control flow:

```
if b then
 {sequential_statement;}
end if;
```

(*b* – condition of type BOOLEAN, e.g. »RISING_EDGE(T)«

- control instructions:

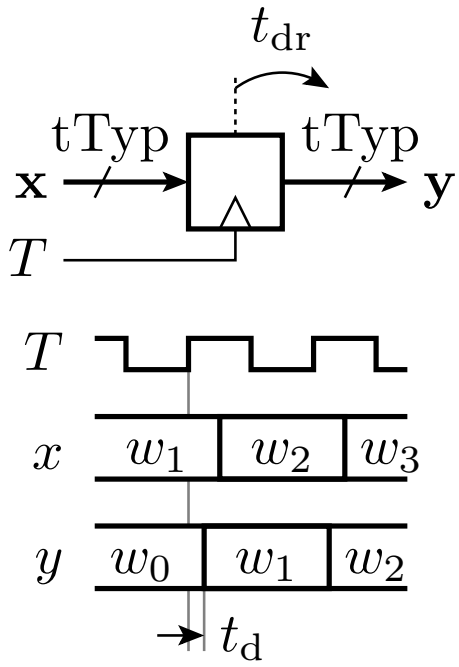
```
assert b [report m] [severity sl];
```

(*m* – additional message text; *sl* – severity level; data type:

```
type SEVERITY_LEVEL is (
  NOTE,
  WARNING,
  ERROR,
  FAILURE);
```

Simple register model

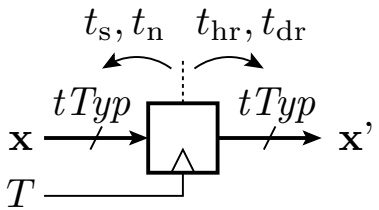
```
signal T: STD_LOGIC;
signal x, y: tTyp;
constant tdr: DELAY_LENGTH:=...;
...
process(T)
begin
 if RISING_EDGE(T) then
  y <= x after tdr;
 end if;
end process;
```

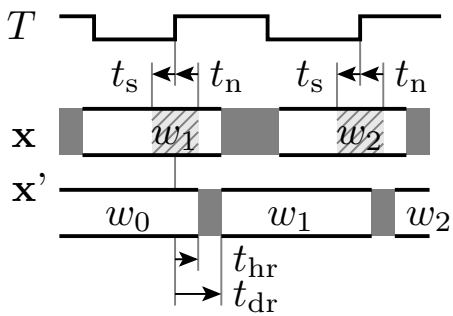


- process with only the clock in the sensitivity list
- output assignment only when a rising edge
- RISING_EDGE(T) ⇒ »T'EVENT and T='1'«; if the process awakens »T'EVENT« is »TRUE«; except when the simulation starts

A complete model

```
process(T)
begin
 if RISING_EDGE(T) then
  if x'LAST_EVENT>ts then
   y <= invalid after thr, x after tdr;
   wait for tn;
   if x'LAST_EVENT<tn then
    y <= invalid;
    report "input hold cond. violated"
        severity WARNING;
  end if;
 else
   y <= invalid after thr;
  report "setup cond. violated"
      severity WARNING;
  end if;
 end if;
end process;
```





signals:

T	clock
x	input signal
x'	(x_del) sampled (output) signal

register parameter:

t_s	setup time
t_n	input hold time
t_{hr}	(output) hold time
t_{dr}	delay time

tTyp	type data signal
w_i	signal value
	sample window
	invalid

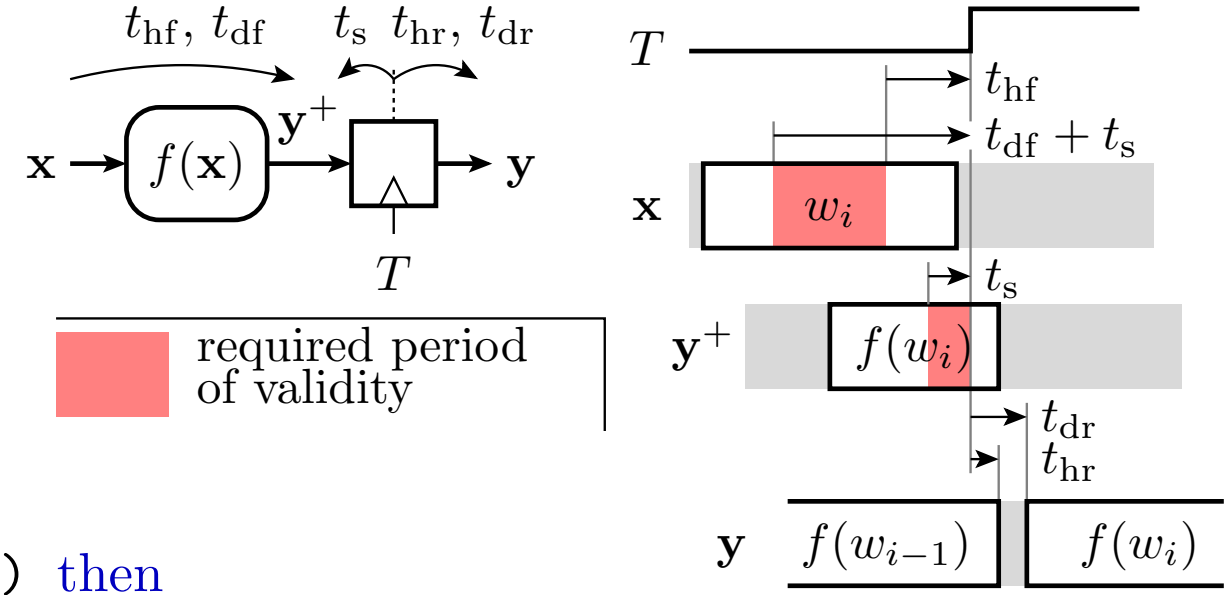
No input hold time check ($t_n = 0$) and warnings

```
process(T)
begin
 if rising_edge(T) then
  if x'last_event>ts then
    y <= invalid after thr, x after tdr;
  else
    y <= invalid after thr;
  end if;
 end if;
end process;
```

- This will be the simulation model of a register generally use in the following.

Processing + Sampling

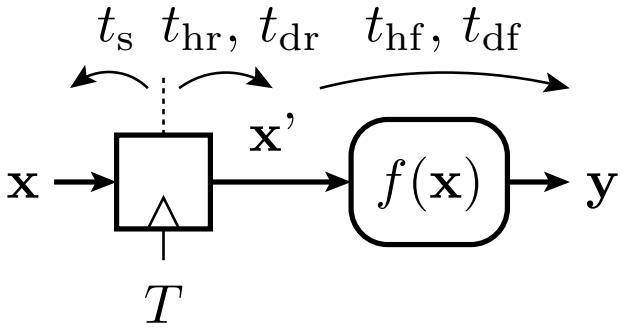
Processing and sampling of the results

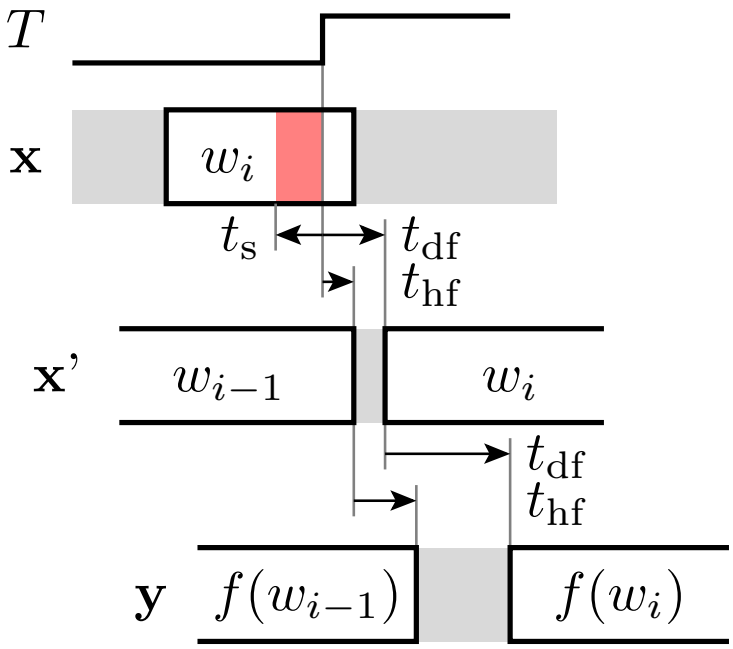



```
process(T)
 begin
  if RISING_EDGE(T) then
   if x'DELAYED(thf)'LAST_EVENT>tdf+ts-thf then
      y<= invalid(1) after thr, f(x) after tdr;
   else
      y<= invalid(1) after thr;
   end if;
  end if;
 end process;
```

(1) für STD_LOGIC 'X'; für STD_LOGIC_VECTOR "XX...X"

Processing of sampled signals



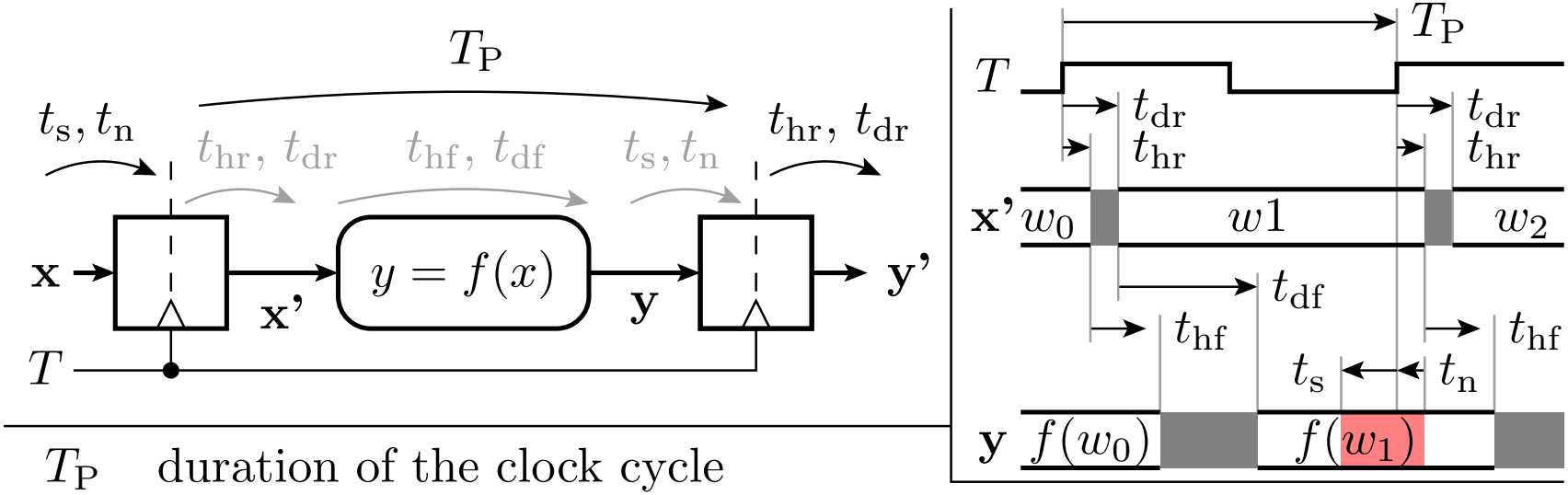



```
process(T)
begin
 if RISING_EDGE(T) then
  if x'LAST_EVENT>ts then
   y<= invalid after thr+thf,
      f(x) after tdr+tdf;
  else
   y<= invalid after thr+thf;
  end if;
 end if;
end process;
```

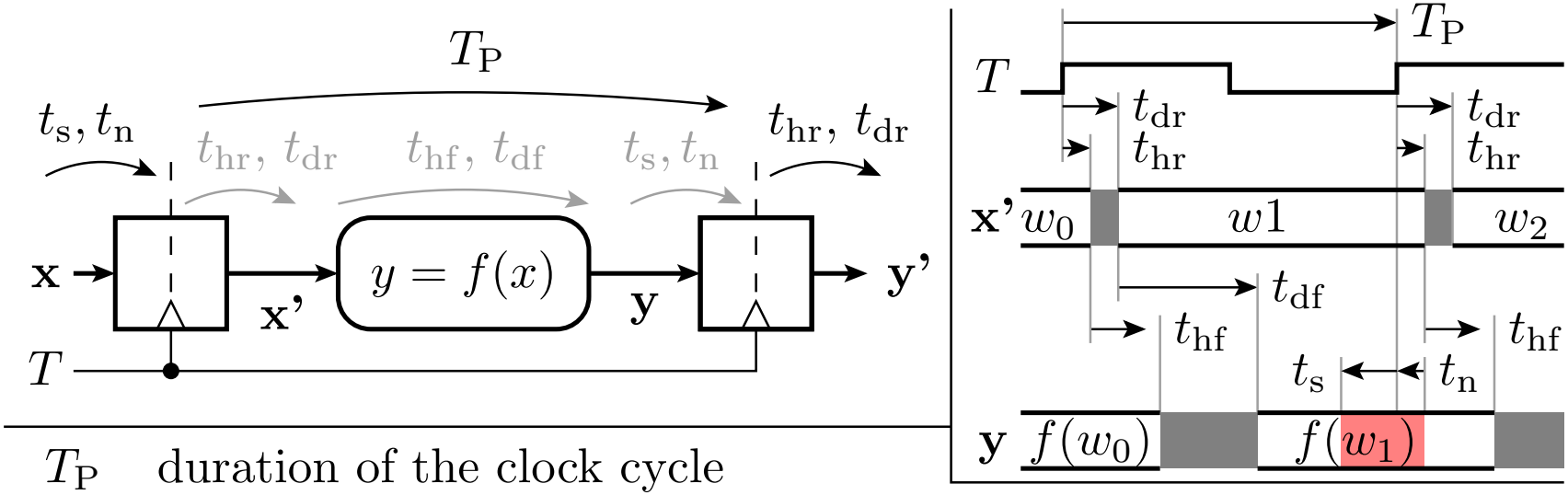
Register transfer function

Register transfer function

- combinatorial function block framed by registers



T_P duration of the clock cycle
x', y' sampled values, in the VHDL decription x_del, y_del

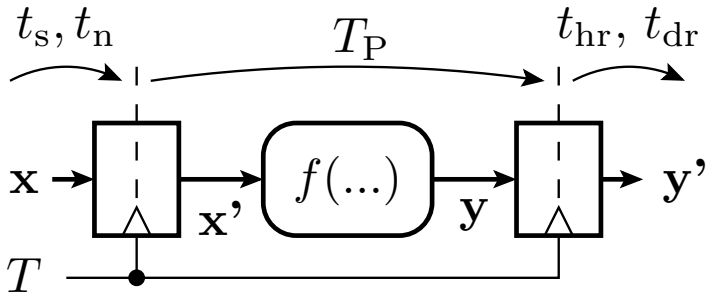


T_P duration of the clock cycle
x', y' sampled values, in the VHDL decription x_del, y_del

- Requirements to sample valid output values

$$T_P \geq T_{Pmin} = t_{dr} + t_{df} + t_s$$
$$t_n \leq t_{nmax} = t_{hr} + t_{hf}$$

- Both requirements can be checked with a run time analysis (befor simulation)
- Simulation model can be greatly simplified



