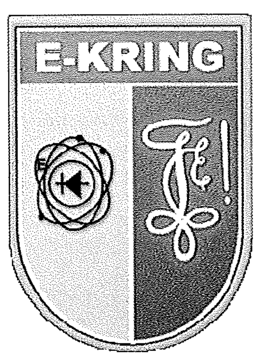


Academiejaar 2009 - 2010

Digitale Elektronica (gemeenschappelijk)

Tweede jaar Ea-ICT / Schakeljaar

J. Meel

Met dank aan

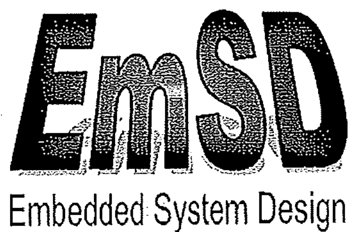
HOGESCHOOL VOOR WETENSCHAP & KUNST | **DE NAYER INSTITUUT**
SINT-KATELIJNE-WAVER

Digitale Elektronica

Bachelor in de Industriële Wetenschappen: elektronica-ICT
2e jaar, 4e semester

2E / SPE

Ba2.EL.02
SPBa2.EL.02



ir. J. Meel

feb 2009

FINITE STATE MACHINE

1. ANALYSE

1.1 DFF , TFF , JF FF

1.2 BINAIRE COUNTER

2. KLASSIFIKATIE

2.1 MEALY

2.2 MOORE

3. SYNTHESE

3.1 ALGEMENE ONTWERPFLOW

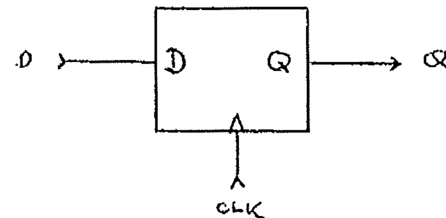
3.2 POSITIEVE FLANKDETEKTOR (MOORE)

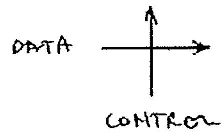
3.3 POSITIEVE FLANKDETEKTOR (MEALY)

1. ANALYSE

1.1 D FF

SCHEMA





TIMING

ν: 0 | 1 | 2 | 3 | 4

CLK

D — T_{SETUP}, T_{HOLD}, D^2, D^3, D^4

Q — T_{CO}, Q^2, Q^3, Q^4

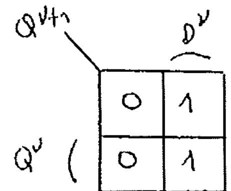
STATE TIME

SAMPLE $D = D^3$

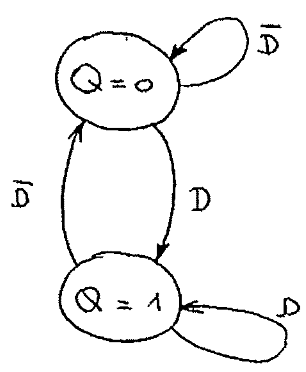
STATE TRANSITION $Q^3 \rightarrow Q^4$

TRANSITIE TABEL

D^ν	Q^ν	$Q^{\nu+1}$
0	0	0
0	1	0
1	0	1
1	1	1
PRESENT INPUT	PRESENT STATE	NEXT STATE



TOESTANDS DIAGRAM



○ = n x state time

→ = state transition : CLK ↑

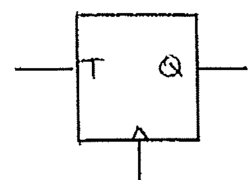
NEXT STATE VERGELIJKING

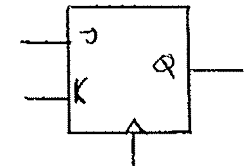
$$Q^{\nu+1} = D^\nu$$

T- FF

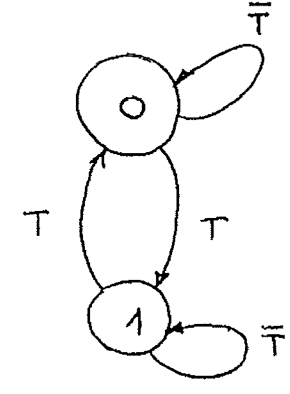
JK - FF

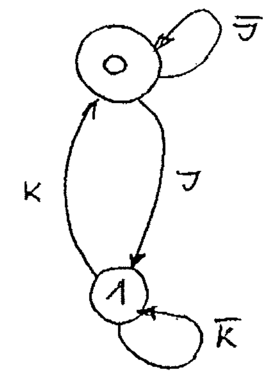
CHEMA





OESTANDS-
IAGRAM





RANSITIETABEL

T^{ν}	Q^{ν}	$Q^{\nu+1}$
0	0	0
0	1	1
1	0	1
1	1	0

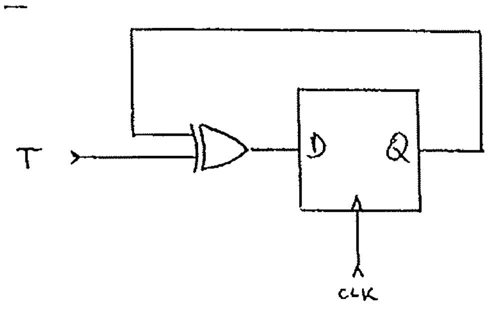
J^{ν}	K^{ν}	Q^{ν}	$Q^{\nu+1}$
0	X	0	0
1	X	0	1
X	0	1	1
X	1	1	0

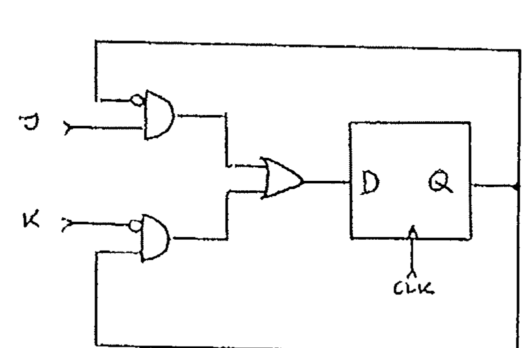
NEXT STATE VGL

$$Q^{\nu+1} = T^{\nu} \oplus Q^{\nu}$$

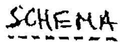
$$Q^{\nu+1} = J^{\nu}\bar{Q}^{\nu} + \bar{K}^{\nu}Q^{\nu}$$

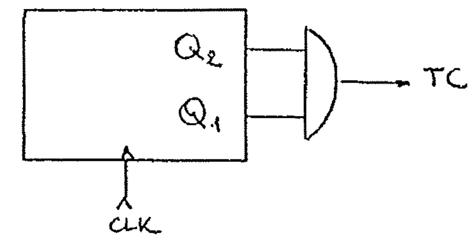
SYNTHESE met DFF



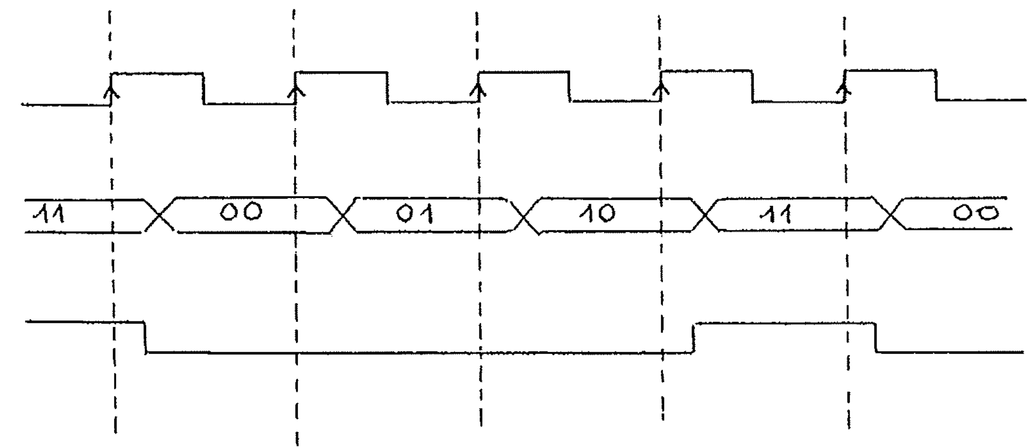


1.2 BINAIRE TELLER

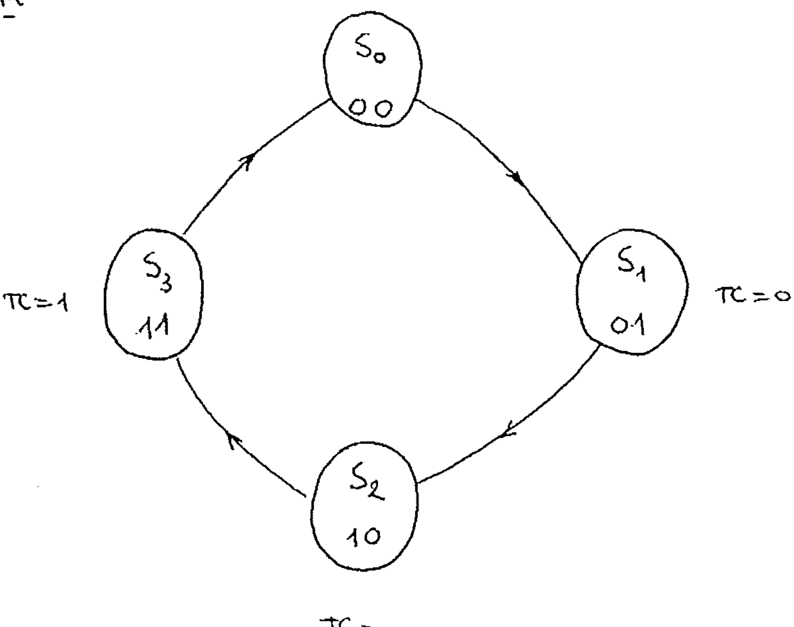




TIMING



TOESTANDS DIAGRAM



KONVENTIE

STATE OUTPUT LIST

STATE NAME

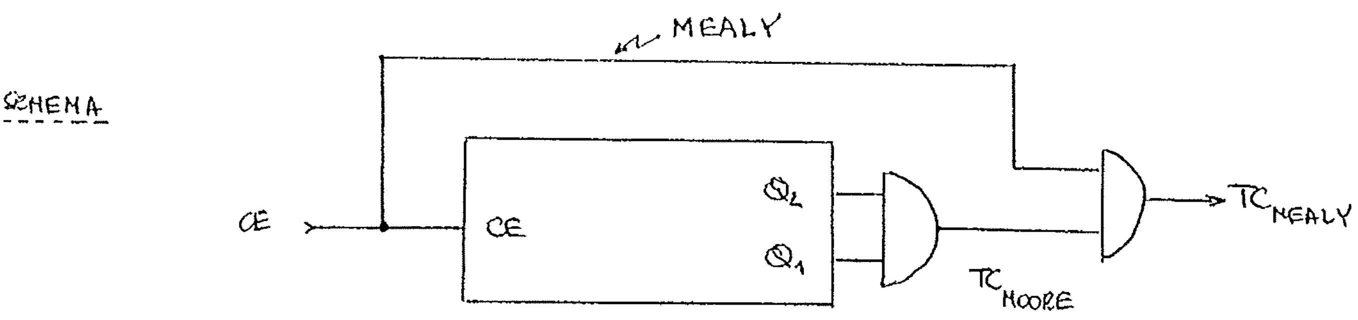
STATE CODE

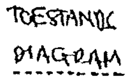
STATE TRANSITION CONDITION

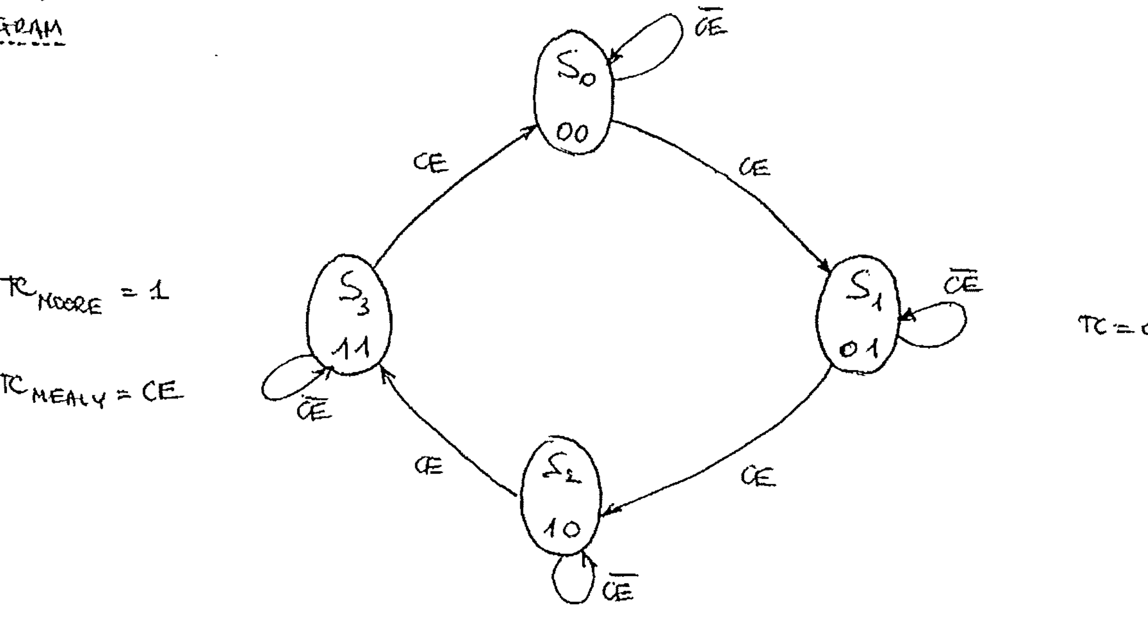
NEXT STATE VGL

$$(\text{STATE CODE})^{\nu+1} = (\text{STATE CODE})^{\nu} + 1$$

SCHEMA







TRANSITIE TABEL

CE	Q^{ν}	$Q^{\nu+1}$	TC_{MOORE}	TC_{MEALY}
0 1	S_0	S_0 S_1	0	0
0 1	S_1	S_1 S_2	0	0
0 1	S_2	S_2 S_3	0	0
0 1	S_3	S_3 S_0	1	0 1

NEXT STATE VGL

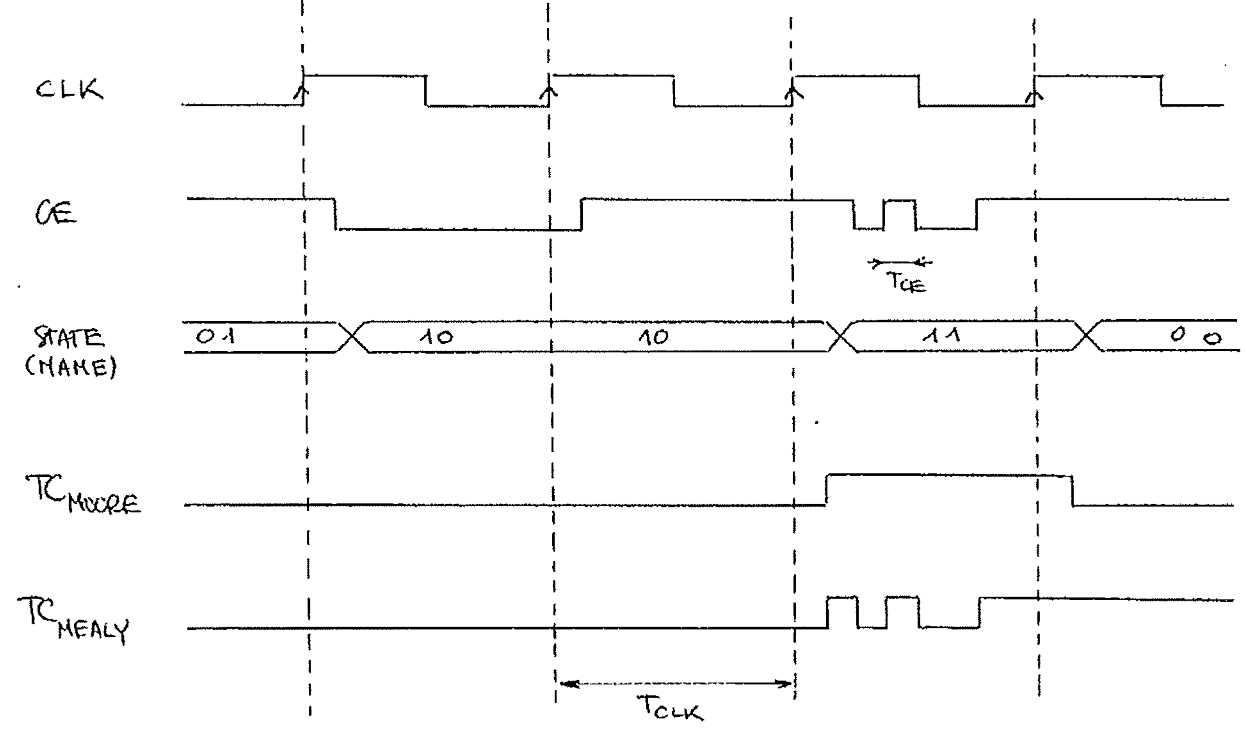
$$Q^{\nu+1} = Q^{\nu} + CE^{\nu} \longrightarrow \boxed{Q^{\nu+1} = f_{q}(Q^{\nu}, I^{\nu})}$$

OUTPUT VGL

$$TC^{\nu}_{MOORE} = Q^{\nu}_{2} . Q^{\nu}_{1} \longrightarrow \boxed{O^{\nu} = f_{o}(Q^{\nu})}$$

$$TC_{MEALY} = Q^{\nu}_{2} . Q^{\nu}_{1} . CE^{\nu} \longrightarrow \boxed{O^{\nu} = f_{o}(Q^{\nu}, I^{\nu})}$$

TIMING



MINIMUM OUTPUT PULSWIDTH : MOORE $T_p = n \cdot T_{CLK}$

MEALY $T_p = \min(T_{CLK}, T_{input})$

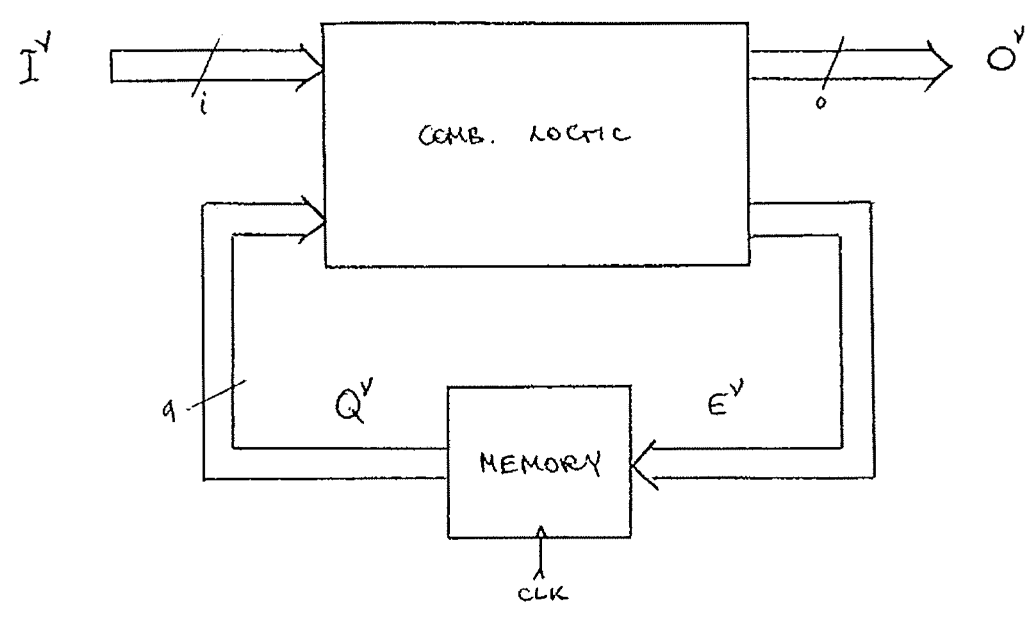
MOORE : $O^v = f_o(Q^v)$ → uitgang wijzigt enkel bij toestandstransities

MEALY : $O^v = f_o(Q^v, I^v)$ → uitgang wijzigt bij toestandstransities én ingangsvariaties.

2. KLASSIFIKATIE

ALGEMENE VOORSTELLING VAN EEN FSM (FINITE STATE MACHINE)

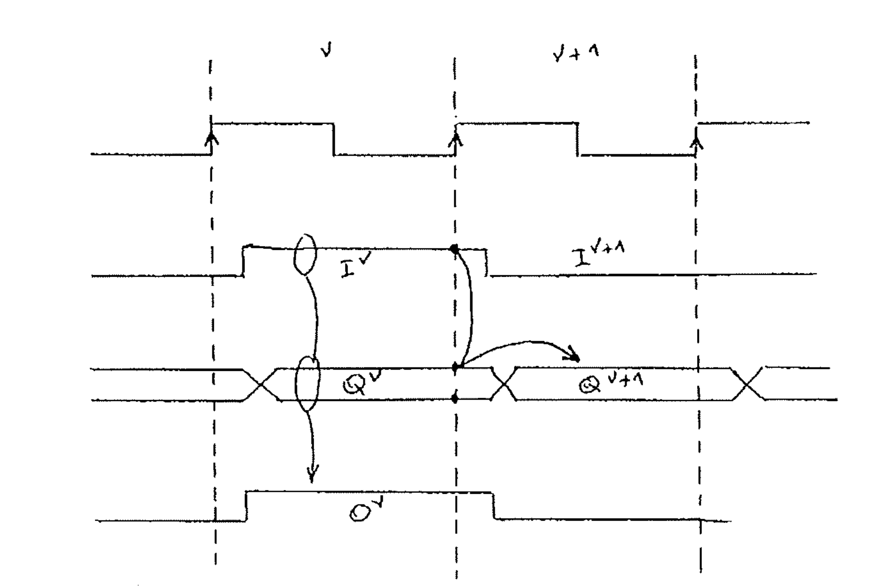
SCHEMA



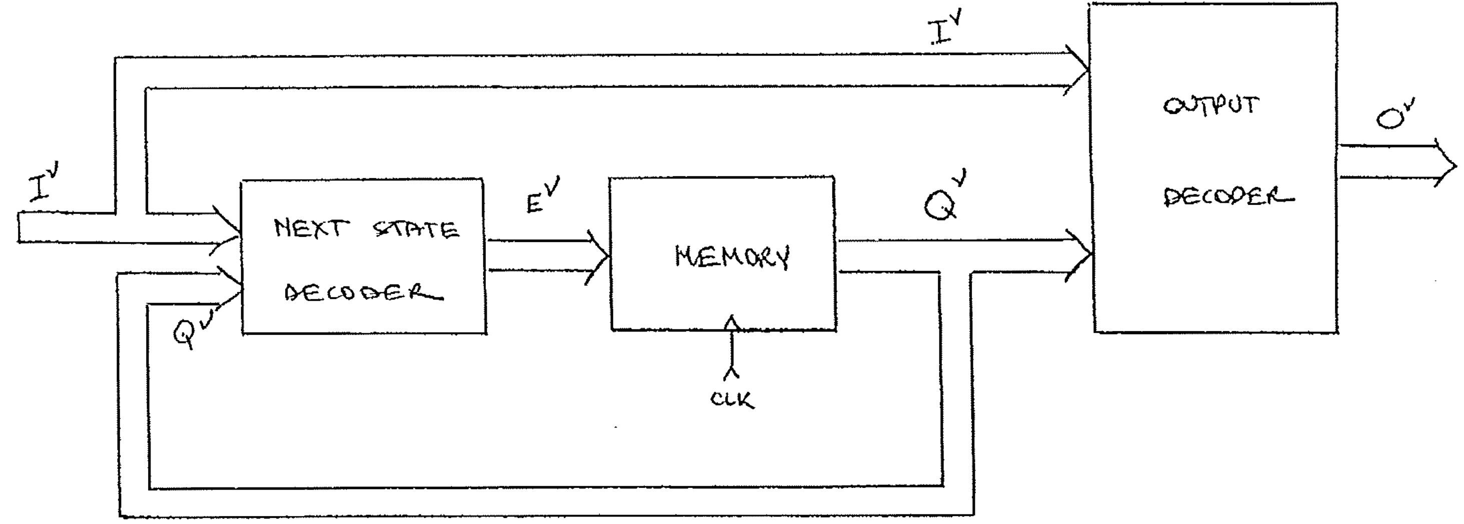
VGL

next state vgl :	$Q^{v+1} = f_q(Q^v, I^v)$	q vgl	
excitatie vgl :	$E^v = f_e(Q^v, I^v)$	q vgl 2q vgl	(D ff, T ff) (SR ff, JK ff)
output vgl :	$O^v = f_o(Q^v, I^v)$	o vgl	