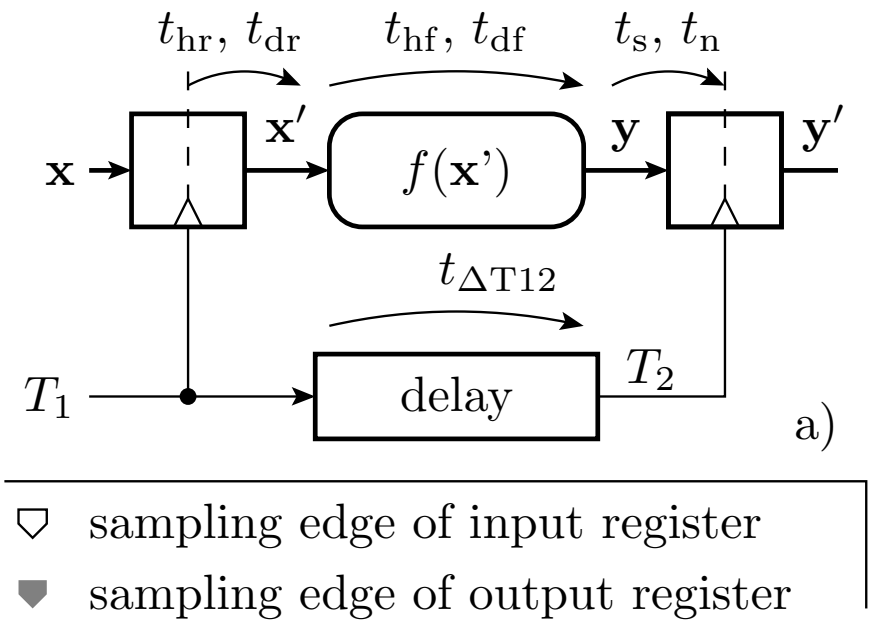

```
Processing: process(T):
begin
 if RISING_EDGE(T) then
   if x'LAST_EVENT>ts then
    x_del <= x; else x <= invalid;
   end if;
   y_del <= invalid after th, f(x_del) after td;
 end if;
end process;
```

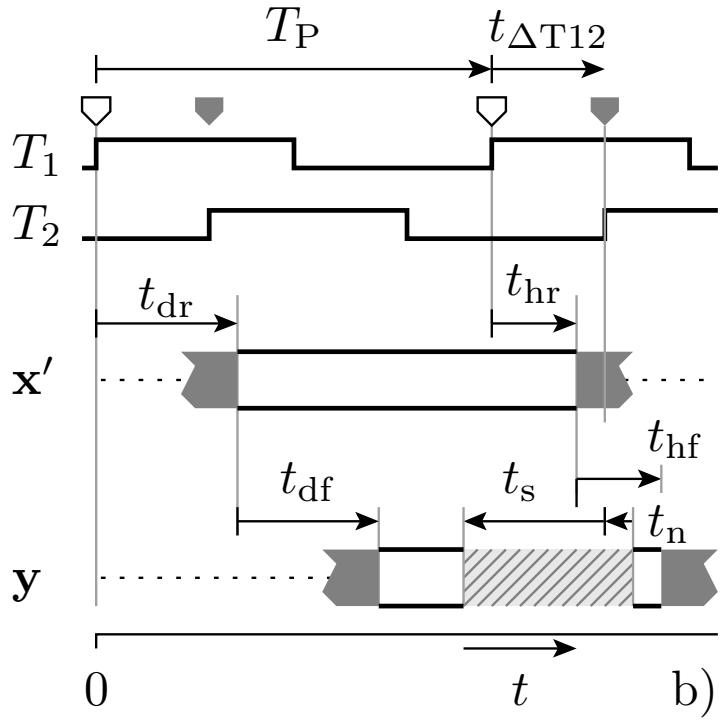
Simplification to multiple processest

- undelayed assignment to x' (x_del)
- no check of setup conditions for **y**

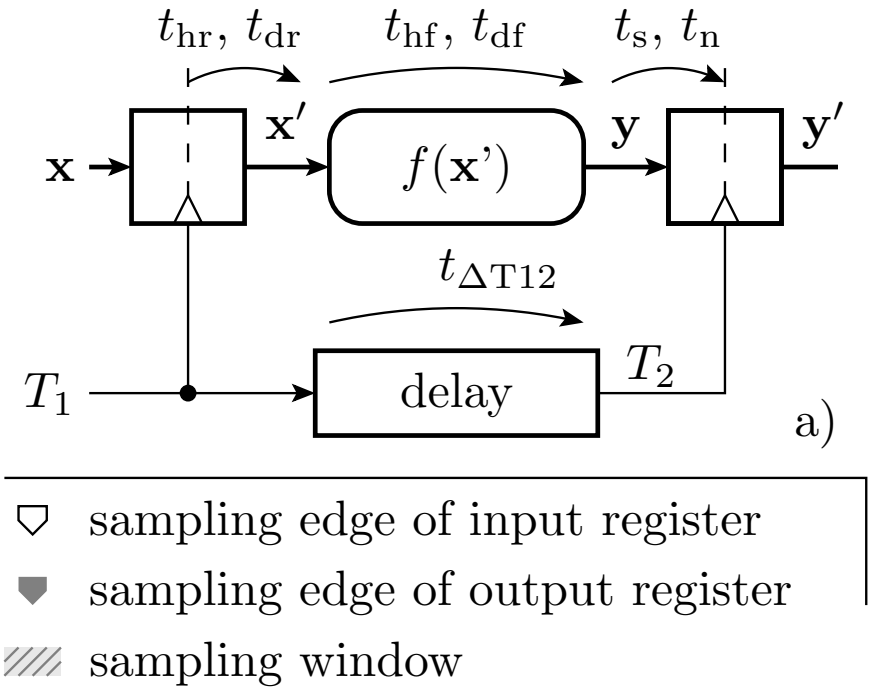
Clock skew

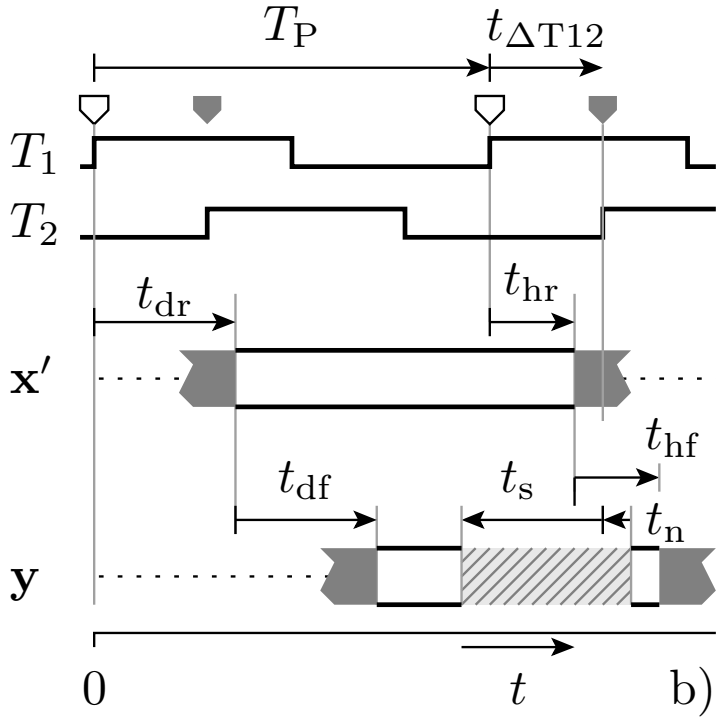
Clock skew





- Clock skew $t_{\Delta T12}$: time offset between active clock edges between input and output registers
- intended to increase the max. clock frequency
- unintended by different delays





Requirements for correct sampling:

- $t_{dr} + t_{df} + t_s \leq T_P + t_{\Delta T12}$
- $t_{hr} + t_{hf} \geq t_n + t_{\Delta T12}$

Requirements for correct sampling:

- $t_{\mathrm{dr}} + t_{\mathrm{df}} + t_{\mathrm{s}} \leq T_{\mathrm{P}} + t_{\Delta \mathrm{T12}}$
- $t_{\mathrm{hr}} + t_{\mathrm{hf}} \geq t_{\mathrm{n}} + t_{\Delta \mathrm{T12}}$

Maximum allowed clock skew:

- $t_{\Delta \mathrm{T12}} \leq t_{\mathrm{hr}} + t_{\mathrm{hf}} - t_{\mathrm{n}}$

Maximum allowed clock frequency:

$$f_{\mathrm{T}} \leq \frac{1}{t_{\mathrm{dr}} + t_{\mathrm{df}} + t_{\mathrm{s}} + -t_{\mathrm{n}} - (t_{\mathrm{hr}} + t_{\mathrm{hf}})}$$

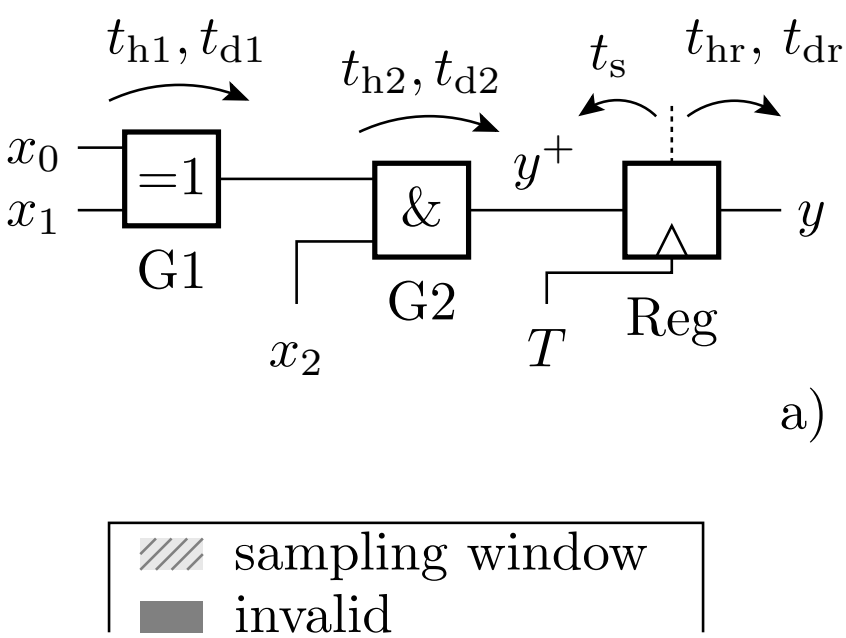
- Reciprocal value of the difference of the sum of the delay, the input hold, the setup and the output hold time.
- Circuits with a large hold time may be clocked faster with an appropriate clock skew then without

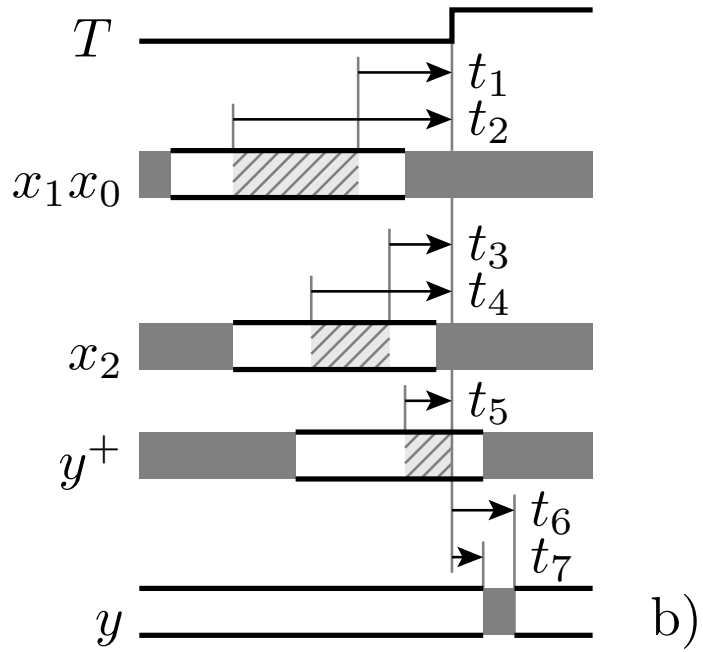
Summary

- Calculation results must be sampled within their periods of vality by registers.
- Registers are discribed by sampling processes. In a sampling process only the clock is in the sensitivity list. Sampling with the active clock edge. Setup or input hold condition violations invalidate register data.
- The description of the the combinatirial circuit before or after the register may include in the sampling processes; simplify simulation.
- Framing by an input and an output register; timing condition check by run time analysis; highly simplified simulation model.

Exercises

Aufgabe 1.12: Desciption by a sampling process

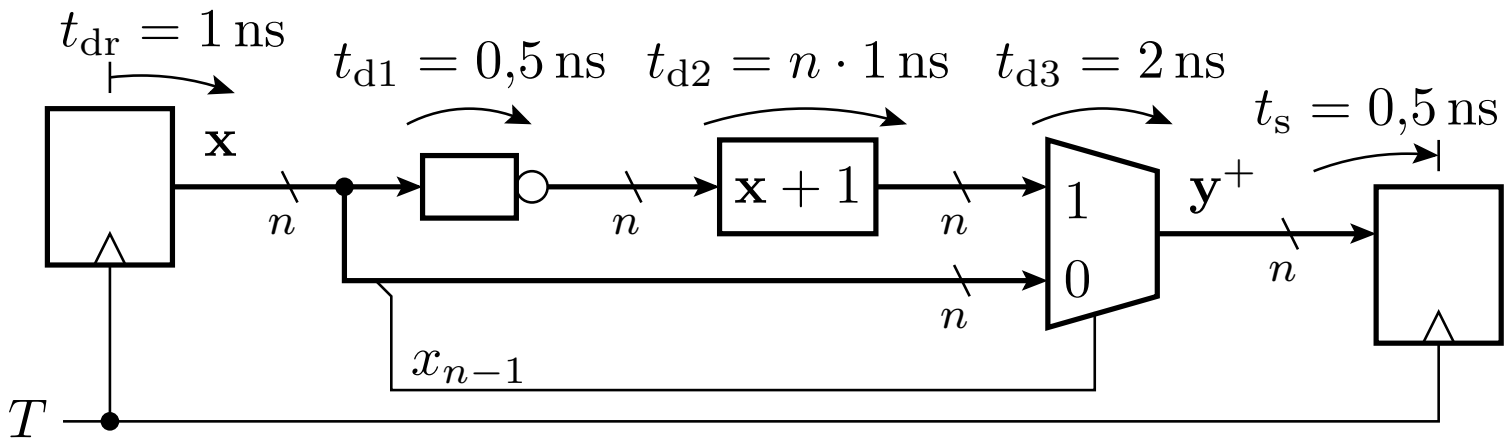




signal x0, x1, x2, y_next, T, y: STD_LOGIC;

- Describe t_1 to t_7 as funcions of the values of $t_{h..}$, $t_{d...}$ etc.
- Describe the whole circuit as a sampling process

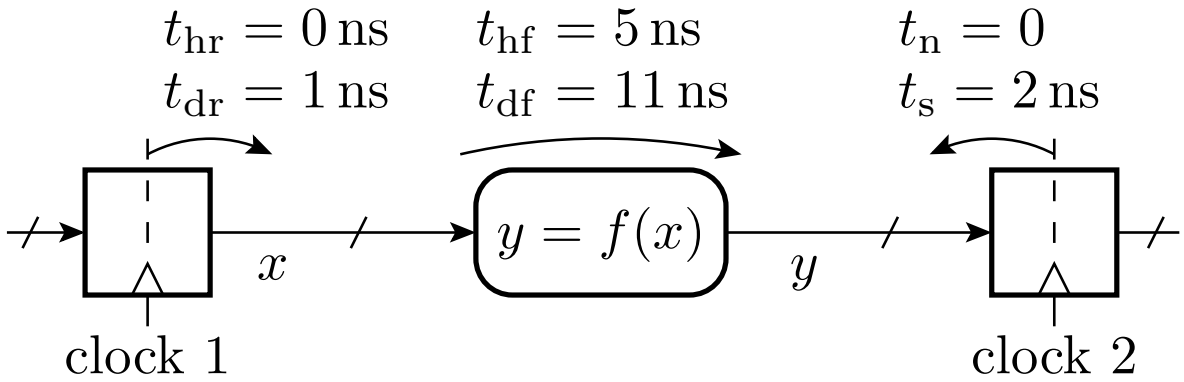
Aufgabe 1.13: Register transfer function



Determin:

- maximum allowed clock frequency as a function of the bit width n?
- maximum allowed clock frequency for $n = 16$?

Aufgabe 1.14: Clock skew



In which interval of time the clock skew has to be, so that for a clock frequncy $f_{Clk} = 100\,MHz$ the setup an the input hold conditions are met?