TIMING



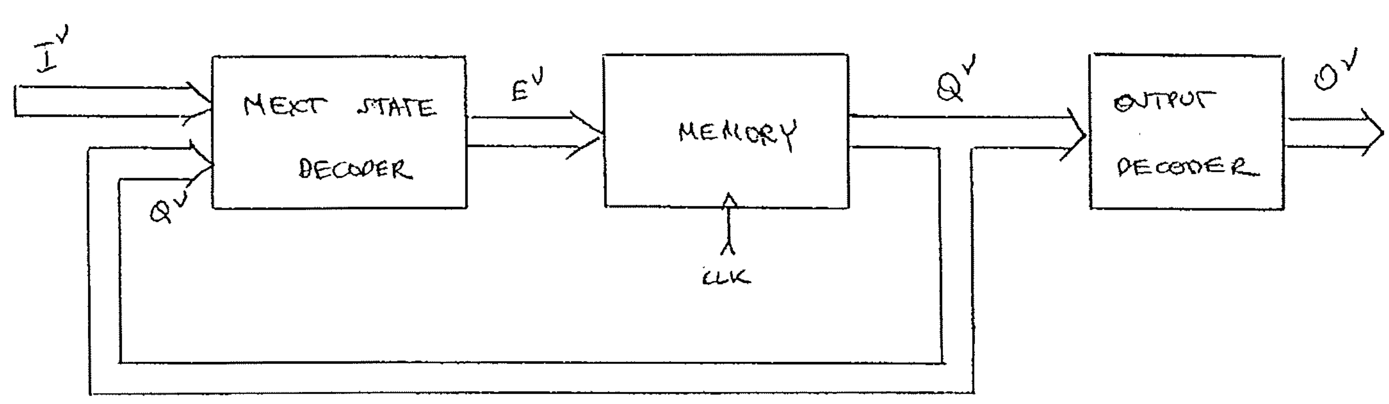
2.1 MEALY



$E^V = f_e(Q^V, I^V)$: next state decoder

$O^V = f_o(Q^V, I^V)$: output decoder

2.2 MOORE



$E^V = f_e(Q^V, I^V)$: next state decoder

$O^V = f_o(Q^V)$: output decoder

3. SYNTHESE

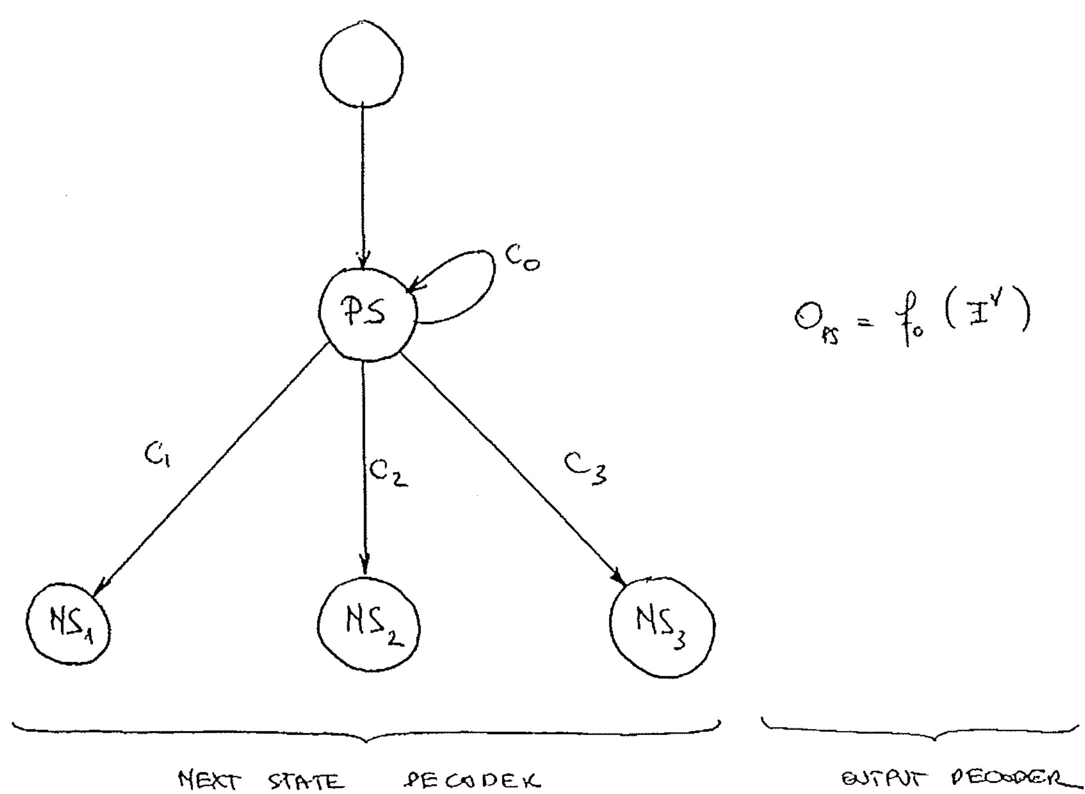
3.1 ALGEMENE ONTWERP FLOW

① PROBLEEMFORMULERING = SPECIFIKATIE

De specifikatie van een synchrone sekwentiële schakeling (FSM) is vaak een woordelijke gedragsbeschrijving. Deze kan worden aangevuld met een schema (inputs / outputs) en een timingdiagram.

② TOESTANDSDIAGRAM = FORMELE SPECIFIKATIE

De woordelijke gedragsbeschrijving moet worden omgezet in een éénduidige, formele specifikatie: het toestandsdiagram (eventueel kan gebruik worden gemaakt van een Hardware Description Language: VHDL, ABEL,..).



PS = PRESENT STATE NS$_i$ = NEXT STATE

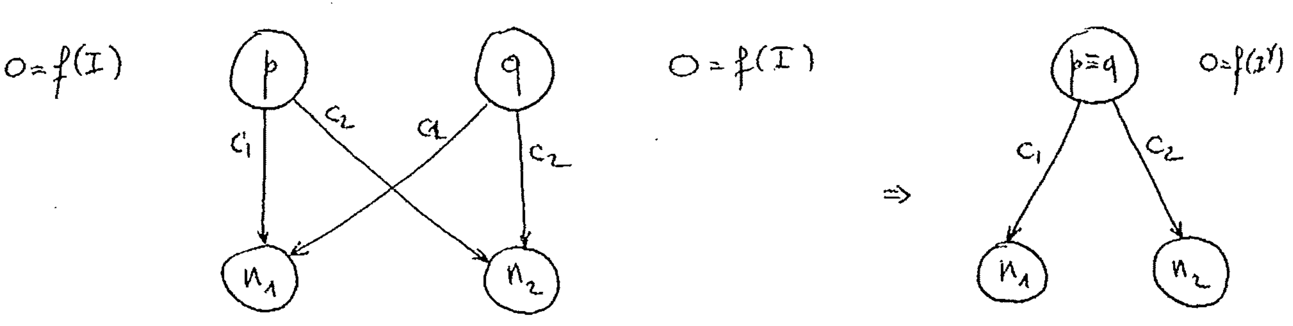
<u>VW</u>: TRANSITIE CONDITIES c_i ZIJN <u>EXCLUSIEF</u>! (slechts <u>1</u> next state)

$$(c_0\bar{c}_1\bar{c}_2\bar{c}_3 + \bar{c}_0 c_1\bar{c}_2\bar{c}_3 + \bar{c}_0\bar{c}_1 c_2\bar{c}_3 + \bar{c}_0\bar{c}_1\bar{c}_2 c_3 = 1)$$

③ TOESTANDSREDUKTIE

Toestandsreduktie is het identificeren en kombineren van ekwivalente toestanden.
Toestanden p en q zijn ekwivalent als:

Voor alle ingangskombinaties : zelfde uitgang
zelfde (of ekwivalente) next state



Voordeel : - mogelijk reduktie van het aantal ff's
- mogelijke reduktie van de kombinatorische logika.

④ TOESTANDSTOEKENNING

Het minimum aantal flip-flop's n nodig om een FSM met r toestanden te implementeren is :

$$r \leq 2^n \quad \Rightarrow \quad n \geq \frac{\ln r}{\ln 2}$$

De toestandsvektor Q is verzameling van de toestandswaarden van de n flip-flop's :

$$Q^v = (Q_n^v, Q_{n-1}^v, \dots, Q_1^v)$$

Deze specifieert op elk ogenblik v de toestand van de FSM.
Een toestandskode is een specifieke (vaste) waarde voor de toestandsvektor:

$$\text{toestandskode}_i = ({}^iQ_n, {}^iQ_{n-1}, \dots {}^iQ_1) \overset{vb}{=} (0, 1, \dots, 1)$$

Toestandstoekenning is de toekenning van een 'unieke' toestandscode aan elke symbolische toestand (PS, NS_1, NS_2, ..) in het toestandsdiagram.

PS = toestandscode i

NS_1 = toestandscode i+1.

⋮

De toestandstoekenning bepaalt de konkrete excitatiegl en outputgl die moeten worden geïmplementeerd en bepaalt dus het aantal poorten nodig voor de realisatie van de FSM. Voor een FSM met n flip-flop's zijn er echter n! mogelijke toekenningen, een grote waarde voor reële toestandsmachines.

Heuristische toekenningsregels (van hoogste naar laagste prioriteit)

a. De globale reset (idle) state krijgt een eenvoudige waarde: 00...0 of 11...1

b. [diagram: toestanden α en β gaan beide met c_1 naar eenzelfde toestand]

Toestanden met eenzelfde next state voor eenzelfde input konditie krijgen een logisch aanliggende toestandstoekenning (= 1 bitpositie verschillend)

c. [diagram: één toestand gaat met c_1 naar α en met c_2 naar β]

Toestanden die de next state zijn van eenzelfde toestand krijgen een logisch aanliggende toestandstoekenning.

d. α $O = f(I^v)$

β $O = f(I^v)$

Toestanden met eenzelfde outputspecifikatie krijgen een logisch aanliggende toestandstoekenning.

Regels b en c reduceren de next state decoder. Regel d reduceert de output decoder. Indien tegenstrijdige vereisten worden bekomen, krijgt de regel met hoogste prioriteit voorrang.

⑤ KEUZE VAN HET TYPE FLIP-FLOP

Het type flip-flop hangt af van de gekozen implementatietechniek (PAL, GAL, EPLD, FPGA, ASIC, ..). Sommigen laten het type flip-flop vrij. De next state vergelijkingen zijn onafhankelijk van het type flip-flop. De exitatievergelijkingen E^v worden wel bepaald door het type flip-flop.

De exitatietabel geeft voor een specifieke flip-flop de nodige kontrole-ingangen om een bepaalde toestandstransitie te genereren.

Q^v	$\rightarrow Q^{v+1}$	D^v	T^v	J^v	K^v
0	0	0	0	0	x
0	1	1	1	1	x
1	0	0	1	x	1
1	1	1	0	x	0

Voor de D-ff zijn de exitatievergelijkingen gelijk aan de next state vergelijkingen. De exitatievergelijkingen voor de T-ff's volgen uit deze voor de D'ff volgens:

$$\begin{cases} Q_i^v = 0 & \rightarrow T_i^v = D_i^v \\ Q_i^v = 1 & \rightarrow T_i^v = \overline{D_i}^v \end{cases}$$

De exitatievergelijkingen voor de JK-ff's volgen uit deze voor de D'ff's volgens:

$$\begin{cases} Q_i^v = 0 & \rightarrow J_i^v = D_i^v \quad K_i^v = x \\ Q_i^v = 1 & \rightarrow J_i^v = x \quad K_i^v = \overline{D_i}^v \end{cases}$$

De D-ff is het meest gebruikte type flip-flop in CMOS technologie (kleinste kost). Door de grote hoeveelheid x's in de vergelijkingen voor de JK-ff, zijn de afzonderlijke J en K exitatievergelijkingen soms eenvoudiger ten koste van meer interkonnekties (Sommige EPLD's laten de keuze van het type flip-flop vrij).

⑥ TRANSITIE TABEL

De transitietabel is een tabulaire voorstelling van het gedrag van de FSM zoals weergegeven in het toestandsdiagram. De toestanden worden voorgesteld met hun state code.

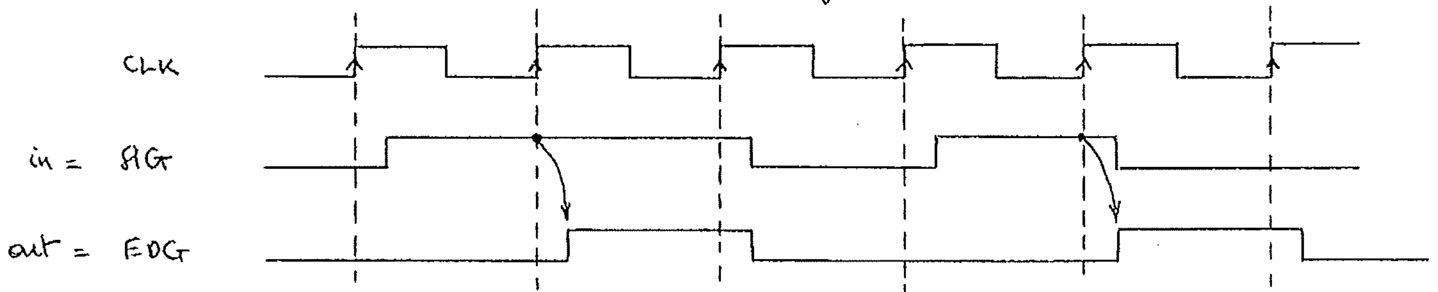
PRESENT INPUTS I^v	PRESENT STATE Q^v	NEXT STATE Q^{v+1}	PRESENT OUTPUT O^v
C_0	PS = 0011	PS = 0011	$f_o(I^v)$
C_1		NS_1 = 0100	
C_2		NS_2 = 0101	
C_3		NS_3 = 0110	

Gebruik makend van de excitatietabel van het gekozen type flip-flop, kan voor elke flip-flop in een waarheidstabel de excitatie worden bepaald om een specifieke toestandstransitie (zoals in de transitietabel aangegeven) te realiseren.

3.2. POSITIEVE FLANKDETECTOR (MOORE)

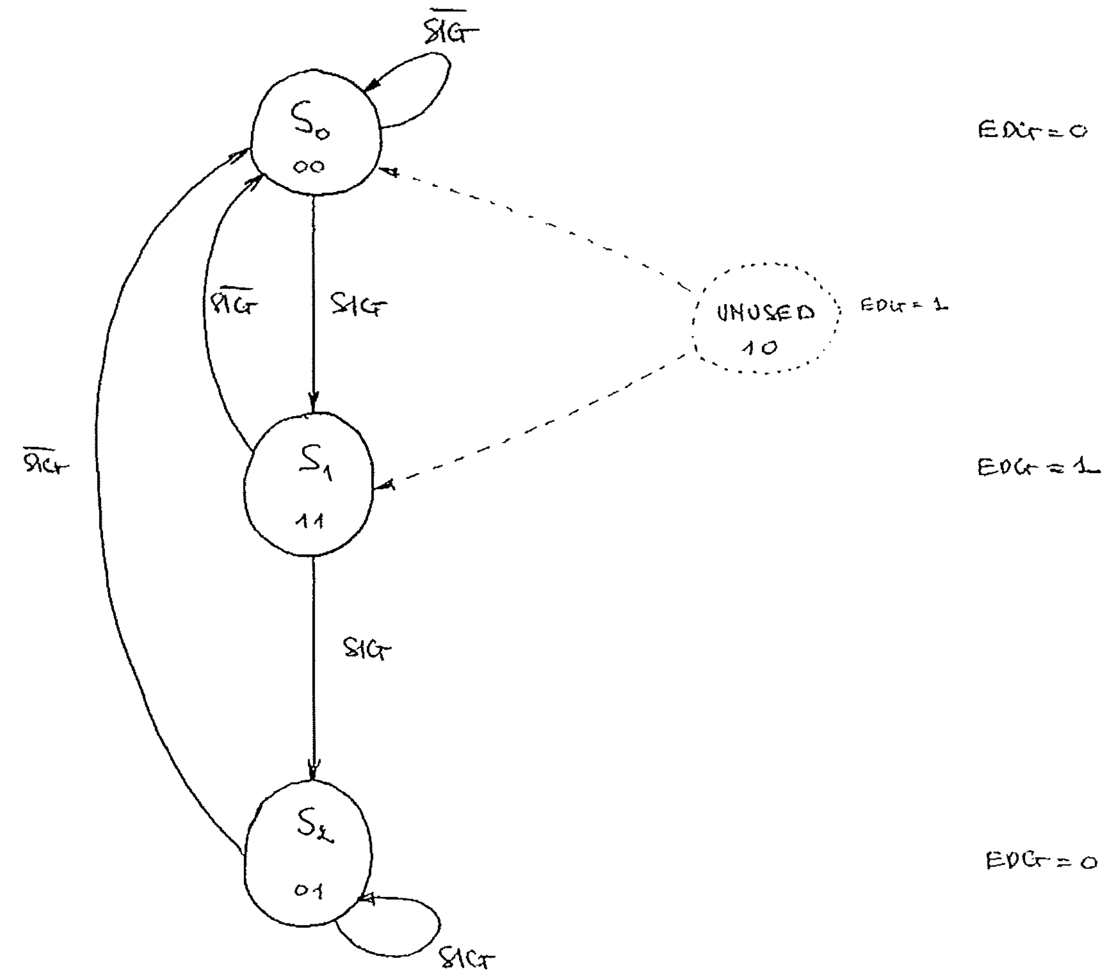
① PROBLEEMFORMULERING

De positieve flankdetektor genereert bij elke positieve flank op een ingangssignaal SIG een puls met een breedte T_{CLK} 'synchroon' met de klok (= MOORE). De schakeling voldoet aan het timingdiagram:



Een minimale realisatie zonder lock-out states wordt voorgesteld.

② TOESTANDSDIAGRAM



Bij de verdere realisatie moet ervoor gezorgd worden dat de UNUSED state voor elke ingangskombinatie een toestandstransitie naar een gebruikte state (S_0, S_1, S_2) heeft.

③ TOESTANDSREDUKTIE

De drie toestanden zijn uniek : S_1 heeft een unieke uitgang.
S_0 en S_2 hebben verschillende nextstates.

④ TOESTANDSTOEKENNING

Voor de realisatie van 3 toestanden zijn 2 flip-flop's nodig ($2^2 = 4$ states)
er is dus 1 ongebruikte toestand (UNUSED) → statevector = (Q_2, Q_1)

Toekenningsregels

a. S_0 is de opstart toestand (reset state) : $S_0 = 00$

b. S_1 en S_2 logisch aanliggend

c. S_0 en S_2 logisch aanliggend

d. S_0 en S_2 logisch aanliggend

Vertaald naar een statemap :

	$\overline{Q_1}$	Q_1
$\overline{Q_2}$	S_0	S_2
Q_2	UNUSED	S_1

$S_0 = 00$

$S_1 = 11$

$S_2 = 01$

UNUSED = 10

⑤ KEUZE VAN HET TYPE FLIP-FLOP

De realisatie met D, T en JK flip-flops worden met elkaar vergeleken.
Uiteraard hangt de keuze samen met de gekozen implementatietechniek
(TTL, PAL, GAL, EPLD, ...).

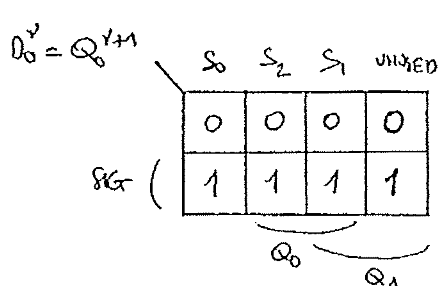
⑥ TRANSITIETABEL

SIG^v	Q_1^v Q_0^v	Q_1^{v+1}	Q_0^{v+1}	EDG^v
0	$S_0 = 0\ 0$	0	0	0
1		1	1	
0	$S_2 = 0\ 1$	0	0	0
1		0	1	
0	$S_1 = 1\ 1$	0	0	1
1		0	1	
0	UNUSED $= 1\ 0$	X (0)	X (0)	X (1)
1		(1)	(1)	(1)

Bij de verdere implementatie moet nagegaan worden dat de x's de UNUSED state niet 'lock-out' maken.

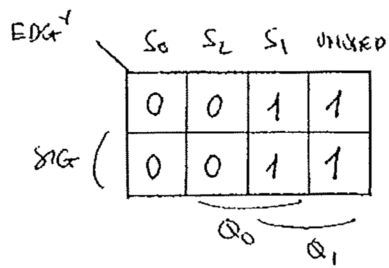
⑦ D-FF REALISATIE

Next-state map en excitatiemap zijn identiek ($Q^{v+1} = D^v$).



$D_1^v = Q_1^{v+1}$

	S_0	S_2	S_1	UNUSED
	0	0	0	0
SIG	1	0	0	1

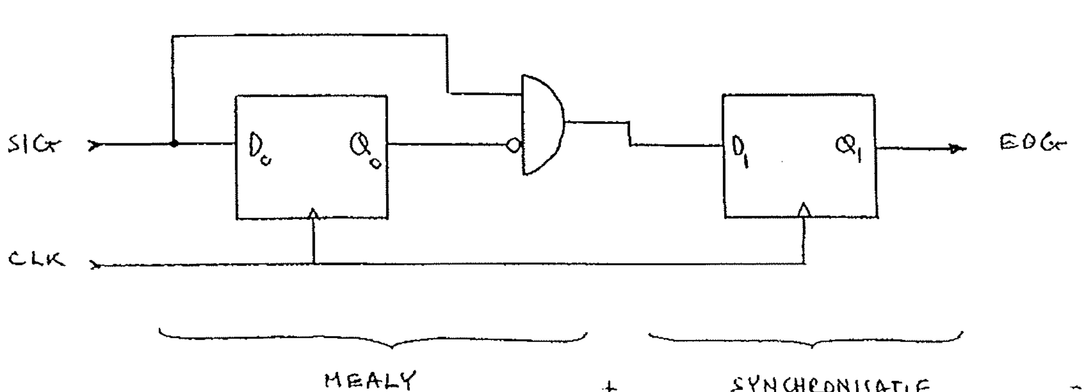


$$Q_0^{v+1} = D_0^v = SIG^v$$
$$Q_1^{v+1} = D_1^v = SIG \cdot \overline{Q}_0^v$$

} next state vgl (lock-out = ok)

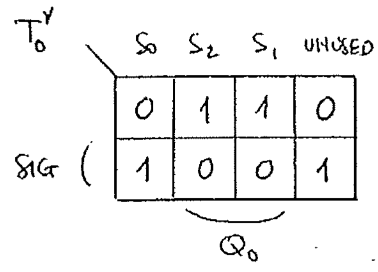
$$EDG^v = Q_1^v$$

output vgl



(8) T-FF REALISATIE

Enkel de exitatievergelijkingen wijzigen, niet de outputvergelijkingen.



T_0^v

	S_0	S_2	S_1	UNUSED
	0	1	1	0
SIG	1	0	0	1

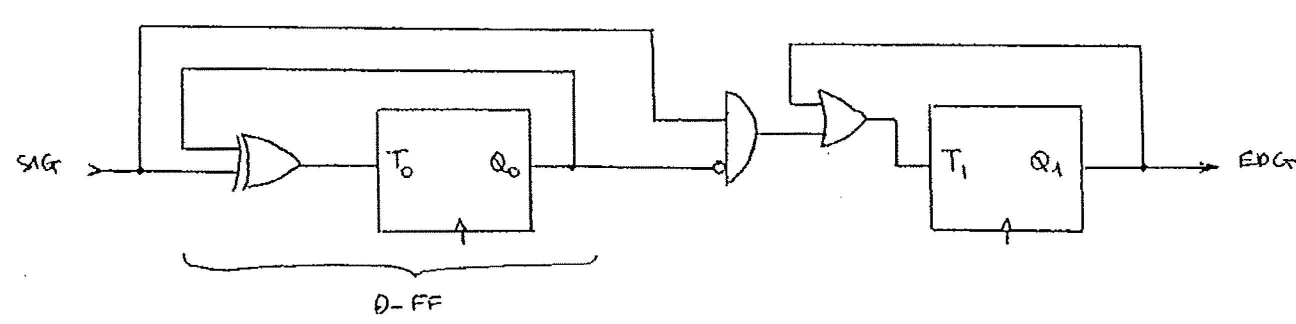
(Q_0 marks the S_2 and S_1 columns.)

T_1^v

	S_0	S_2	S_1	UNUSED
	0	0	1	1
SIG	1	0	1	1

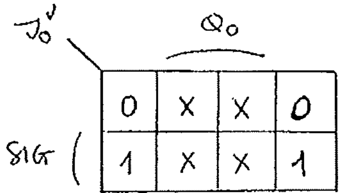
(Q_1 marks the S_1 and UNUSED columns.)

$$T_0^v = Q_0 \oplus SIG \qquad T_1^v = Q_1 + \overline{Q_0}\, SIG$$



lockout preventie : $\overline{SIG}$ = UNUSED → S_0 , SIG = UNUSED → S_2

(9) JK-FF REALISATIE



J_0^v (Q_0 marks the middle two columns)

	0	X	X	0
SIG	1	X	X	1

$J_0^v = SIG^v$

K_0^v (Q_0 marks the middle two columns)

	X	1	1	X
SIG	X	0	0	X

$K_0^v = \overline{SIG}^v$

J_1^v (Q_1 marks the last two columns)

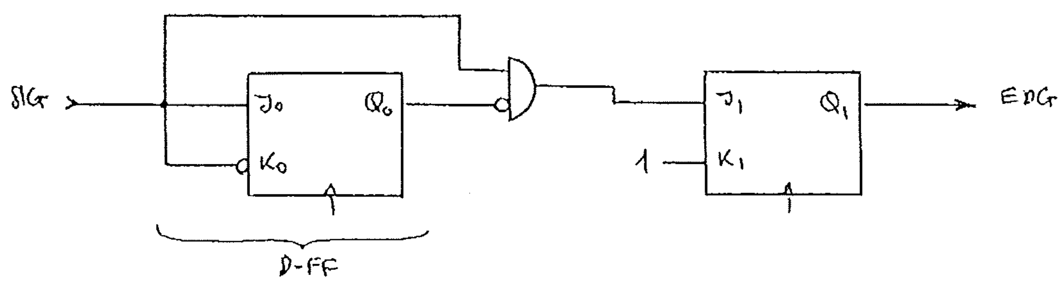
	0	0	X	X
SIG	1	0	X	X

$J_1^v = SIG^v \cdot \overline{Q_0}'$

K_1^v (Q_1 marks the last two columns)

	X	X	1	1
SIG	X	X	1	1

$K_1^v = 1$



lock-out preventie : $\overline{SIG}$ = UNUSED → S_0 , SIG = UNUSED → S_2

3.3. POSITIEVE FLANKDETEKTOR (MEALY)

① PROBLEEM FORMULERING

De positieve flankdetektor genereert bij elke positieve flank op het ingangssignaal SIG 'onmiddellijk' een puls EDG met een maximale breedte T_{CLK} (klokperiode). De schakeling voldoet aan het timingdiagram:

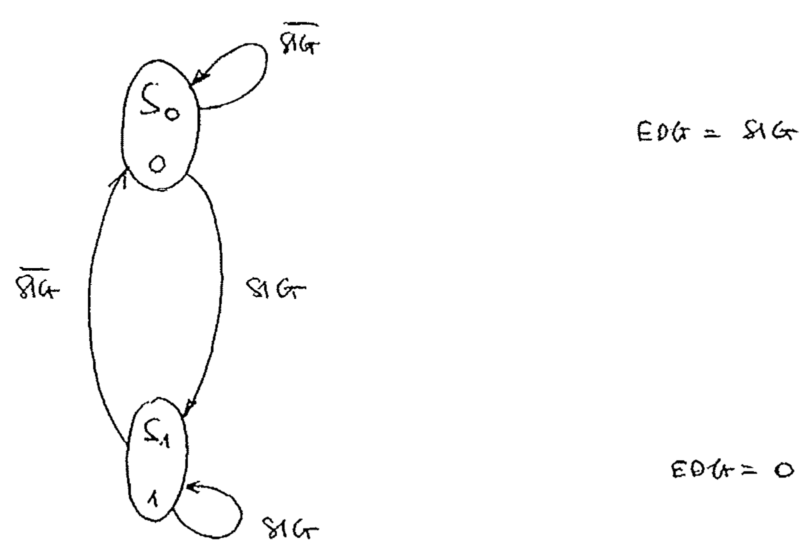
CLK

in = SIG

out = EDG

(In tegenstelling met de MOORE automaat, kan de uitgang van de MEALY automaat ogenblikkelijk reageren op een ingangsvariatie: Een minimum pulsbreedte van T_{CLK} is niet langer gegarandeerd!")