Asynchronous input

- asynchronous: »without synchronisation«
- the time delay between input change and the active clock edge is a transactions is a uniformly distributed random variable in the range:

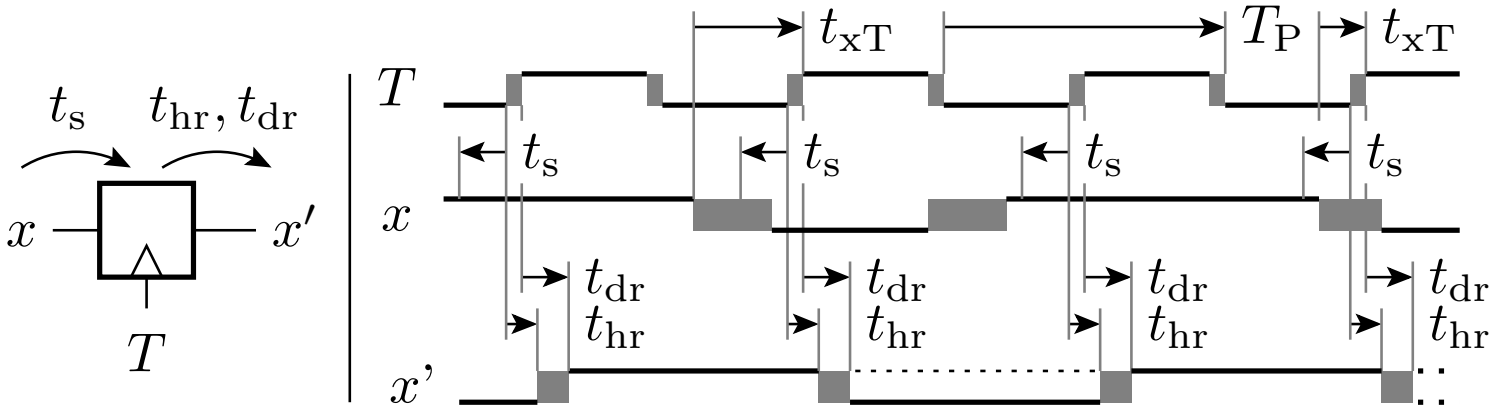
$$t_{\mathrm{xT}} \in \{0, T_{\mathrm{P}}\}$$

- also asyncronous signals may only be evaluated within their periods of vality
- special circuit solutions and VHDL descriptions are required

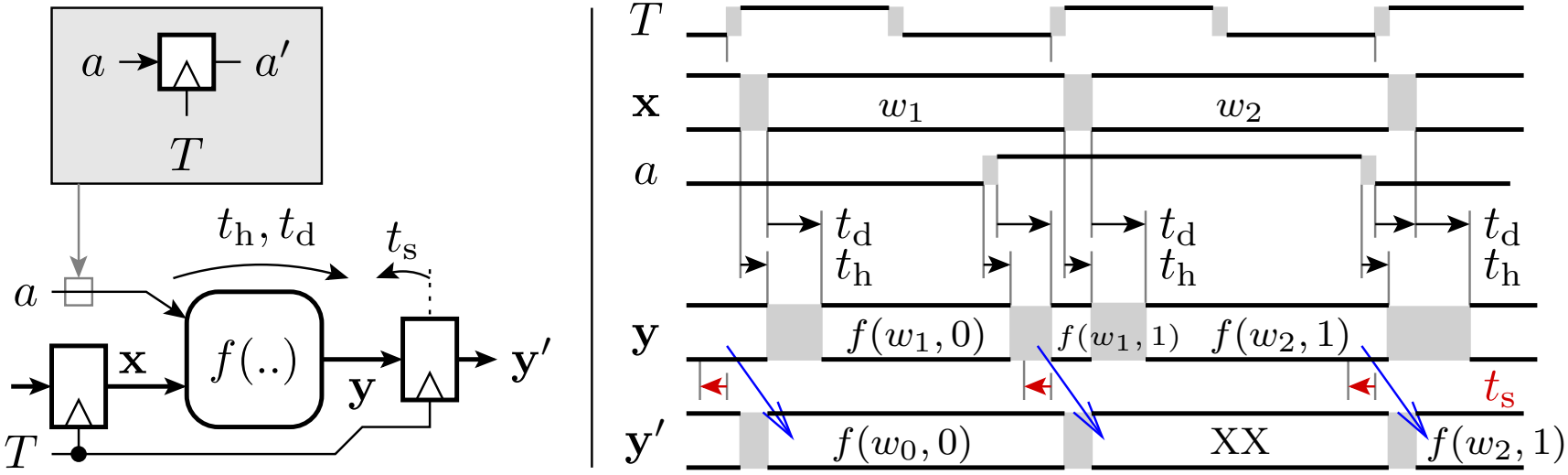
Sampling a single bit

Sampling of binary input signals

- a single bit has, even if invalid during the sampling period, has the sampling value »0« or »1«; sampled signals are free of glitches and stable during the clock period



Design error: Processing of unsampled inputs

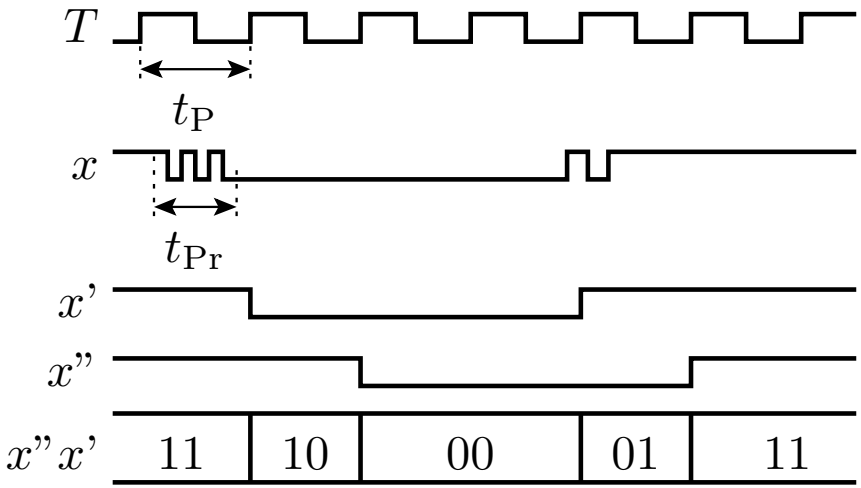


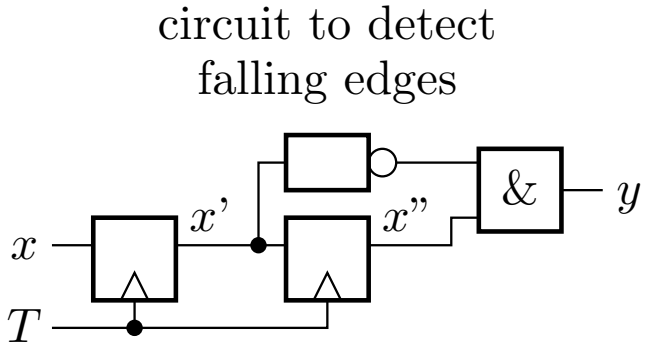
- (processed) asynchronous inputs are withe a sertain probability invalid during the sample period
- the sampled value of »invalid« is a random value in $\mathbf{a}' \in \{0,1\}^n$, mostly wrong, often anallowed, unpredictable behaviour
- Workaround: additional bitwise sampling of the asyncronous input

Switch de-bouncing

De-bouncing of signals from switches and keys

- typ. 1 bit input element: mechanical switch or key
- Bouncing: multiple on and off by mechanical vibrations
- de-bouncing: sampling with a period larger than the bouncing time
- Edge detection: sampling twice, check for difference

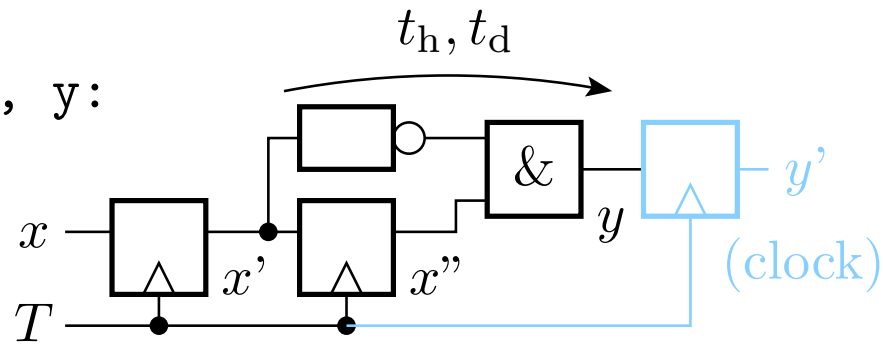




t_{Pr}	maximum bouncing time
$t_P > t_{Pr}$	sample period

Take care, without unreliable sampling!

```
signal T, x, x_del, x_del2, y:
  STD_LOGIC;
...
process(T)
begin
 if RISING_EDGE(T) then
   if x='1' then x_del<='1';
   else '0'; end if;
   x_del2 <= x_del;
 end if;
end process;
y <= 'X' after th not x_del and x_del2 after td;
```

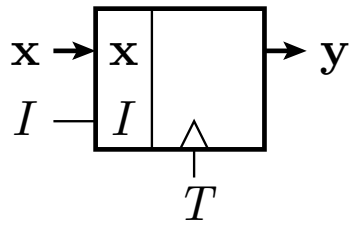


- Sampling x: »'1'↦'1', sonst↦'0'« (template for »sampled_value∈ $\{0,1\}$
- combinatorial output, shortly invalid after each active clock edge; may be glitches
- to use y as a clock, sample again

Asynch. initialization

Register with initialization input

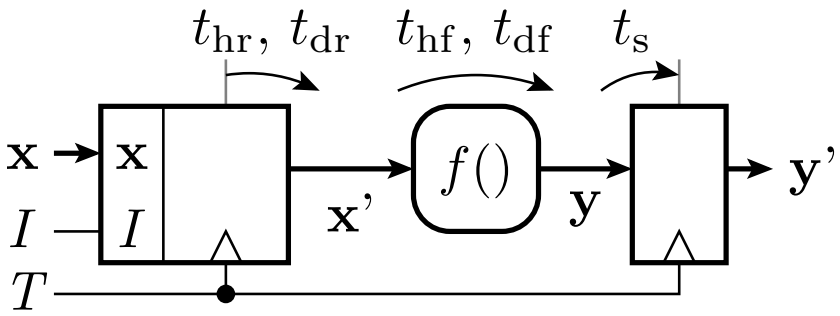
```
signal x, y: tTyp;
signal T,  I: STD_LOGIC;
constant aw: tTyp:=...;
...
process(T, I)
begin
 if I='1' then
  y <= aw;
 elsif RISING_EDGE(T) then
   y <= x;
 end if;
end process;
```

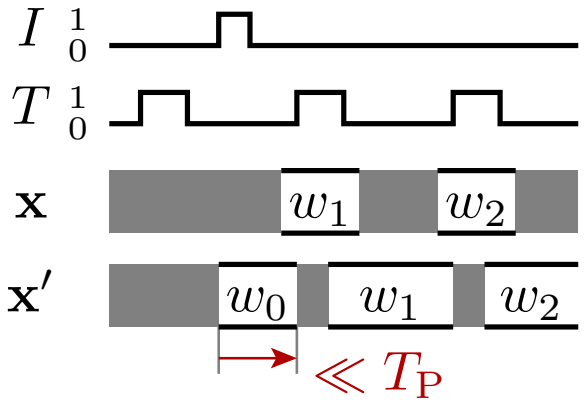


- register stage before initialization "UU...U" (uninitialized/invalid)
- ≫aw≪ any valid value

Potential malfunction

initialization pulse to short



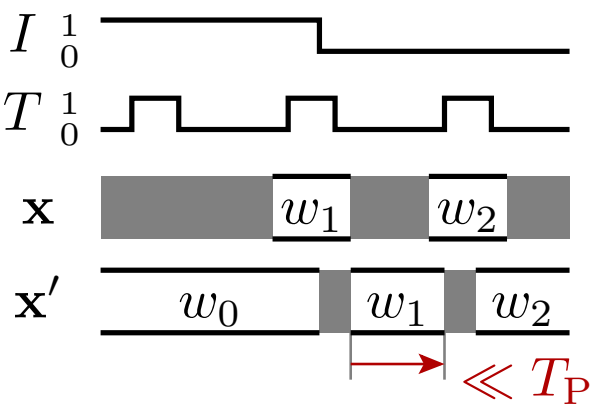


- Even if the input register is initialized correctly, duration ist to short, in witch the initial value is stable at the register output. So the subsequent register may sample an invalid signal.

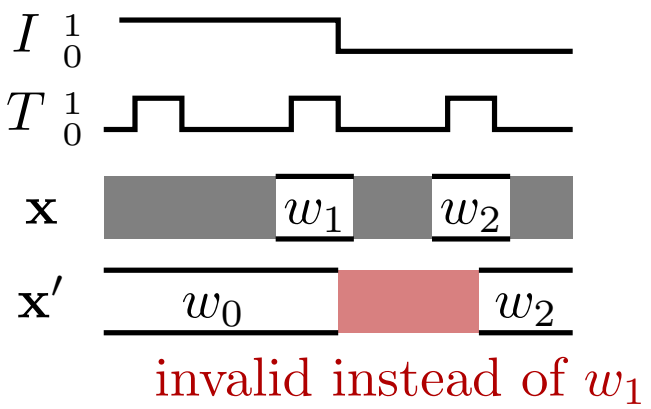
Register of master-slave flipflops:

- master takes over before the active clock edge
- slave takes over after the active clock edge
- I initialization of the slave

deactivation of I in the slave phase

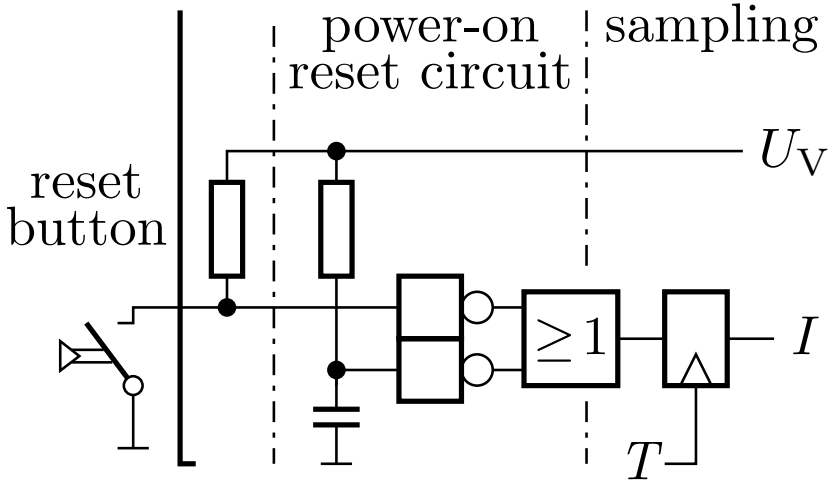


deactivation of I at the end of the slave phase



Robust initialization

```
signal T, I,  I_POR,
    I_Tast: STD_LOGIC;
...
I_POR <= '1', '0'
  after 1 ms;
process(T)
begin
 if RISING_EDGE(T) then
  if I_POR='0' or I_Tast='0' then I<='1';
  else I<='0';
  end if;
 end if;
end process;
```

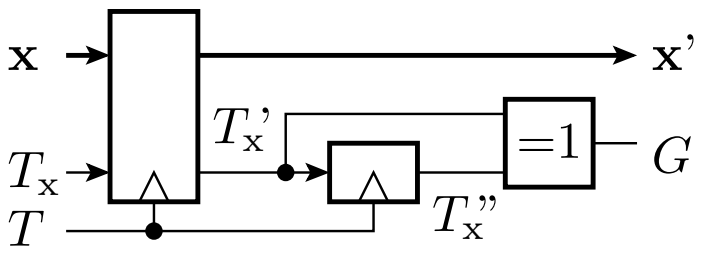


- power-on reset: after applying voltage active for $t_{POR} \approx R \cdot C$
- sampling to align to the active clock edge