② TOESTANDSDIAGRAM



S_0 = wacht op 1

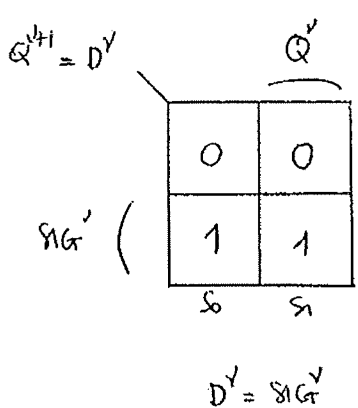
S_1 = wacht op 0

De EDG specifikatie is nu ook afhankelijk van de ingangskonditie => MEALY

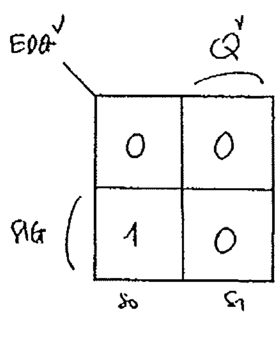
③ TRANSITIETABEL

SIG^{ν}	Q^{ν}	$Q^{\nu+1}$	EDG^{ν}
0 1	$S_0 = 0$	0 1	0 1
0 1	$S_1 = 1$	0 1	0

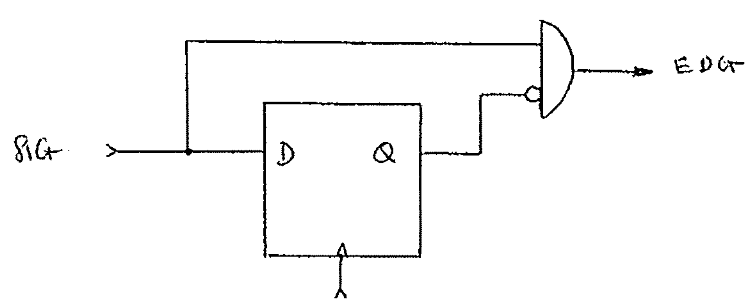
④ DFF REALISATIE



$D^{\nu} = SIG^{\nu}$

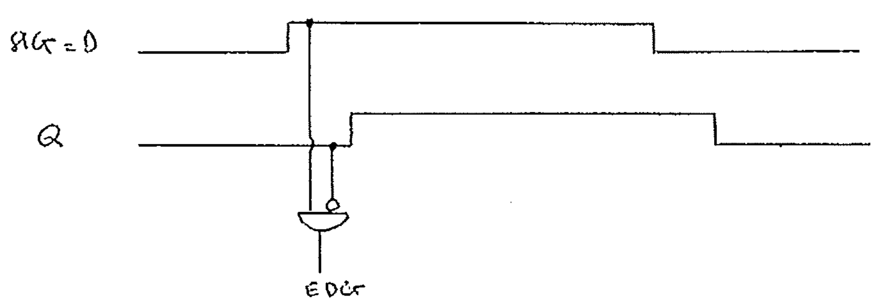


$EDG^{\nu} = SIG^{\nu} . \overline{Q}^{\nu}$



WERKINGSPRINCIPE

De inkomende golfvorm wordt vergeleken met een vertraagde versie



HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

Digitale Elektronica
Finite State Machines

ir. J. Meel
may 2008

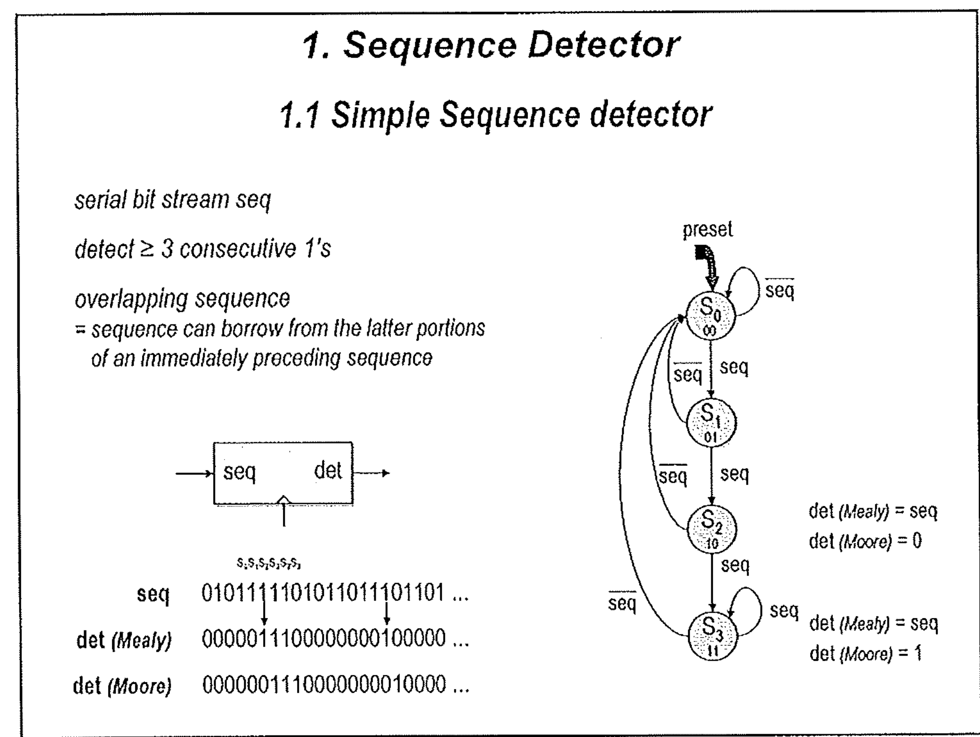
1. Sequence Detector

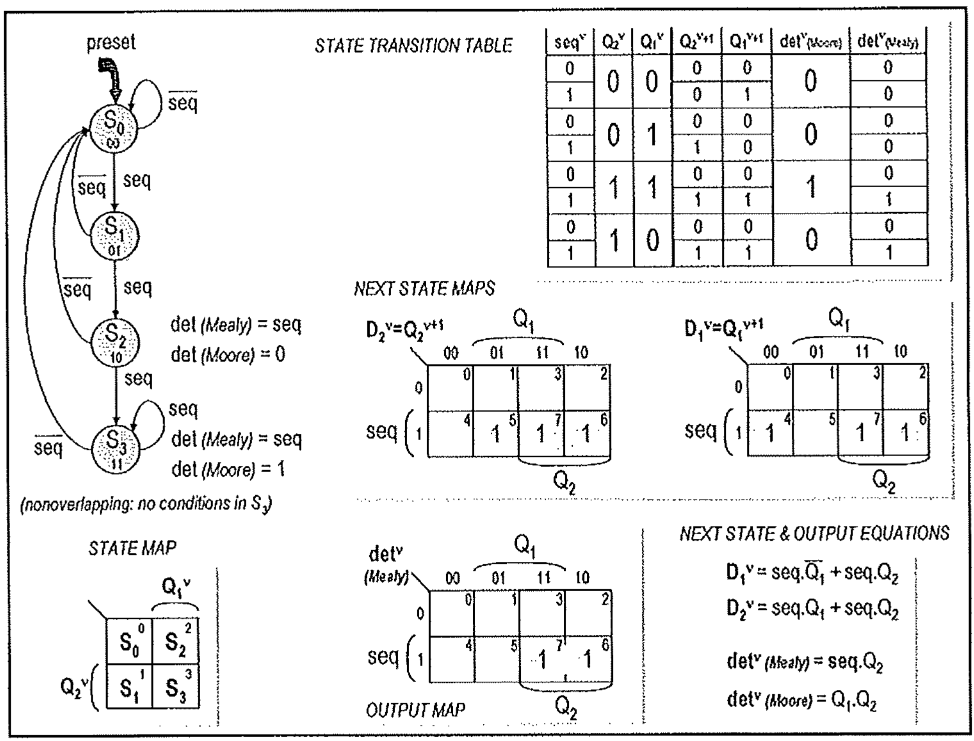
1.1 Simple Sequence detector

serial bit stream seq

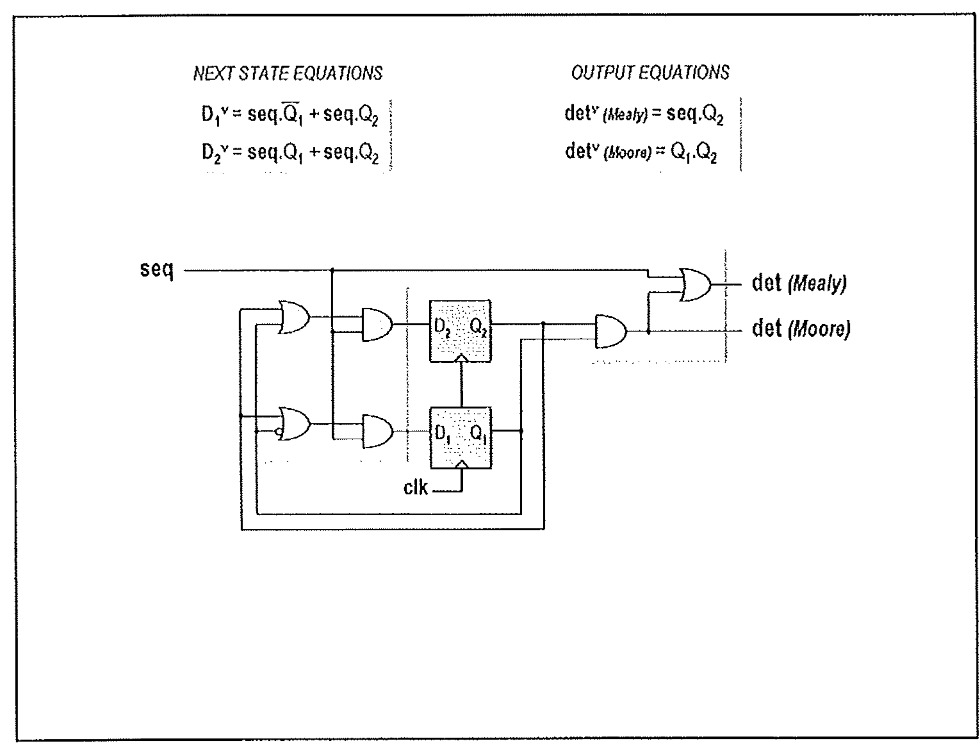
detect ≥ 3 consecutive 1's

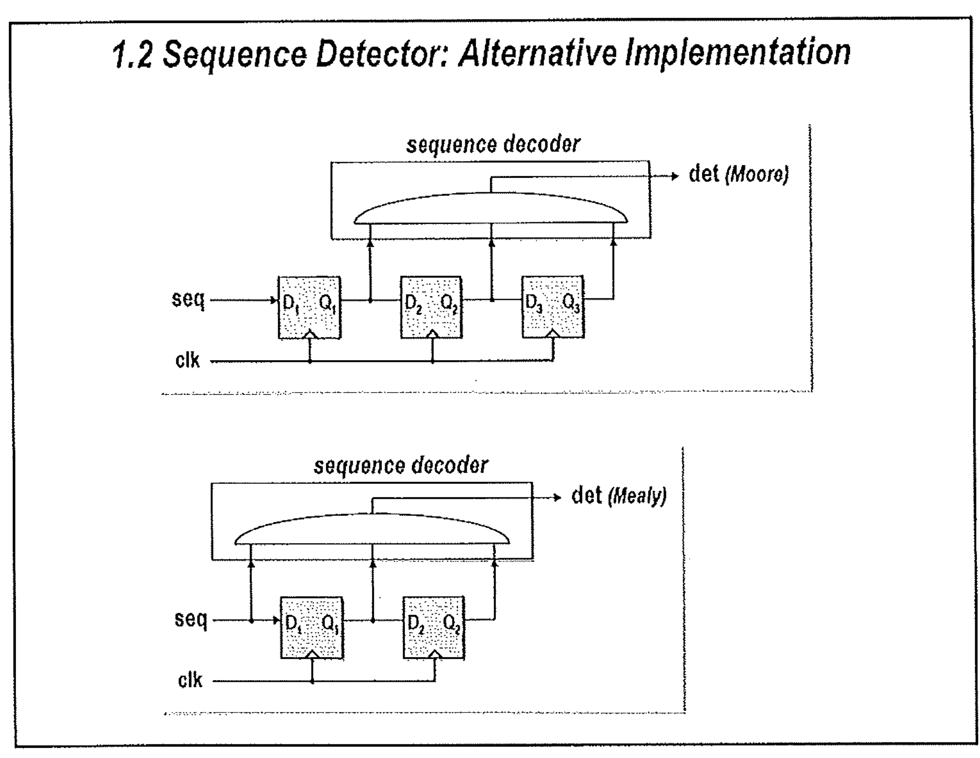
overlapping sequence
= sequence can borrow from the latter portions of an immediately preceding sequence