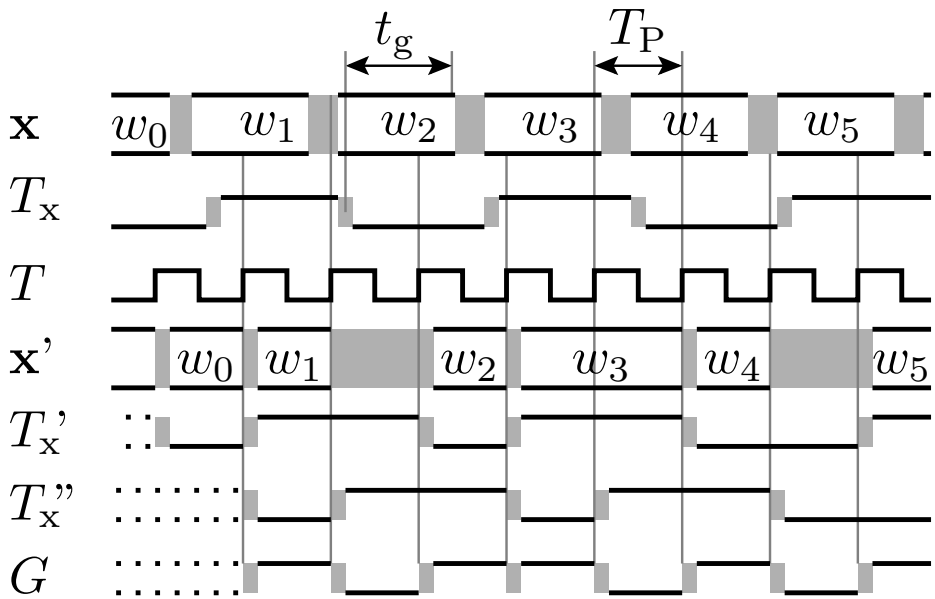
Parallel interface

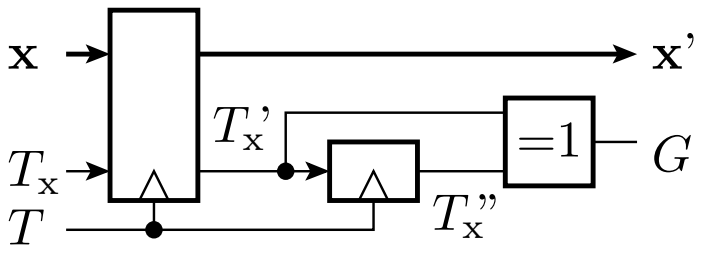
Asynchronous parallel interface

- the receiver of asynchronous parallel data need additional information about the period of validity, e.g. the clock of the transmitter
- in the example x should be valid from each edge of T_x for a duration of t_g

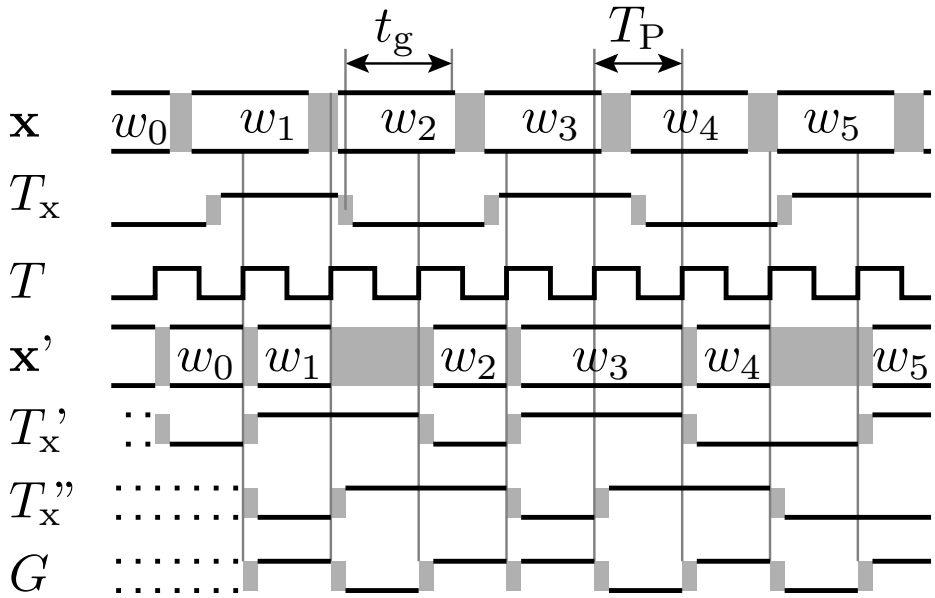


x	asynchronous input signal
T_x	send clock
T	system clock
x'	sampled input signal
G	validation signal for **x'**

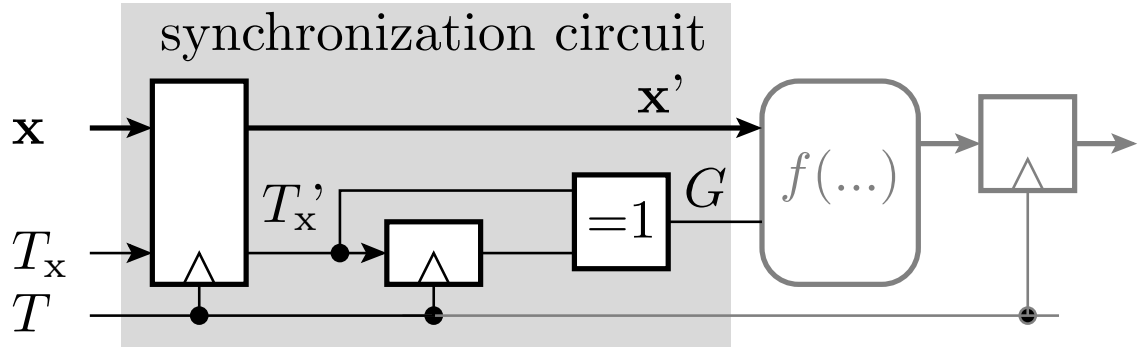




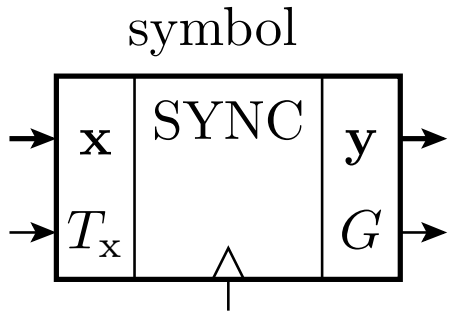
x	asynchronous input signal
T_x	send clock
T	system clock
x'	sampled input signal
G	validation signal for **x'**



- all input signals sampled with a period $T_P < t_g$
- the clock is sampled twice
- sampled data are valid if the sampled transmitter clock differs from the twice sampled




```
signal T, Tx, Tx_del, Tx_del2, G: STD_LOGIC;
signal x, y: STD_LOGIC_VECTOR(...);
  ...
process(T)
begin
 if RISING_EDGE(T) then
   y<= x;
   if Tx='1' then tx_del <= '1';
  else tx_del <= '0';
  end if;
   tx_del2 <= Tx_del;
 end if;
end process;
G <= Tx_del xor Tx_del2;
```

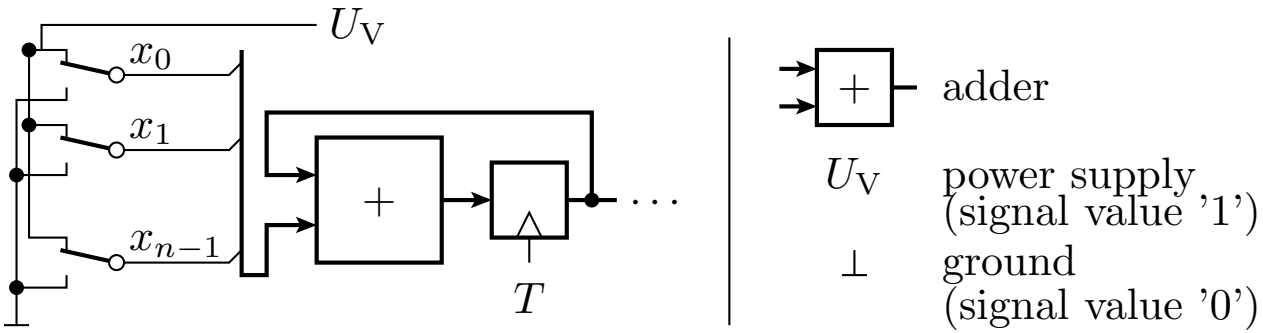


Summary

- sample asyncronous signals before processing
- in addion de-bounce signals from mechanical switches; sample period larger bouncing time
- sample asynchronous initialization signals
- asynchronous parallel input signals needs additional validity information; sampling by a synchronization circuit
- forgotten sampling causes non-recurring malfunctions; difficult to locate; reduced reliability

Exercises

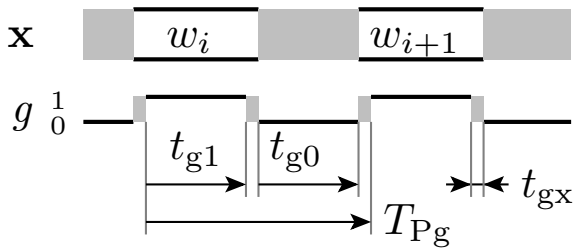
Aufgabe 1.15: Adding up asynchronous input



- Under which operational conditions the circuit may have non-recurring malfunctions?
- How the circuit has to be changed to work reliable?

Aufgabe 1.16: Asynchronous parallel transmission

During an asynchronous transmission data signal x should be valid if the also transmitted validity signal g is $g \neq 0$:



t_{g0}	min. time $g = 0$
t_{g1}	min. time $g = 1$
t_{gx}	max. time $g =$ X
T_{Pg}	periodic time of g

- What clock frequency is required in the receiver circuit to sample x and g[2]?
- How the validity signal in the receiver has to be generated, so that it is exactly valid for one receiver clock for each sampled data word?

[2] Duration of $g = 0$ and $g = 1$ should be equal.

Aufgabe 1.17: Hand clock

Design a circuit with an input button x, en input clock T ($f_\mathrm{T} = 50\,\mathrm{MHz}$) and internal clock divider to produce a de-bounced hand clock without glitches:

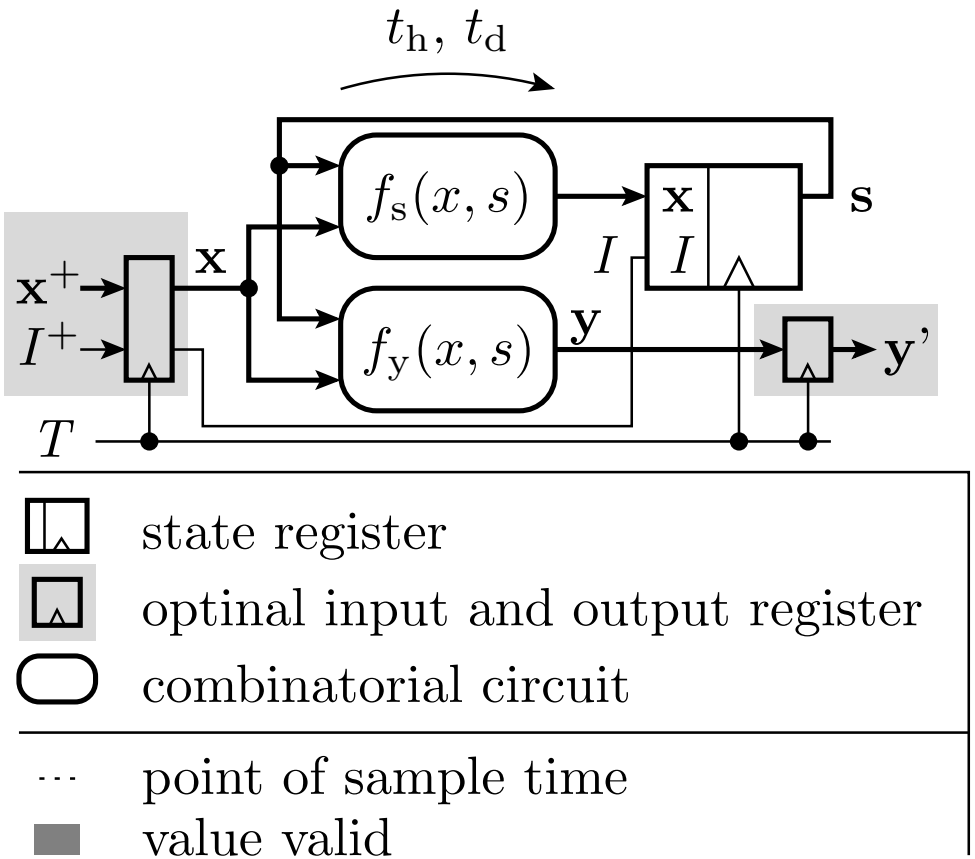
```
entity HandClock is
 port(x: in std_logic;      -- input signal from button
      T: in std_logic;      -- input clock, 50MHz
      Tout: out std_logic); -- hand clock, de-bounced, no glitche
end entity;
```

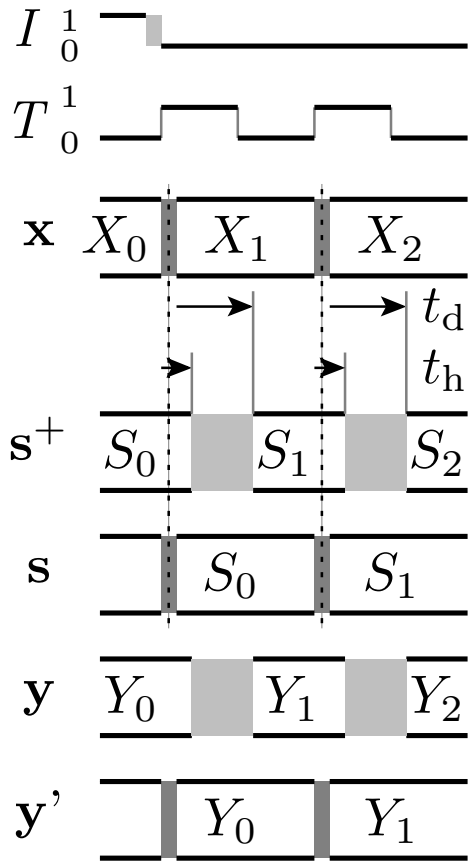
- Maximum bouncing time 20 ms
- Minimum activation time 100 ms.
- Divider proportion for the clock should be a power of two.

1. What is the advantage of a power of two for the divider proportion? (Note, the circuit has no initialization input.)
2. What values are allowed for the divider proportion?
3. Draw the whole circuit with the clock divider as a black box.
4. Describe the complete circuit in VHDL.

Sequential circuit

Sequential circuit





- sequential: in consecutive steps
- Calculation of the circuit outputs in multiple time steps
- combinatorial circuit + registers for sampling and storing

Finite state machine

Finite state machine

general function model to describe sequential operations; defining value ranges as symbol sets:

- range of input values: $\Sigma = \{E_1, E_2, \ldots\}$
- range of states: $S = \{Z_1, Z_2, \ldots\}$ (one is initial state)
- range of output values: $\Pi = \{A_1, A_2, \ldots\}$

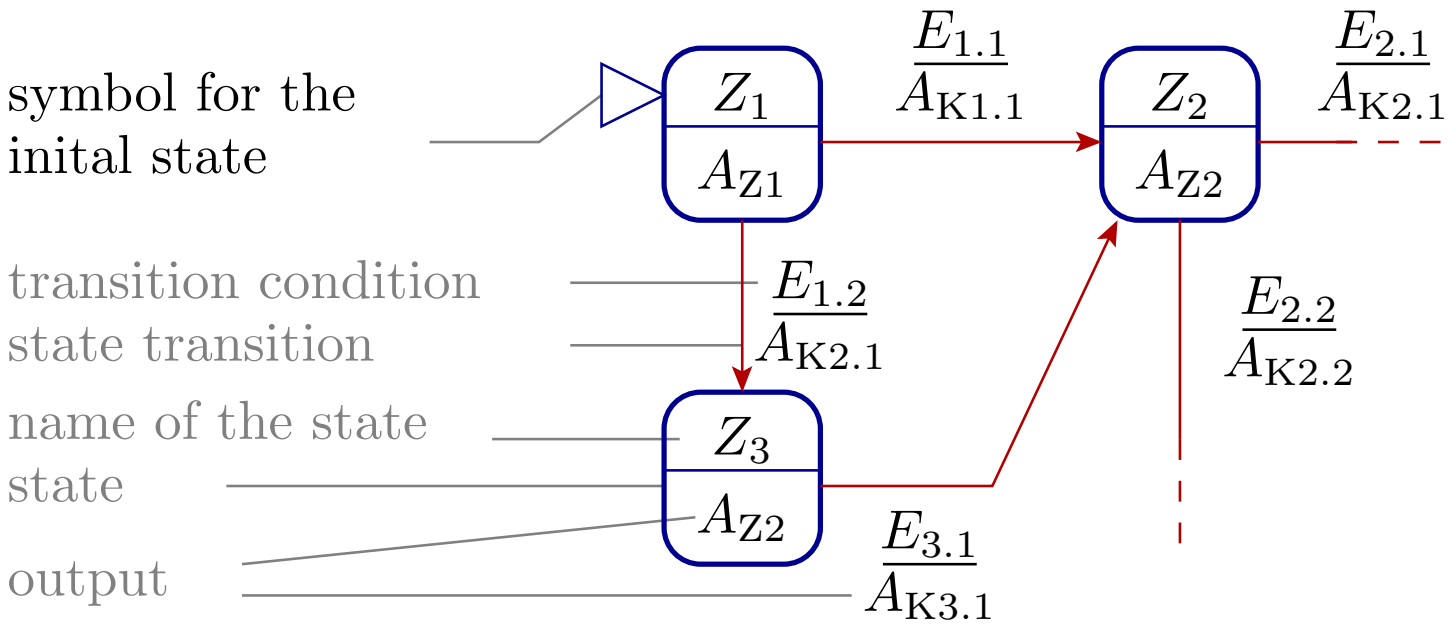
and operation as mappings:

- transfer function: $f_\mathrm{s} : S \times \Sigma \to S$
- output function: $f_\mathrm{a} : S \times \Sigma \to \Pi$

Digital circuits as finite state machines:

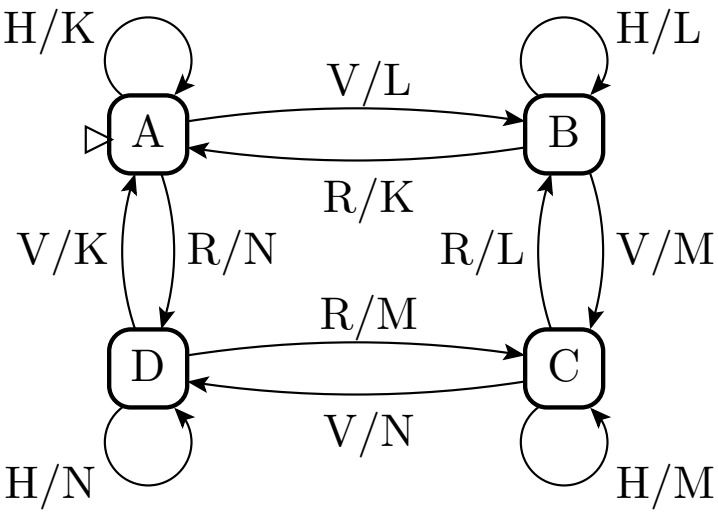
- symbols $\mapsto$ bit vectors; input, output $\mapsto$ signals
- state $\mapsto$ register; transfer and output function $\mapsto$ combinatorial circuit
- state transition synchronized to a clock
- initial state = initialing value of the state register

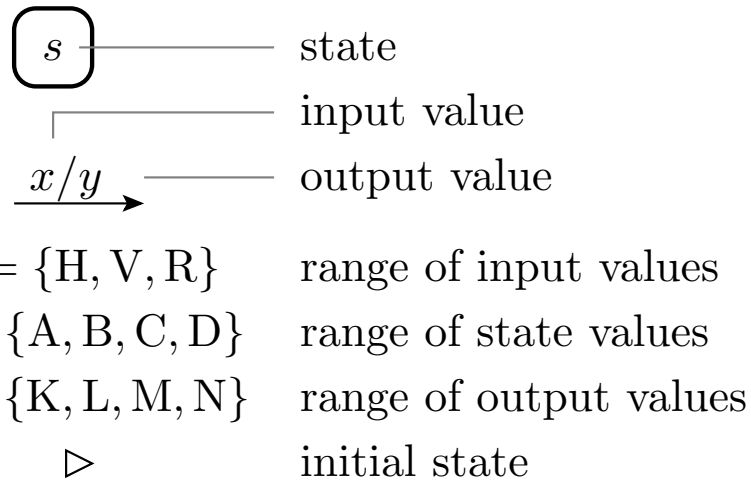
Description by a state graph



- output depends only upon the state / assigned to states / the same for all outgoing edges $\mapsto$ Moore automaton
- output also depends on input / assigned to the outgoing edges $\mapsto$ Mealy automaton

Up-/down counter as a Mealy automaton





Meanings of the input values: H – stop; V – up; R – down

state transition and output function with symbolic values ⇒ state coding ⇒ state transition and output function with bitvectors

	$s^+ = f_s(\mathbf{x}, \mathbf{s})$	$y = f_a(\mathbf{x}, \mathbf{s})$
s	x : V H R	V H R
A	B A D	L K N
B	C B A	M L K
C	D C B	N M L
D	A D C	K N M

V	00
H	01
R	10
A	00
B	01

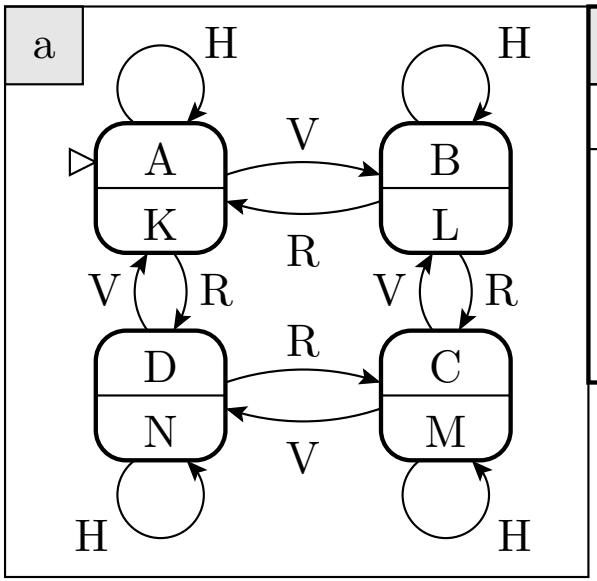
C	10
D	11
K	00
L	01
M	10
N	11

	$\mathbf{s}^+ = f_s(\mathbf{x}, \mathbf{s})$	$\mathbf{y} = f_a(\mathbf{x}, \mathbf{s})$
s	x : 00 01 10	00 01 10
00	01 00 11	01 00 11
01	10 01 00	10 01 00
10	11 10 01	11 10 01
11	00 11 10	00 11 10

- state graph: obviously well suited to specify target functions
- extractable tabular state transfer and output function
- state coding: assigning bit vectors to symbols
- translation to a presentation with bit vectors

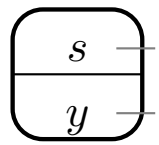
Up-/down counter as a Moore automaton

a



b	$\mathbf{s}^+ = f_\mathrm{s}(\mathbf{x},\mathbf{s})$			$\mathbf{y} = f_\mathrm{y}(\mathbf{s})$
s	x : V	H	R	
A	B	A	D	K
B	C	B	A	L
C	D	C	B	M
D	A	D	C	N

$\{\mathrm{H},\mathrm{V},\mathrm{R}\}$	input set
$\{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D}\}$	state set
$\{\mathrm{K},\mathrm{L},\mathrm{M},\mathrm{N}\}$	output set

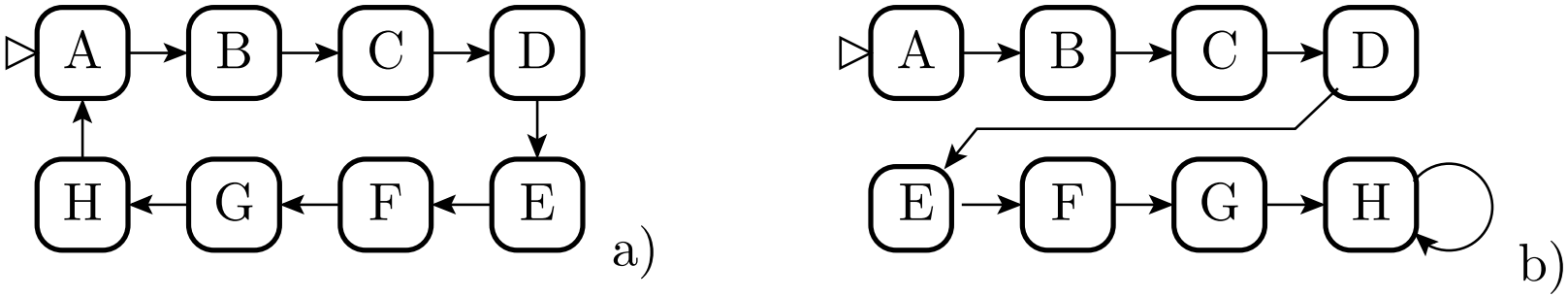


s	state
y	output value
x	input value
$\mathbf{x}$	input
$\mathbf{s}$	actual state
$\mathbf{s}^+$	next state
$\mathbf{y}$	output
$f_\mathrm{s}(...)$	transfer function
$f_\mathrm{a}(...)$	output function

- the output depends on the state

Autonomous automaton

- no input; one outgoing edge per state



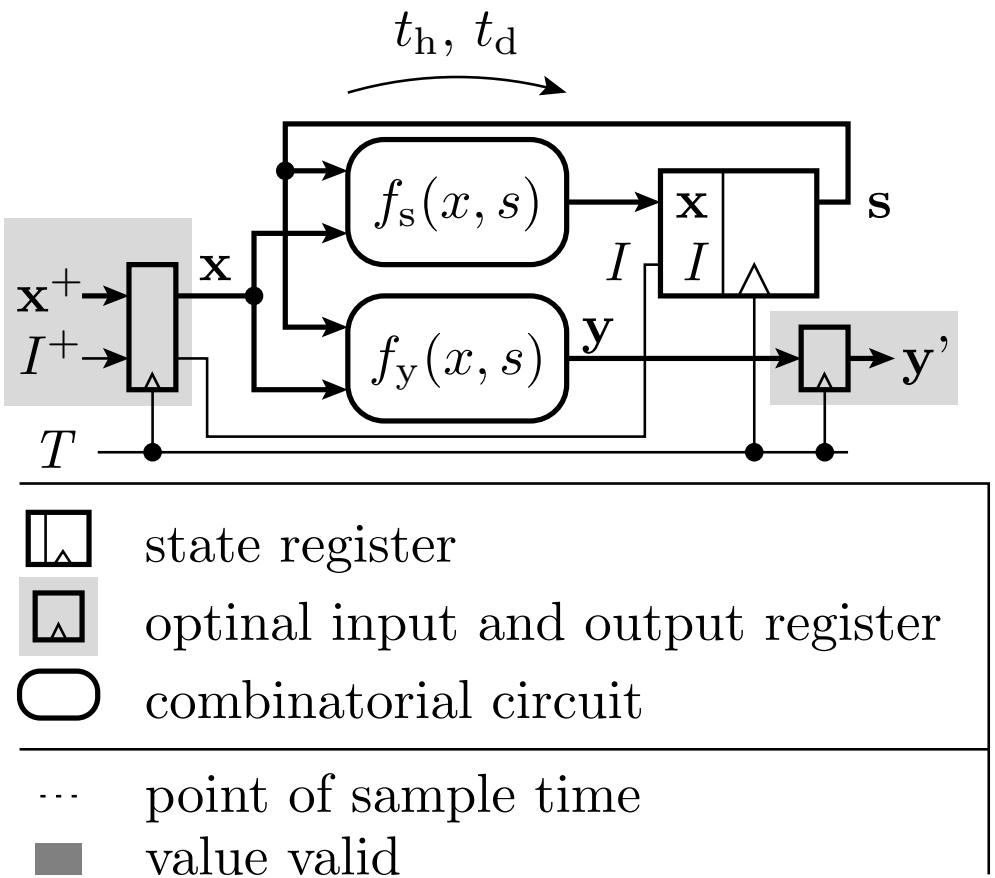
a) cyclical automaton; example application clock divider
b) without cycle but a final state; example initialization run

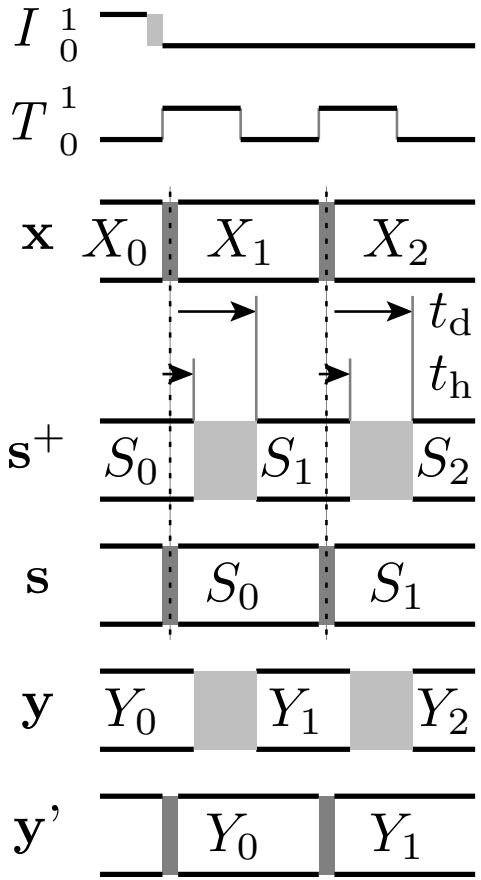
---

If no outputs are assigned in the state graph the state is the output.