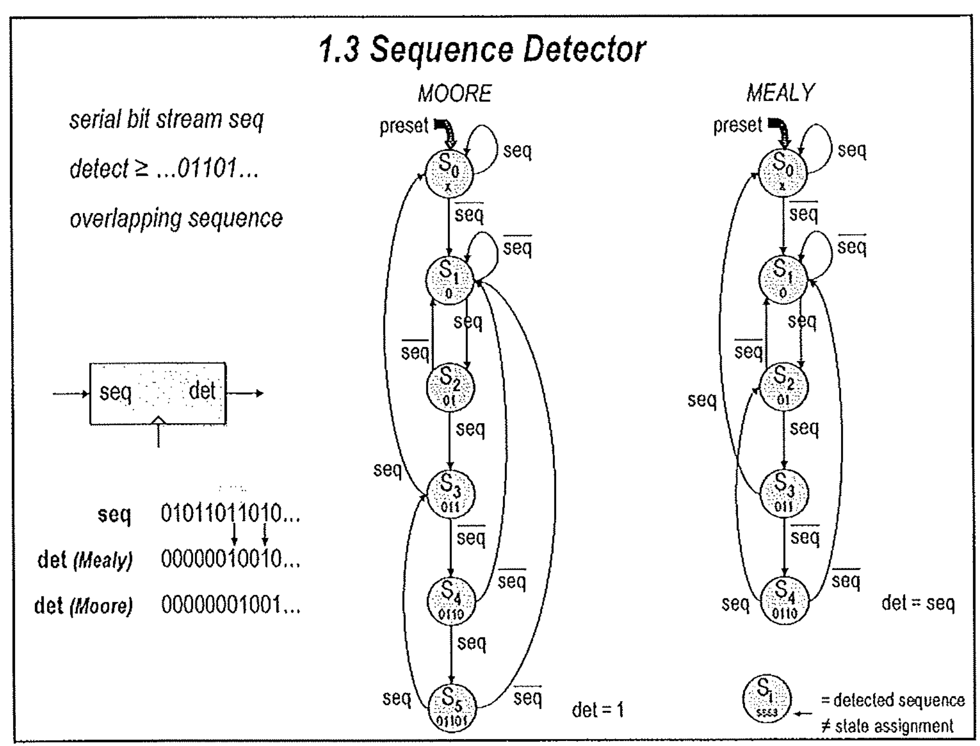


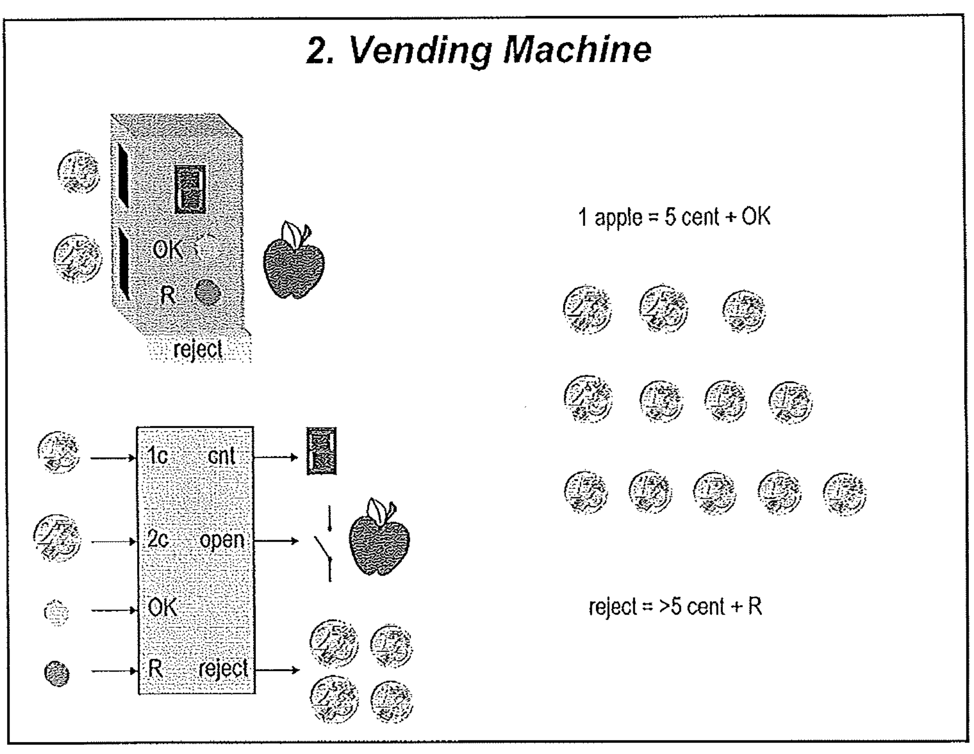


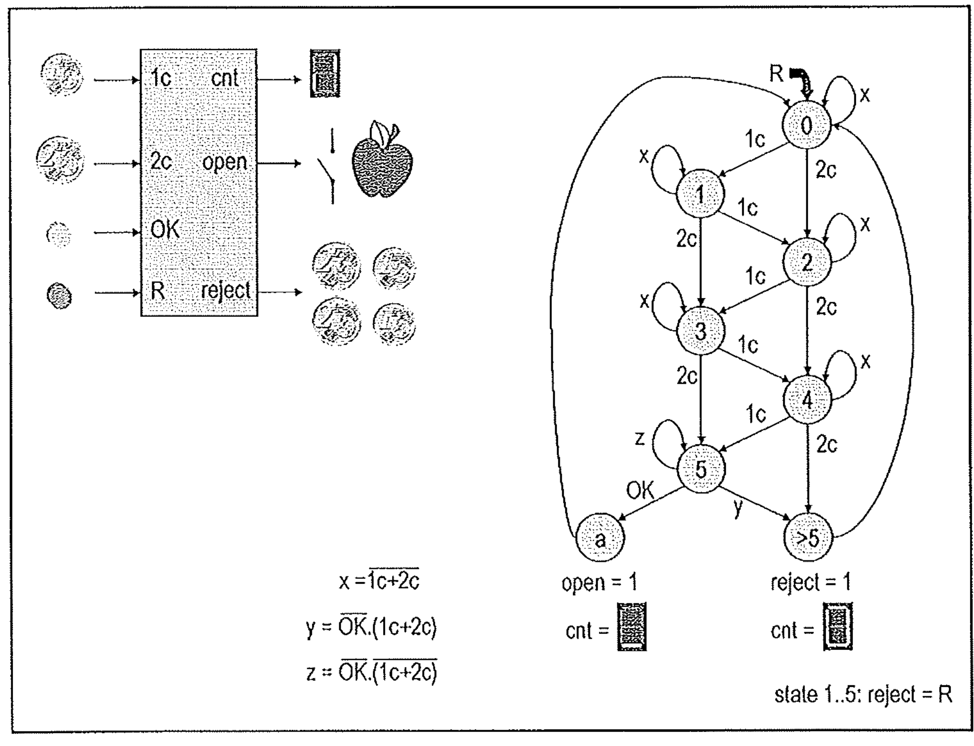
seq^v	Q_2^v	Q_1^v	Q_2^{v+1}	Q_1^{v+1}	det^v(Moore)	det^v(Mealy)
0	0	0	0	0	0	0
1			0	1		0
0	0	1	0	0	0	0
1			1	0		0
0	1	1	0	0	1	0
1			1	1		1
0	1	0	0	0	0	0
1			1	1		1



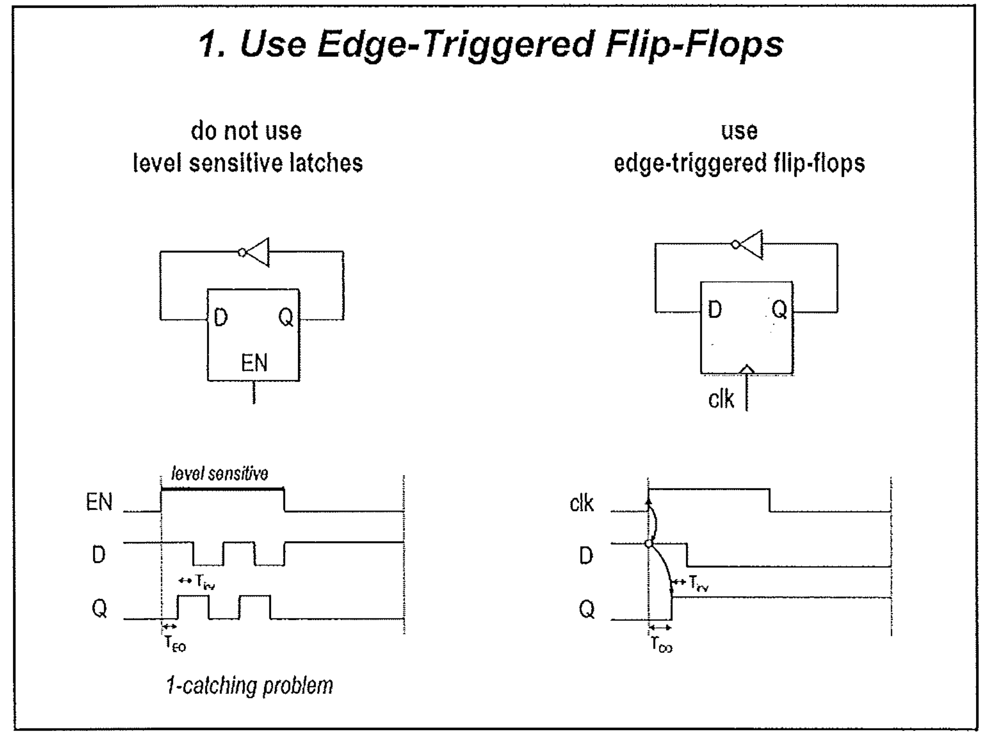

1.2 Sequence Detector: Alternative Implementation
sequence decoder
det (Moore)
seq
clk
sequence decoder
det (Mealy)
seq
clk

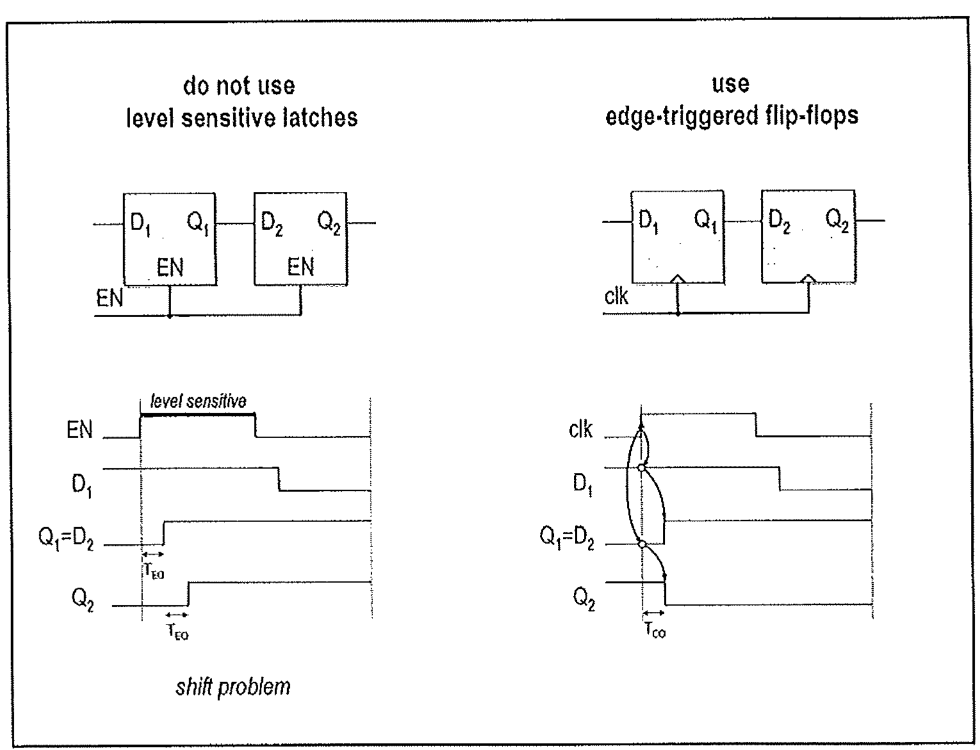

1.3 Sequence Detector
serial bit stream seq
detect ≥ ...01101...
overlapping sequence
seq
det
seq 01011011010...
det (Mealy) 00000010010...
det (Moore) 00000001001...
MOORE
MEALY
preset
det = 1
det = seq
= detected sequence
≠ state assignment