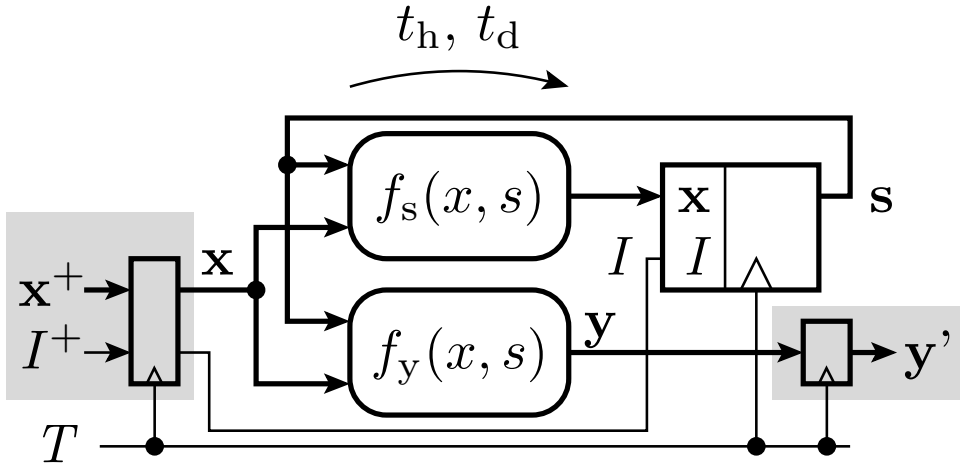
Automaton ⇒ VHDL

Circuit structure of automatons





- the input and the initialization signal must be aligned to the clock

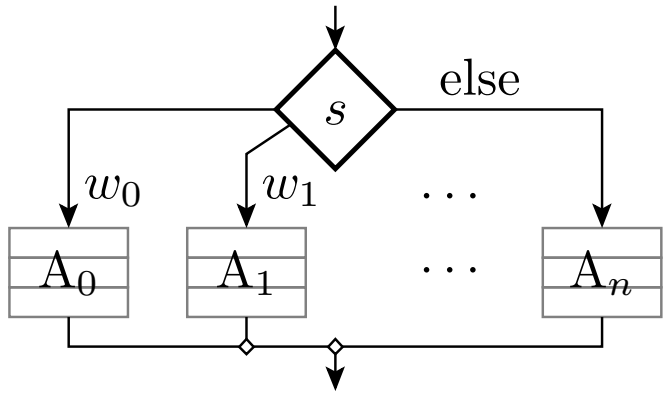


- the optional input and output register are generally not counted as part of the automaton
- initializing the state register ⇒ switching to the initial state
- state register + transfer function ⇒ sampling process with initialization (T, I in the sensitivity list)
- output function without sampling ⇒ combinatorial process (**x**, **s** in the sensitivity list)

VHDL case instruction to describe tabular functions

case *expression* **is**
when *value* { |*Wert* } =>
instruction { *instruction*}
{**when** *Wert* { |*Wert* } =>
instruction { *instruction*}}
⟦**when others** =>
instruction { *instruction*}⟧
end case ;

s
w_0
A_0
w_1
A1
A_n

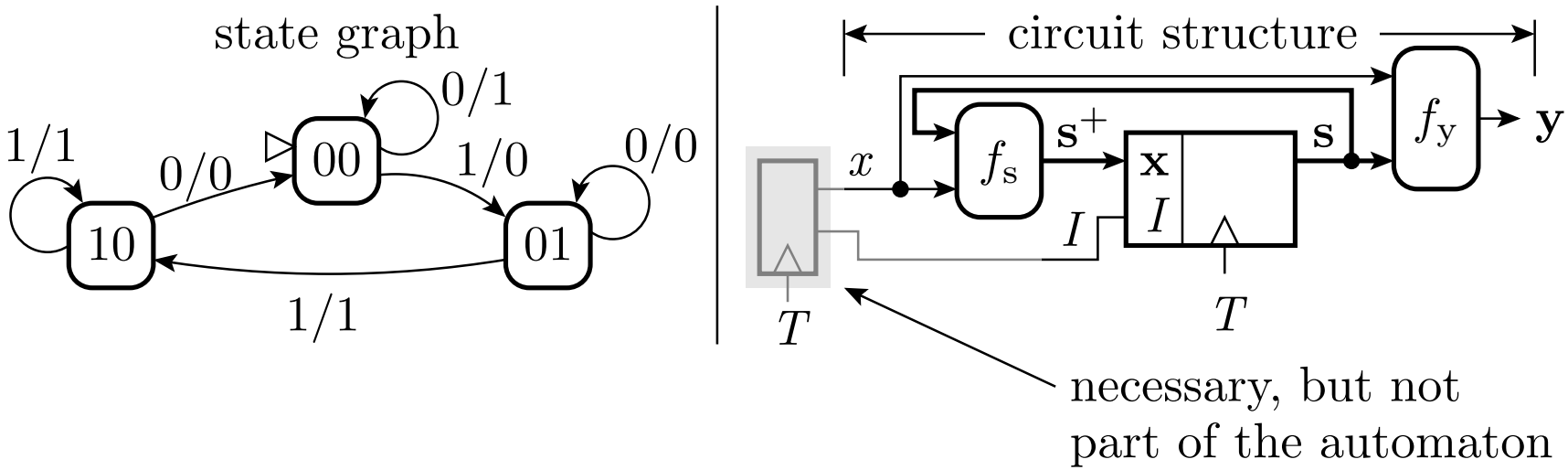


s value to select
w_i values to bee selected i
A_i instruction sequence i

- the transfer an the output functions derived from the state graph best be described by case and if statements, e.g.:

```
case state is
 when Zi => if input=... then state <= Zj; ...
```

Example of a mealy automaton

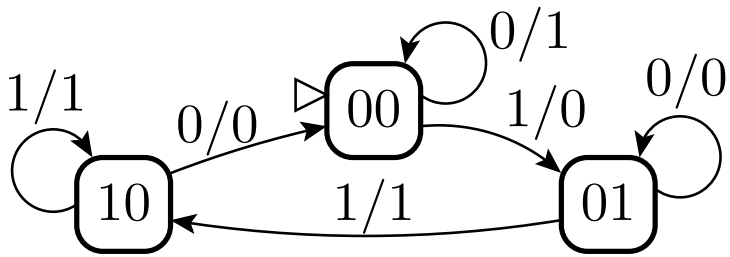


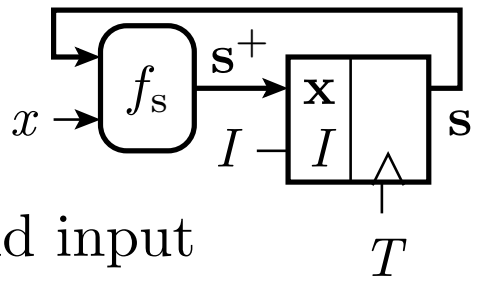
- input range: $\{0, 1\} \Rightarrow$ bit
- Range oft states: $\{00, 01, 10\} \Rightarrow$ 2 bit vector
- output range: $\{0, 1\} \Rightarrow$ bit

```
signal x, y, T, I: STD_LOGIC;
signal s: STD_LOGIC_VECTOR(1 downto 0);
```

Transfer function in a sample process

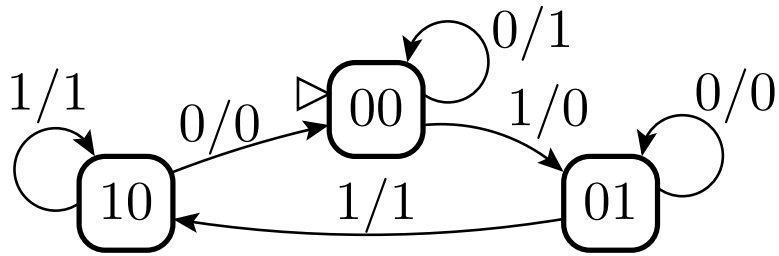
```
process(I, T)
 variable sx: STD_LOGIC_VECTOR(2 downto 0);
begin
 if I='1' then
  s <= "00";
 elsif RISING_EDGE(T) then
 sx := s & x;
 case sx is
  when "00"&'0' | "10"&'0' => s <= "00";
  when "01"&'0' | "00"&'1' => s <= "01";
  when "10"&'1' | "01"&'1' => s <= "10";
  when others              => s <= "XX";
 end case;
 end if;
end process;
```





Selection expression: concatenation of state and input

Output function as combinatorial process

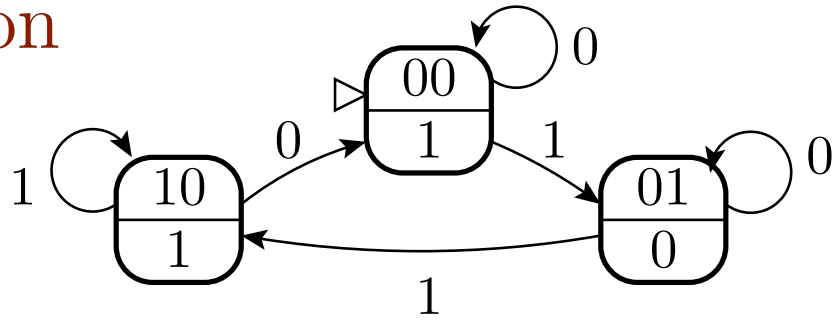


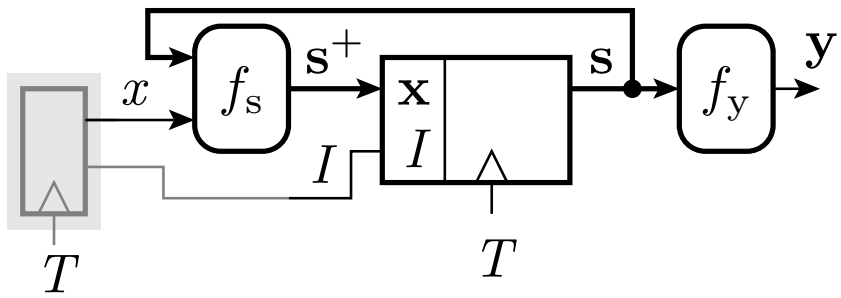

```
process(x, s)
 variable sx: STD_LOGIC_VECTOR(2 downto 0);
begin
 sx <= s & x;
 case sx is
   when "00"&'1' | "01"&'0' | "10"&'0' => y <= '0';
   when "00"&'0' | "01"&'1' | "10"&'1' => y <= '1';
    when others => y <= 'X';
  end case;
end process;
```

The selection expression is again a concatenation of the state and the input.

Example Moore automaton

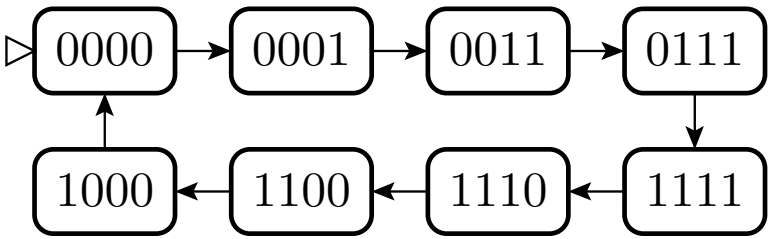
```
process(s)
begin
 case s is
  when "01"   => y <= '0';
  when "00"|"10"  => y <= '1';
  when others => s <= 'X';
 end case;
end process;
```

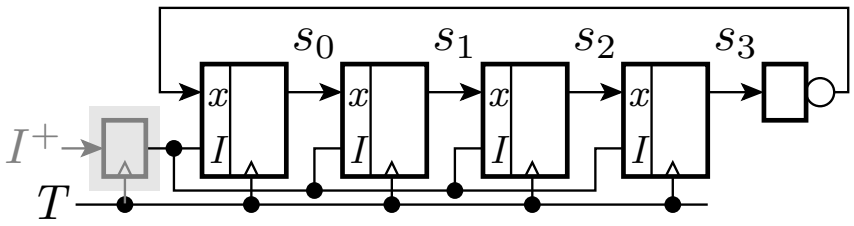




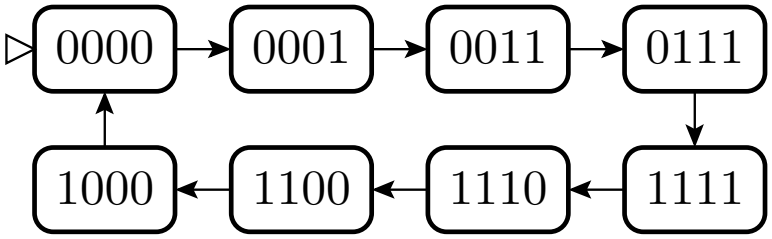
- input, state and output range and transfer function are the same as in the example before
- the output only depends upon the state

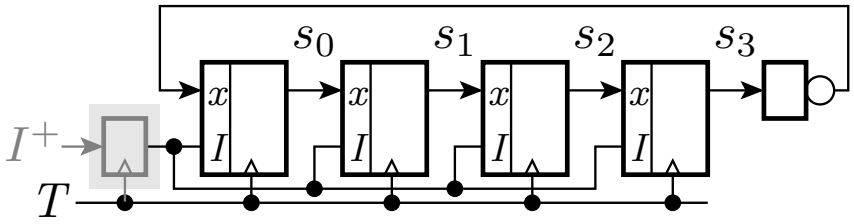
Autonomous Automaton, example Johnson counter





- shift register, that alternating filled first with ones and than with zeros
- cycle length $2 \cdot n$ (n – number of register bits)
- very low register register delay; high clock frequency
- application as fast prescaler, e.g. to measure frequencies in the GHz range






```
signal T, I: STD_LOGIC;
signal s: STD_LOGIC_VECTOR(3 downto 0);
...
process(I, T)
begin
 if I='1' then s <= "0000";
  elsif RISING_EDGE(T) then
    s <= s(2 downto 0) & (not s(3));
  end if;
end process;
```

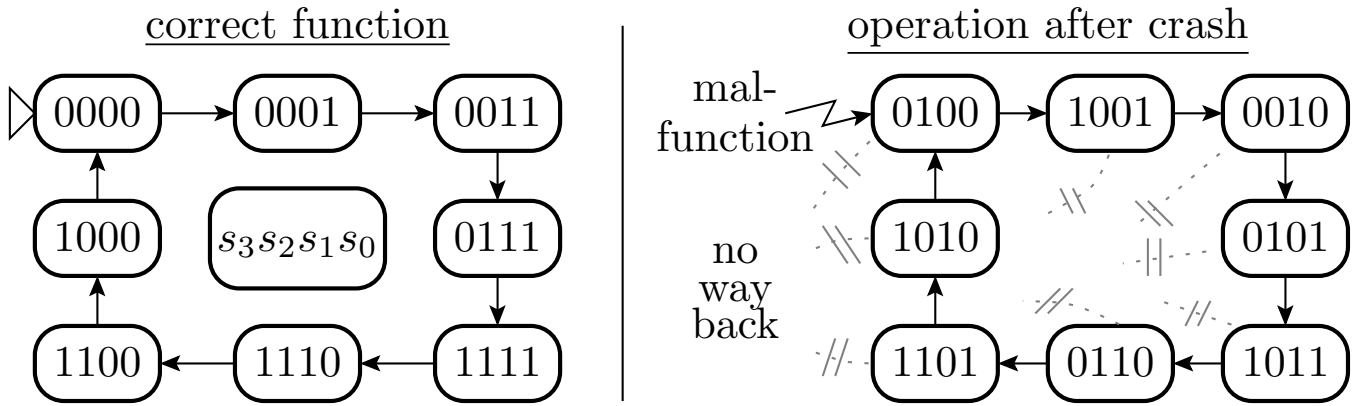
- For some automatons the transfer and the output function can be described much simpler than with case and if statements.

System crash

Illegal states and system crash

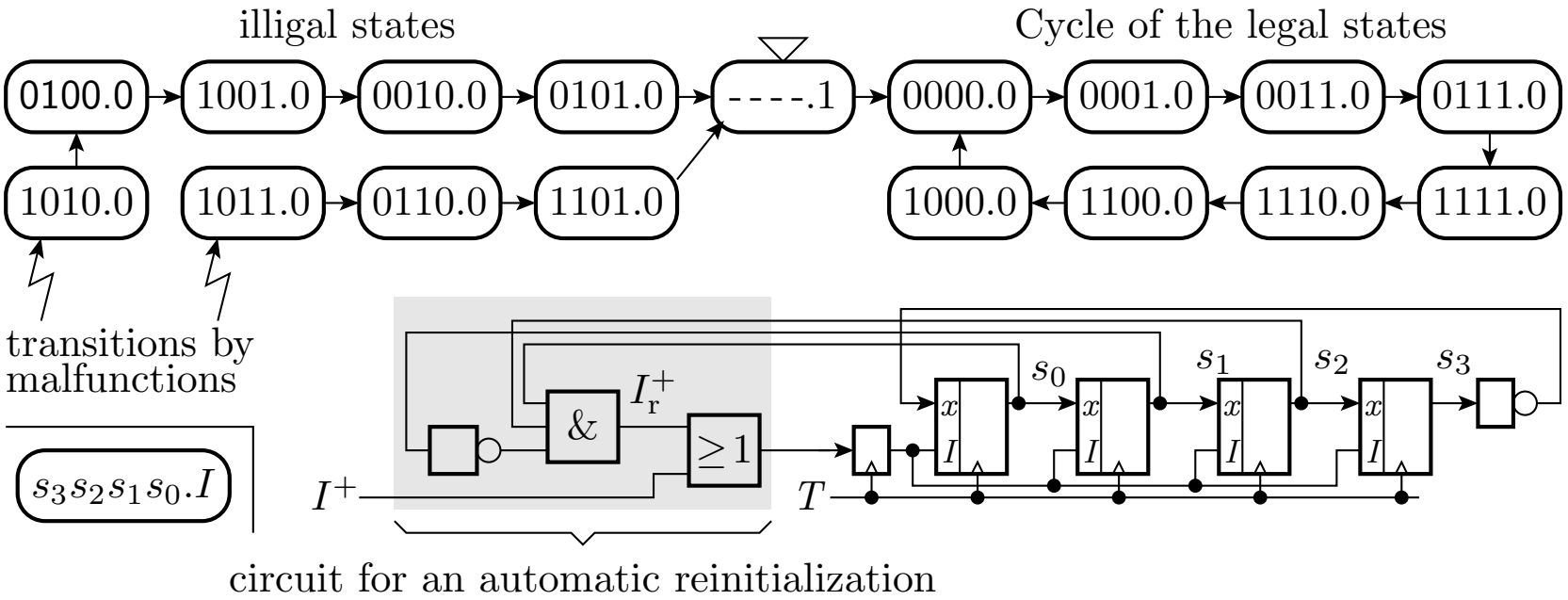
- n memory elements $\Rightarrow 2^n$ states; not all are used
- What happens in the unused (illegal) states?