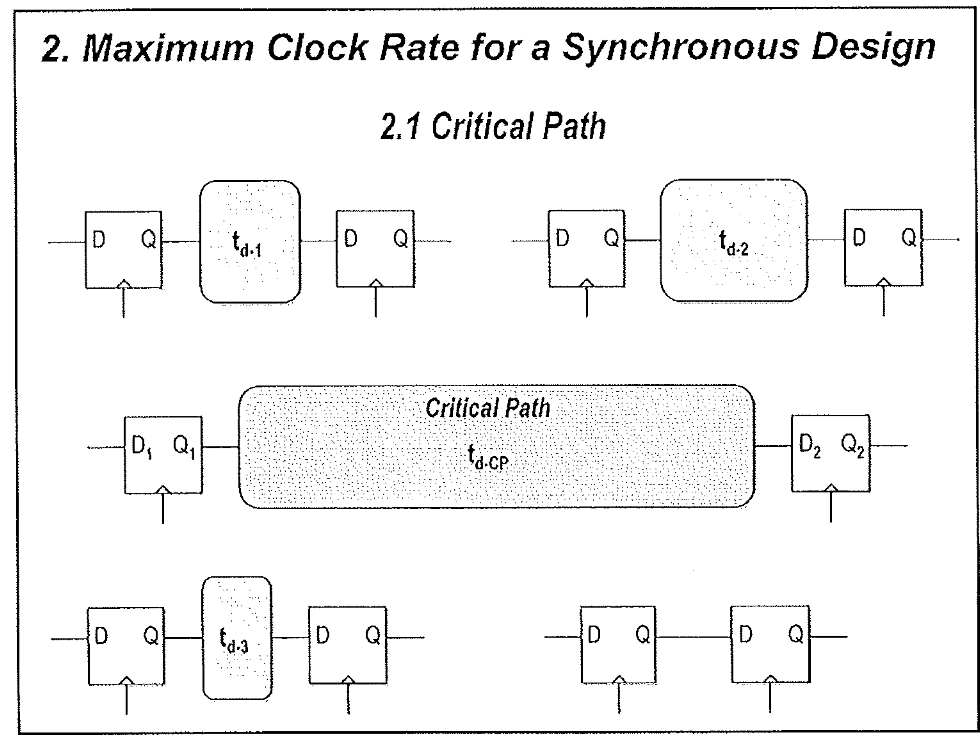

2. Vending Machine
OK
R
reject
1 apple = 5 cent + OK
1c
cnt
2c
open
OK
R
reject
reject = >5 cent + R

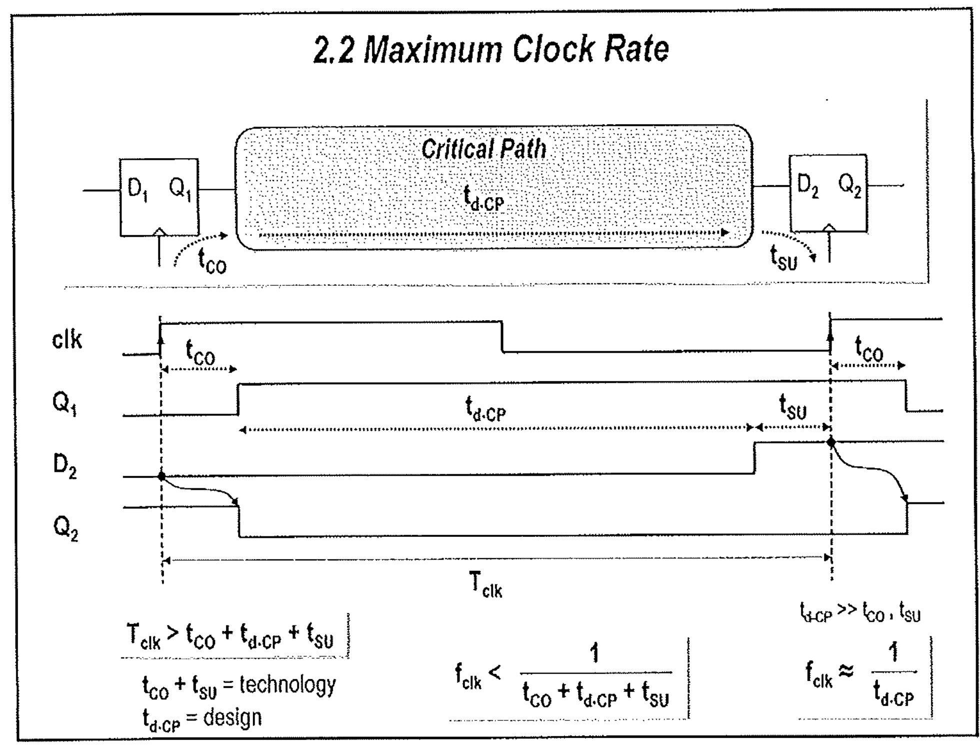

1c
cnt
2c
open
OK
R
reject
R
0
1
2
3
4
5
a
>5
1c
2c
x
z
y
OK
x = $\overline{1c+2c}$
y = $\overline{OK}$.(1c+2c)
z = $\overline{OK}$.$\overline{(1c+2c)}$
open = 1
cnt =
reject = 1
cnt =
state 1..5: reject = R

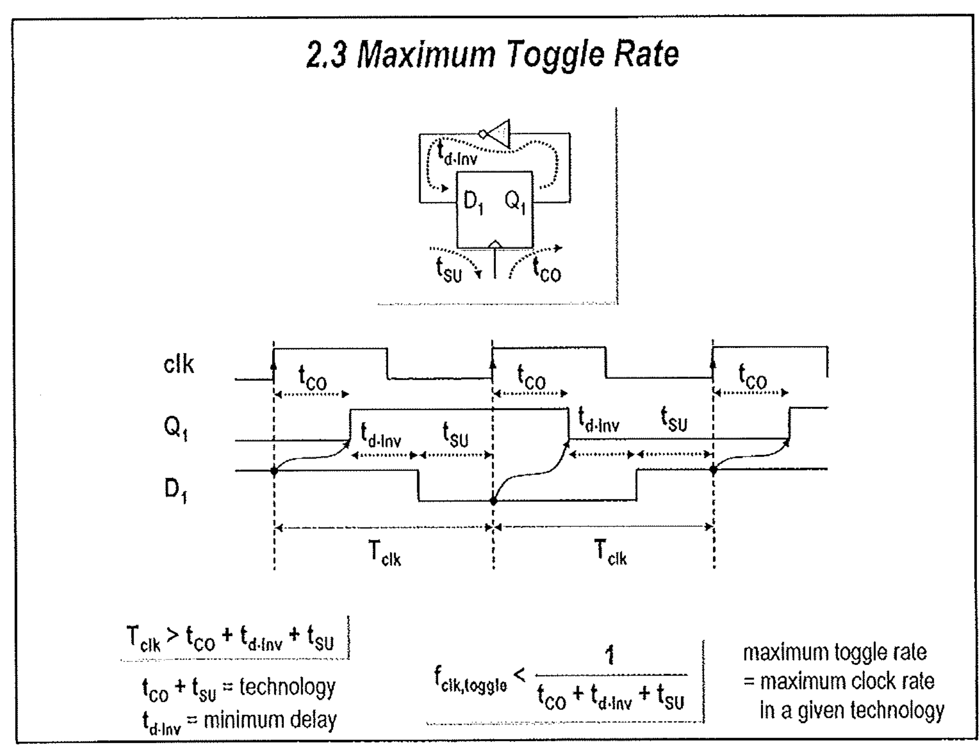
HOGESCHOOL VOOR WETENSCHAP & KUNST
DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER
Digitale Elektronica
Synchronous Design
EmSD
Embedded System Design
ASSOCIATIE K.U.LEUVEN
ir. J. Meel
march 2008

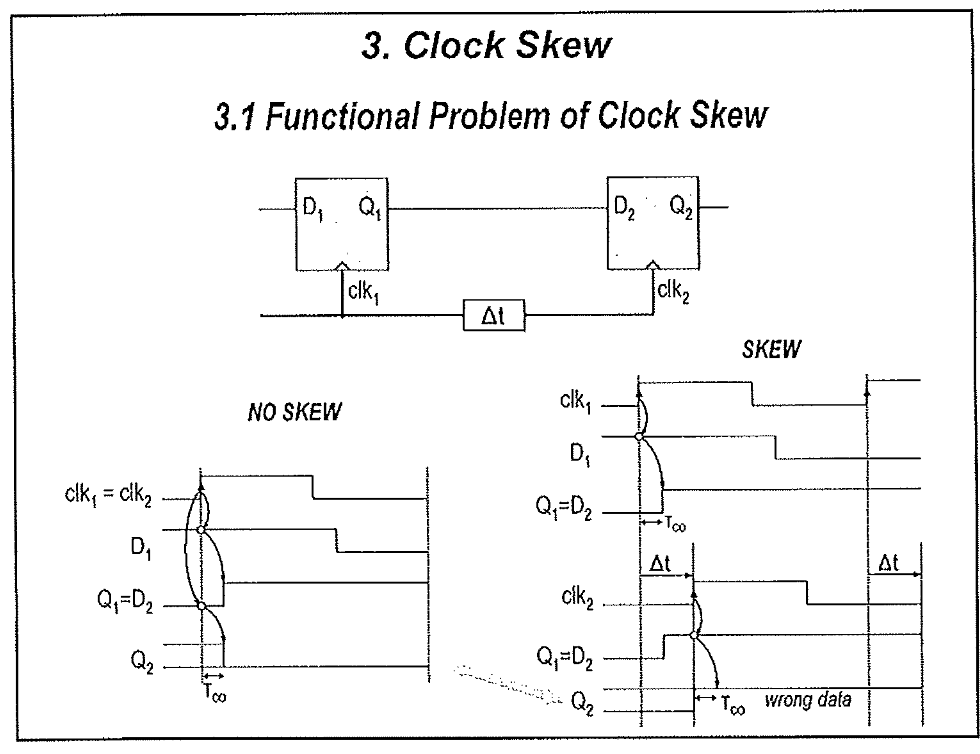

1. Use Edge-Triggered Flip-Flops
do not use
level sensitive latches
use
edge-triggered flip-flops
D
Q
EN
D
Q
clk
level sensitive
EN
D
Q
1-catching problem
clk
D
Q

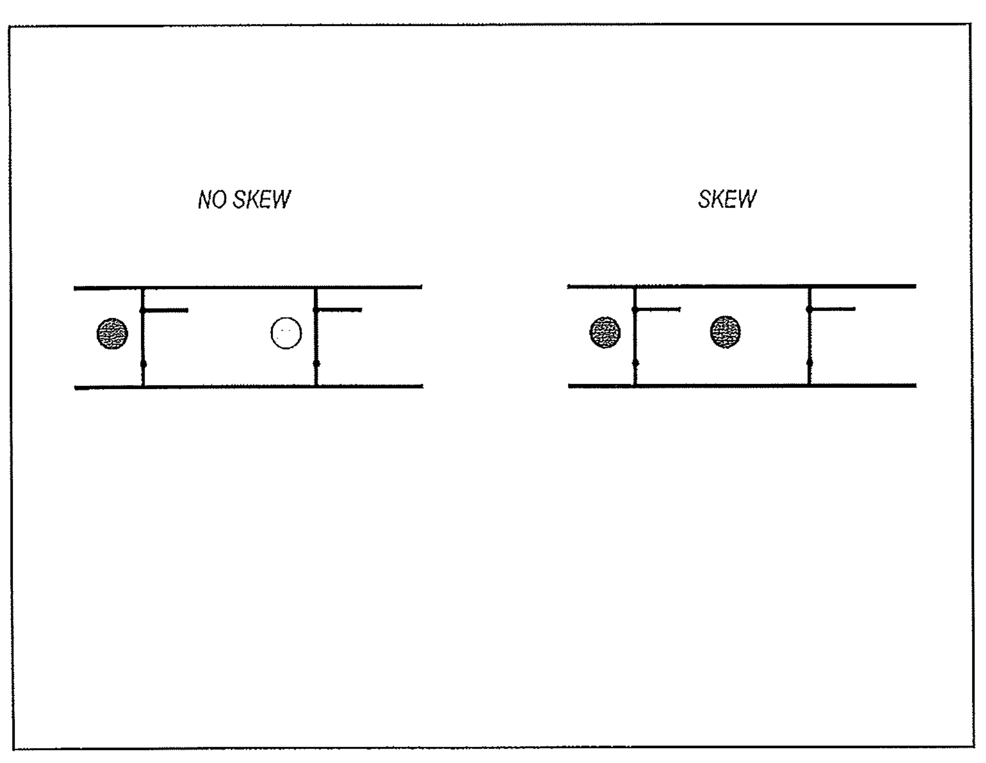

do not use
level sensitive latches
use
edge-triggered flip-flops
D1 Q1
D2 Q2
EN
clk
level sensitive
EN
D1
Q1=D2
Q2
TEO
TCO
shift problem