Example 4 Bit Johnson counter:



- Automaton cycles illegal states
- till reinitialization no reasonable behavior$\Rightarrow$ crash

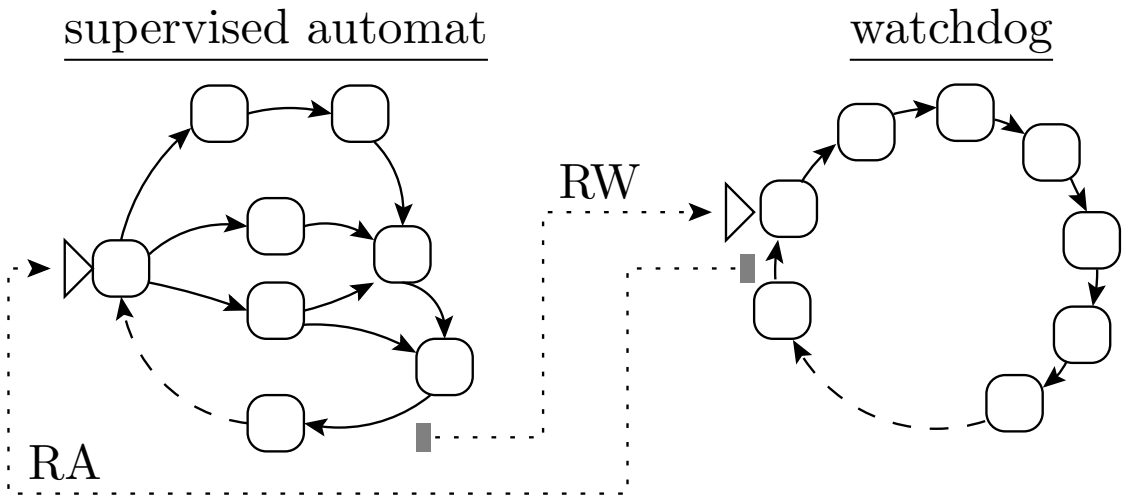
Automated crash recovery



circuit for an automatic reinitialization

- adding the illegal states; defining edges to leave them
- In the example a reset is performed automatically, if the Johnson counter reaches the state »1101« or »0101«.

Watchdog



RA An overflow of the watchdog timer reinitialized the automaton

RW cyclic in certain progamm points (programm points or transmissions), the automat resets the watchdog counter

- $N = 1000$ memory elements $\Rightarrow 2^{1000}$ states, most unused (illegal); Technique on slide before impractical
- The alternative is time monitoring: if if in a given time interval no state to reset the watchdog is reached, the watchdog reinitialized the system
- standard solution for microprocessors

Combination lock

Control design of a combination lock

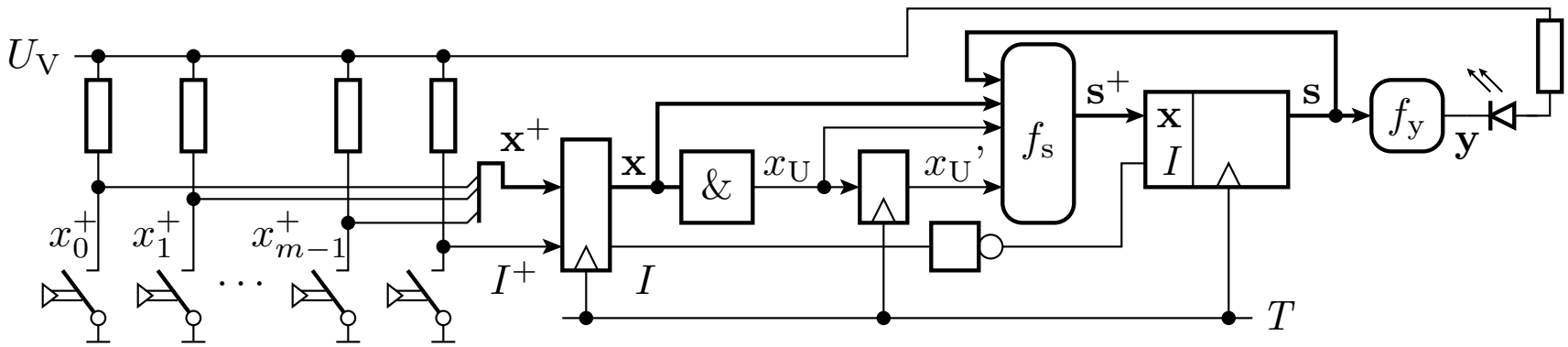
- Input sequence: reset + right number sequence ⇒ output LED turns on
- Input sequence: reset + wrong number sequence ⇒ output LED stays off

---

Design flow:

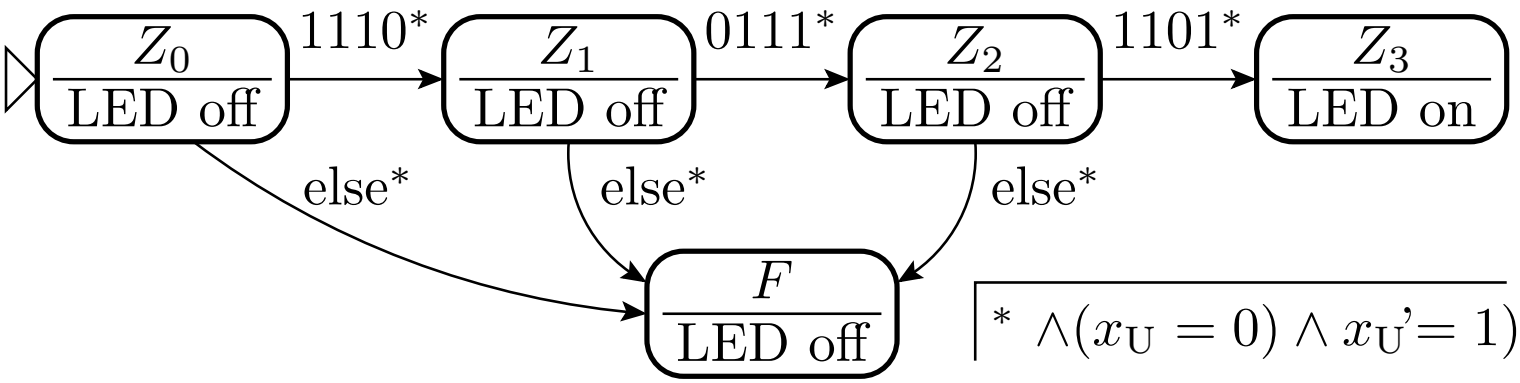
- Circuit sketch, input elements, sampling register, clock circuitry; transfer and output function as circuit blocks still to design
- Specification of the state graph
- State encoding (if not given input and output encoding)
- Description in VHDL
- Simulation
- Synthesis, ...

Circuit sketch



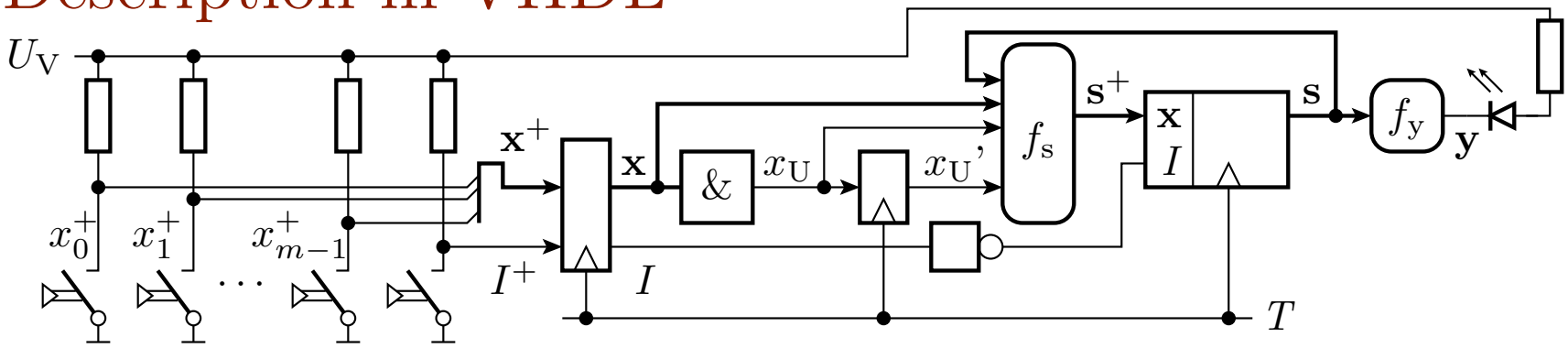
- m number buttons + reset key; asynchronous de-bounced; sampling e.g. with $T_P \approx 10\,\text{ms}$; »0« if pressed
- valid input: sequence no key pressed - single key pressed; multiple key; multiple keys pressed should be handled in the same way as wrong key pressed
- Output LED + series resistor; on when $y = 0$
- Moore automaton (output assigned to the states)
- Reinitialization with the sampled reset value »0«
- x_U – AND instruction of the key signals; if no button is pressed during the sampling »1«, else »0«

Designing the state flow



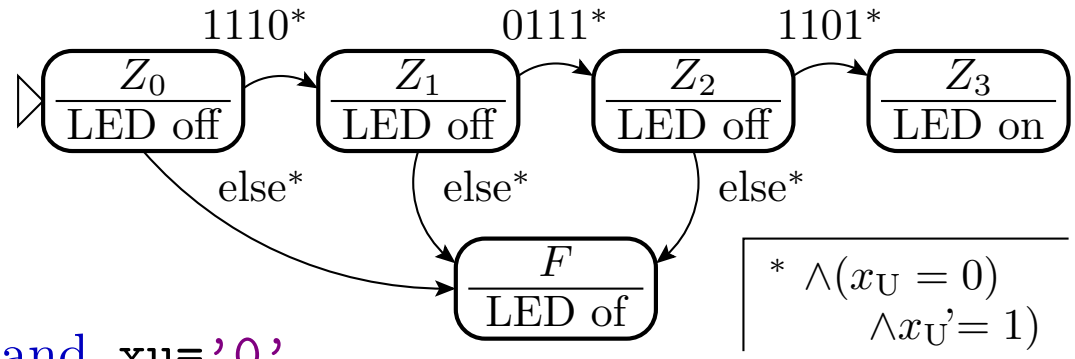
- Acceptor automaton; Z_i name of the state; i – number of the next secret key
- transition condition: active clock edge $\wedge\ x_U = 0$ (key pressed) $\wedge\ x_U$' (step before no key pressed) $\wedge s \neq F \wedge s \neq Z_3$ (not final state)
- correct number sequence 0-3-1 $\Rightarrow$ input vector sequence 1110-0111-1101 ($m = 4$ number keys)
- wrong input $\Rightarrow$ error state F
- The final states F and Z_3 are left only by reinitialization.

Description in VHDL




```
signal T, I_next, I, xu, xu_del: STD_LOGIC;
signal s: STD_LOGIC_VECTOR(2 downto 0);
signal x_next, x: STD_LOGIC_VECTOR(3 downto 0);
...
process(T) begin
 if RISING_EDGE(T) then
   x <= x_next;
   I <= I_next;
   xu_del <= xu;
 end if;
end process;
xu <= x(0) and x(1) and x(2) and x(3);
```

```
process(I, T)
 variable v: STD_LOGIC_VECTOR(6 downto 0);
begin
 if I='0' then
   s <= "000";
 elsif RISING_EDGE(T) and xu='0'
    and xu_del ='1' and s(2)='0' then
   v:= s & x_del;
   case v is
     when "000" & "1110" => s<="001";
     when "001" & "0111" => s<="010";
     when "010" & "1101" => s<="100";
     when others => s <= "111";
   end case;
 end if;
end process;
--- concurrent signal asignment to the output y
y <= not s(2) or s(1) or s(0);
```



state	coding
Z_0	000
Z_1	001
Z_2	010
Z_3	100
F	111

final states: 1--

Summery

Circuit design using finite state machines:

- sketch of the circuit with the Automaton as a Black-Box (creative)
- specify operation by a state graph (creative)
- translate description to VHDL (prescription like)

Operation graph

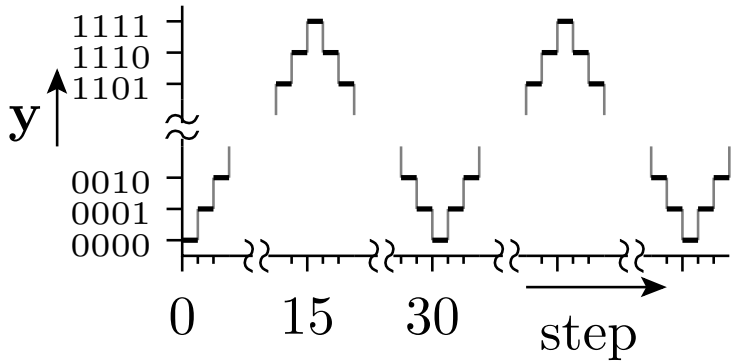
Operation graph

The number of states and edges in an state graph grow exponentially with number of input and state bits.

- How information can be structures better?
- As in a computer program:
  - also a operation graph, which controls data operation
  - limited range of operation: add, count, logical bit operations ...
- Operation graph: extended state graph, that in addition controls Operation and checks operational results for conditional state transfer.
  - Definition of operands and operations to describe the target function as a register transfer function
  - Describing the operation flow as graph.

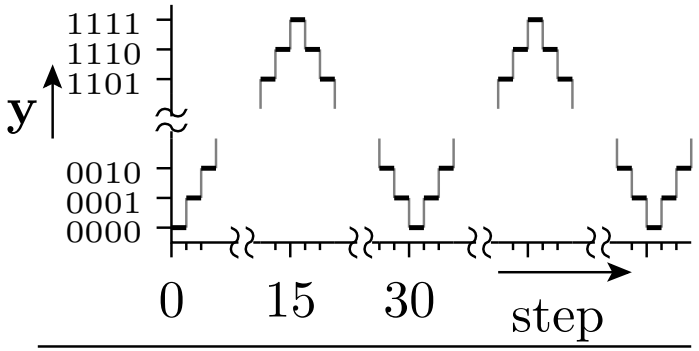
Even a single count operation may simplify the operation flow dramatically.

Triangle signal generator

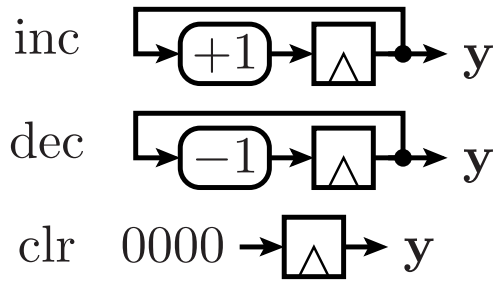


- autonomous automaton, which periodically cycles the 30 states
- state space separable in counting direction (up, down) and count value (»0000« to »1111«)
- the two unused state tuple (»0000«, down) and (»1111«, up) are illegal
- required operations: clr (initialize with »0000), inc (count up) and dec (count down)

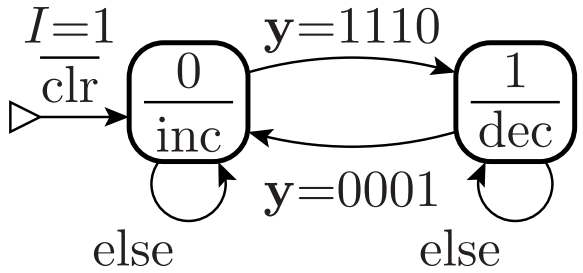
```
library Tuc;
use Tuc.Numeric_Sim.all;
...
signal T, I, s: STD_LOGIC;
signal y: tUnsigned(3 downto 0);
...
process(I, T)
begin
 if I='1' then
  s   <=  '0'; y <= "0000";
 elsif RISING_EDGE(T) then
  case s  is
   when '0' => y <= y +"1";
     if y="1110" then s<='1'; end if;
   when '1' => y <= y -"1";
     if y="0001" then s<='0'; end if;
   when others => y <= "XXXX";
  end case;
 end if;
end process;
```



operations of the count register

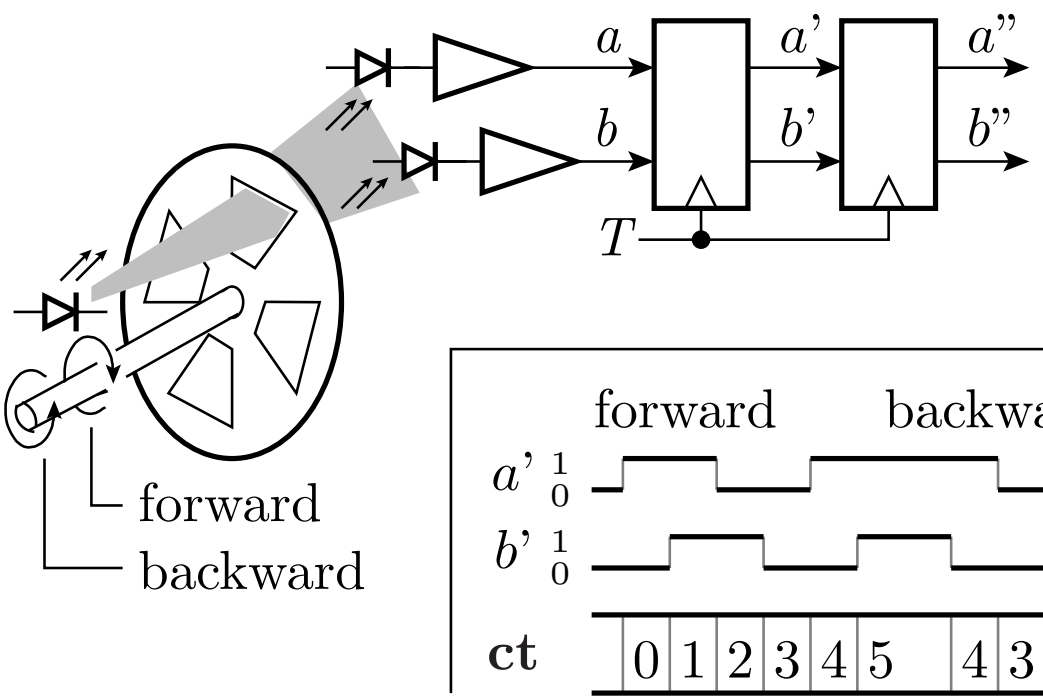


operation graph



Quadrature encoder

Displacement measuring with quadrature encoder



a"	b"	a'	b'	ct
0	0	0	0	—
0	0	0	1	−1
0	0	1	0	+1
0	1	0	1	—
0	1	0	0	+1
0	1	1	1	−1
1	0	1	0	—
1	0	1	1	+1
1	0	0	0	−1
1	1	1	1	—
1	1	1	0	−1
1	1	0	1	+1

- perforated disk at a wheel axle e.g. of a mobile robot
- during turning ahead the signal a and during during turning back the signal b changes first

- stop / don't count: $a"b" = a'b'$
- illegal: $a"b" = \bar{a}'\bar{b}'$

rotation forward / count up

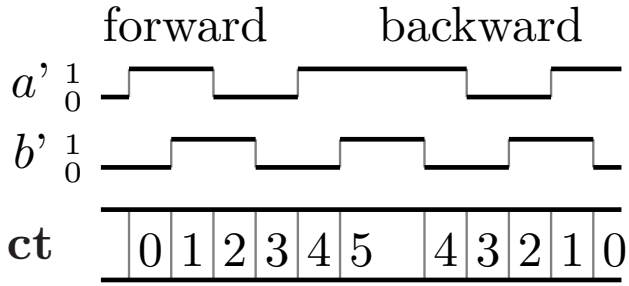
- first a turns on $(a"b"a'b' = 0010)$
- then b turns on $(a"b"a'b' = 1011)$
- then a turns off $(a"b"a'b' = 1101)$
- at last b turns off $(a"b"a'b' = 0100)$.

rotation backward / count down

- first b turns on$(a"b"a'b' = 0001)$
- then a turns on$(a"b"a'b' = 0111)$
- then b turns off$(a"b"a'b' = 1110)$
- at last a turns off$(a"b"a'b' = 1000)$

transfer function:

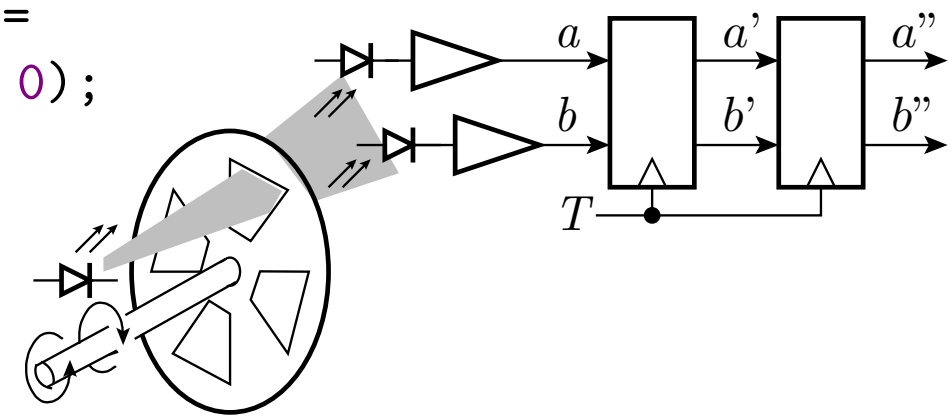
$$a"b" <= a'b'$$



$a"$ $b"$	a' b'	**ct**
0 0	0 0	—
0 0	0 1	−1
0 0	1 0	+1
0 1	0 1	—
0 1	0 0	+1
0 1	1 1	−1
1 0	1 0	—
1 0	1 1	+1
1 0	0 0	−1
1 1	1 1	—
1 1	1 0	−1
1 1	0 1	+1

```
library Tuc; use Tuc.Numeric_Sim.all;
...
signal T, I, I_del: STD_LOGIC;
signal ab: STD_LOGIC_VECTOR(1 downto 0);
signal ab_del: STD_LOGIC_VECTOR(3 downto 0);
signal ct: tSigned(15 downto 0);

--- Abtastprozess ohne Initialisierung
process(T)
begin
 if RISING_EDGE(T) then
   I_del <= I;
   ab_del(1 downto 0) <= ab;
   ab_del(3 downto 2) <=
         ab_del(1 downto 0);
 end if;
end process;
```



a"	b"	a'	b'	**ct**
0	0	0	0	—
0	0	0	1	−1
0	0	1	0	+1
0	1	0	1	—
0	1	0	0	+1
0	1	1	1	−1
1	0	1	0	—
1	0	1	1	+1
1	0	0	0	−1
1	1	1	1	—
1	1	1	0	−1
1	1	0	1	+1

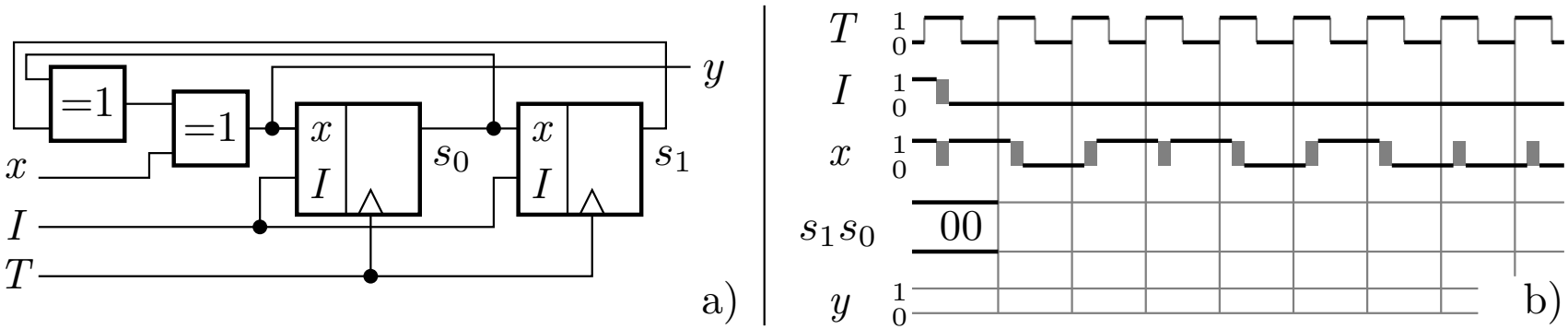
```
--- Zählerprozess
process(I_del, T)
begin
 if I_del='1' then
  ct <= "0000000000000000";
 elsif RISING_EDGE(T) then
  case ab_del  is
   when "0010" |"1011" |"1101" |"0100" => ct <= ct +"1";
   when "0001" |"0111" |"1110" |"1000" => ct <= ct -"1";
   when "0000" |"0101" |"1010" |"1111" => null;
   when others => ct <="X...(16×X)...XX";
  end case;
 end if;
end process;
```

Summary

- functional specification as state graph; recipe like implementation in VHDL and further in a circuit
  - state ⇒ state register, sampling process
  - initial value ⇒ initial value of the state register
  - transfer and output function ⇒ sample or combinatorial process with case distinction on case and input
- illegal states; danger to crash; typical error handling reinitialization; reset button, watchdog
- complex functions with large input and state ranges
  - describing target function by an operation sequence
  - specification of operands and operations as register transfer functions
  - operation graph
  - recipe like mapping to VHDL

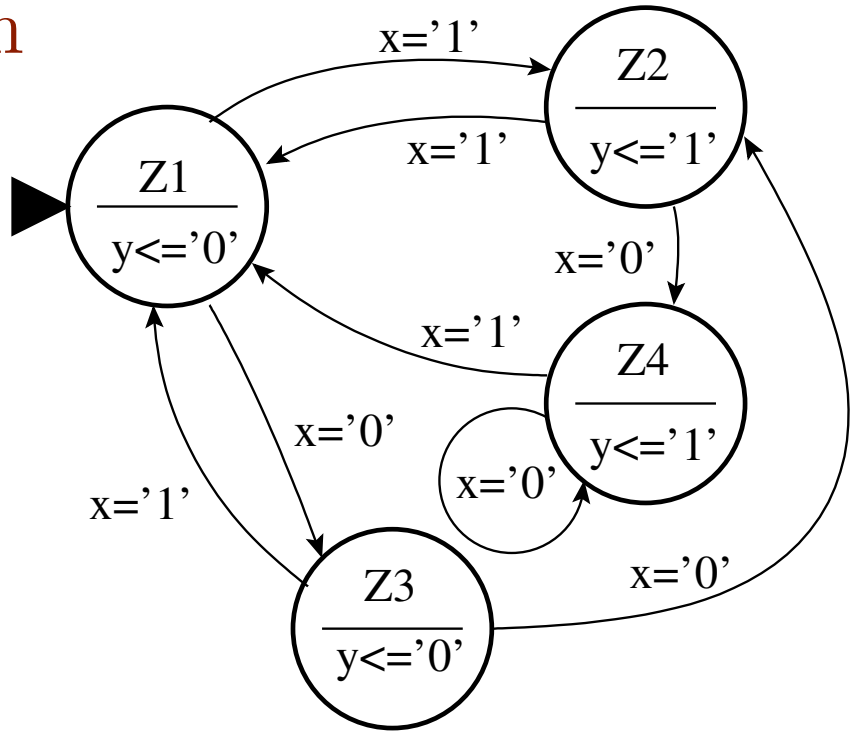
Exercises

Aufgabe 1.18: Feedback shift register



- Determine the next states $s_1^+ s_0^+$ and the output value y for each variation of the actual state $s_1 s_0$ and the input value of x
- draw the state graph
- Add the signal flow for $s_1 s_0$ and y in the figure right.

Aufgabe 1.19: Automaton



symbolic state name	Z1	Z2	Z3	Z4
assigned bit vector	"00"	"01"	"10"	"11"

- state transition with the rising clock edge
- initialization with I='1'

1. Fill in the state transition table

input	0	1	0	1	0	1	0	1
state	00	00	01	01	10	10	11	11
next state								
output								

2. Drawing of the complete circuit (input sampling, state register, transfer function, output function)
3. How many memory elements are required?
4. Description in VHDL (entity + architecture).

Aufgabe 1.20: Monitor for a transmitter signals

For a transmitter signal it has to be monitored, that the difference Δ between the number of transmitted ones minus the transmitted zeros does not exceed the range of -3 to 3. In case of violation error signal y should become active. Example wave form:

x	1	0	0	0	0	0	1	0	1	1	1	1	1	1	1	...
Δ	0	1	0	-1	-2	-3	-3	-2	-3	-2	-1	0	1	2	3	3
y	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	...

(x – input; Δ – number of »0« minus number of »1«; y – error signal)

To do:

- Draw state graph

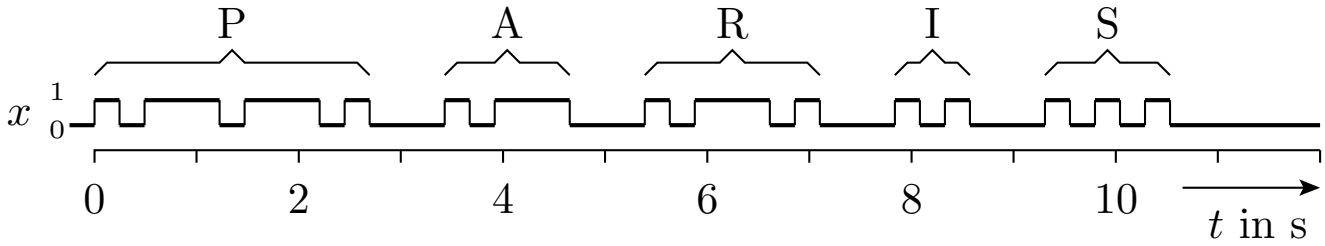
Aufgabe 1.21: Clock divider

Design a clock divider with division factor $2 \cdot n$, which with each n-th active input clock edge events the output clock. To do:

- Draw the signal flow of the input and the output clock for $n = 2$.
- Describe the target function withe an operation graph and a counter.
- VHDL description. Parameter n should be declared as a constant.

Hint: The data type of the counter signal may also be a number type (INTEGER, NATURAL or POSITIVE).

Aufgabe 1.22: Morse receiver



Morse signals consist of

- short pulses (dot, $t_P = 200...300\,\text{ms}$) and
- long pulses (dash, $t_S = 600...900\,\text{ms}$).

- clock frequency: $f_T = 20\,\text{Hz}$
- bouncing time less than clock period.
- two low active input button
  - signal x: Morse_signal
  - signal I: Initialization_signal

- Three output bits, to be activated for one clock
  - signal p: after receiving a dot
  - signal s: after receiving a dash
  - signal err: after receiving a pulse of unallowed length
- Draw complete circuit with buttons, sample registers, state registers and the transfer and the output function as black box .
- Operation graph using a counter
- VHDL description (only declaration of the date objects and the processes)