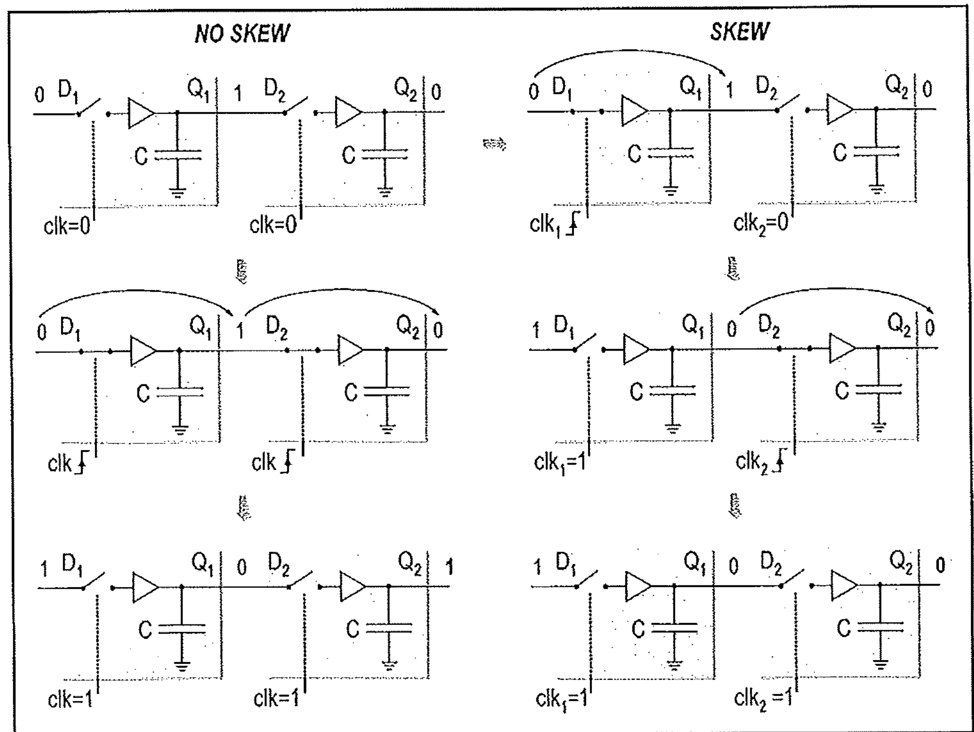

2. Maximum Clock Rate for a Synchronous Design
2.1 Critical Path
D Q
td-1
td-2
td-3
Critical Path
td-CP
D1 Q1
D2 Q2

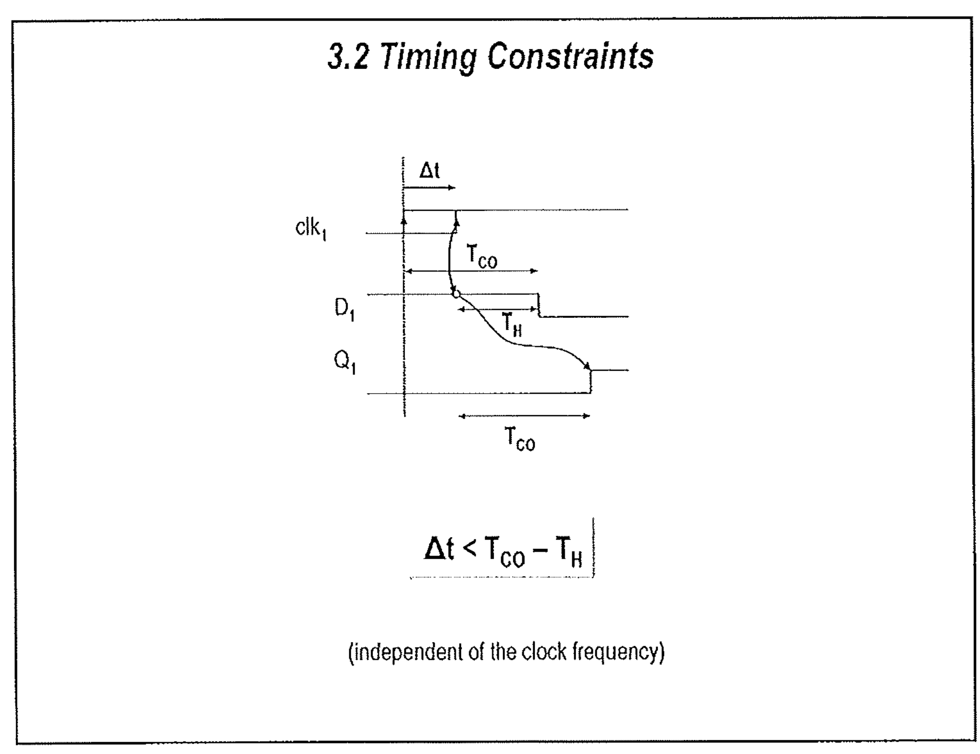

2.2 Maximum Clock Rate
Critical Path
$t_{d\text{-}CP}$
D_1 Q_1
D_2 Q_2
t_{CO}
t_{SU}
clk
Q_1
D_2
Q_2
T_{clk}
$T_{clk} > t_{CO} + t_{d\text{-}CP} + t_{SU}$
$t_{CO} + t_{SU}$ = technology
$t_{d\text{-}CP}$ = design
$f_{clk} < \frac{1}{t_{CO} + t_{d\text{-}CP} + t_{SU}}$
$t_{d\text{-}CP} >> t_{CO}, t_{SU}$
$f_{clk} \approx \frac{1}{t_{d\text{-}CP}}$

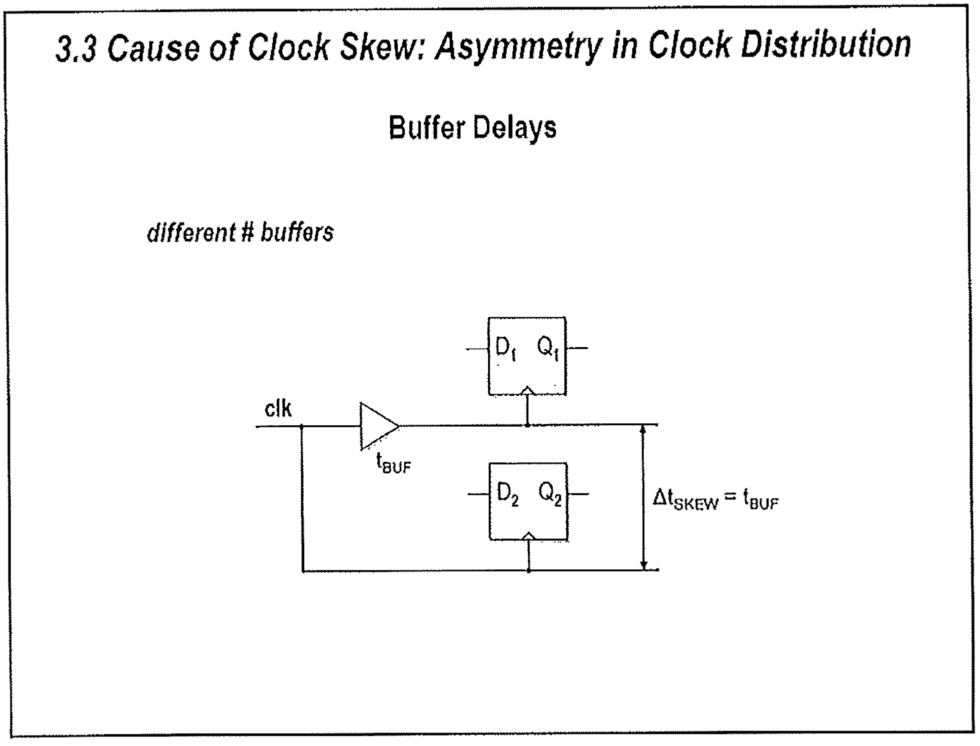

2.3 Maximum Toggle Rate
$t_{d\text{-}Inv}$
D_1 Q_1
t_{SU}
t_{CO}
clk
Q_1
D_1
T_{clk}
$T_{clk} > t_{CO} + t_{d\text{-}Inv} + t_{SU}$
$t_{CO} + t_{SU}$ = technology
$t_{d\text{-}Inv}$ = minimum delay
$f_{clk,toggle} < \frac{1}{t_{CO} + t_{d\text{-}Inv} + t_{SU}}$
maximum toggle rate
= maximum clock rate
in a given technology

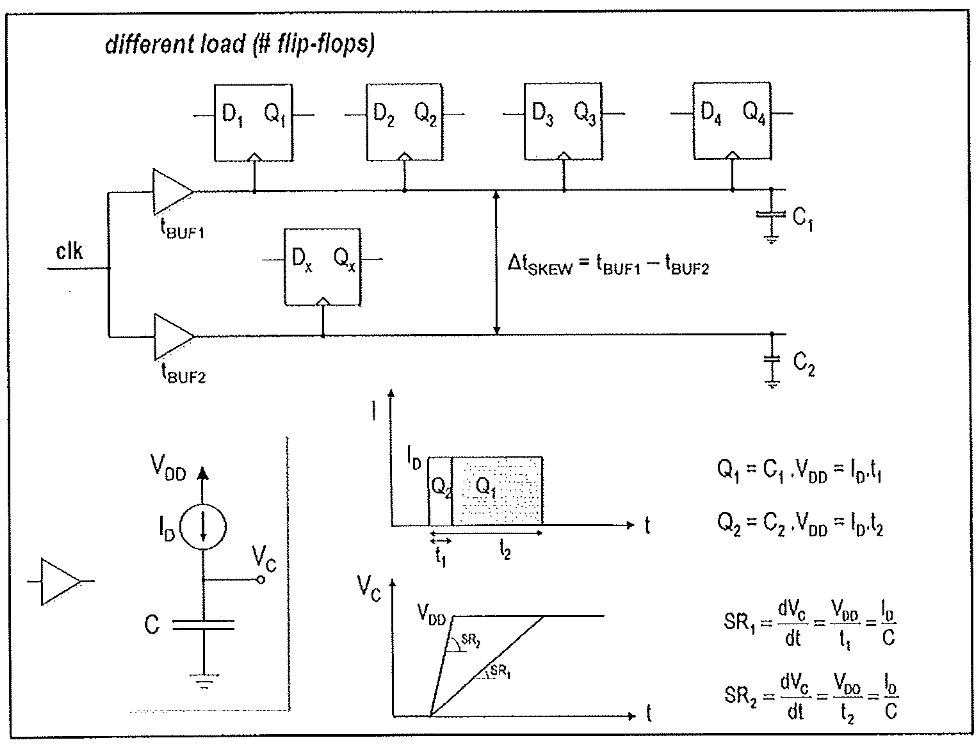

3. Clock Skew
3.1 Functional Problem of Clock Skew
D1
Q1
D2
Q2
clk1
clk2
Δt
NO SKEW
SKEW
clk1 = clk2
D1
Q1=D2
Q2
Tco
clk1
clk2
Δt
wrong data

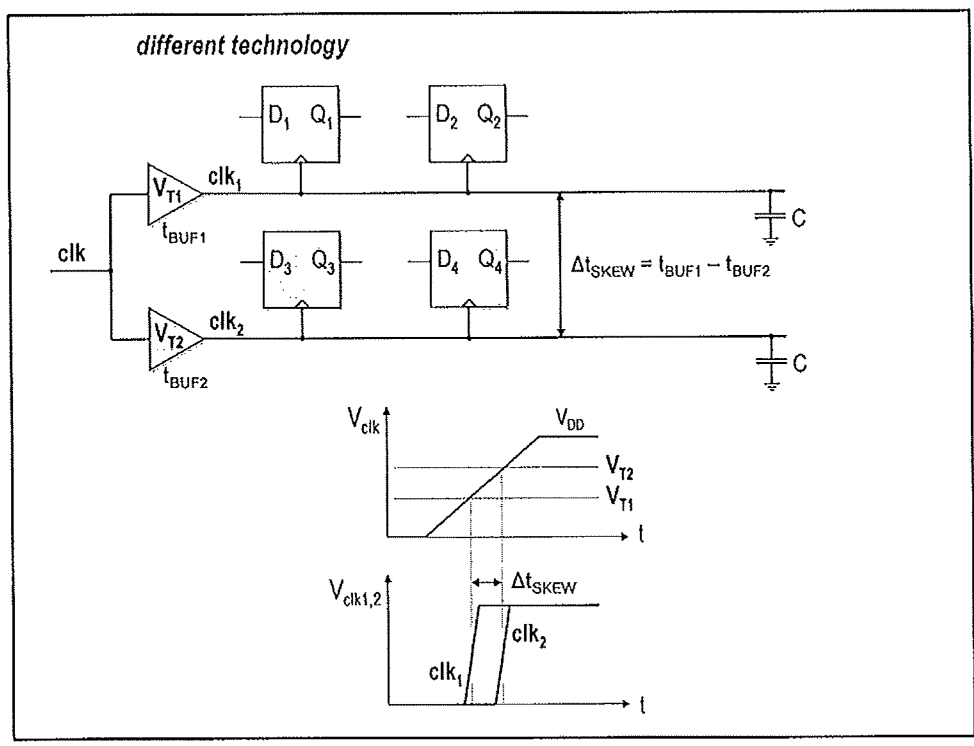

NO SKEW
SKEW

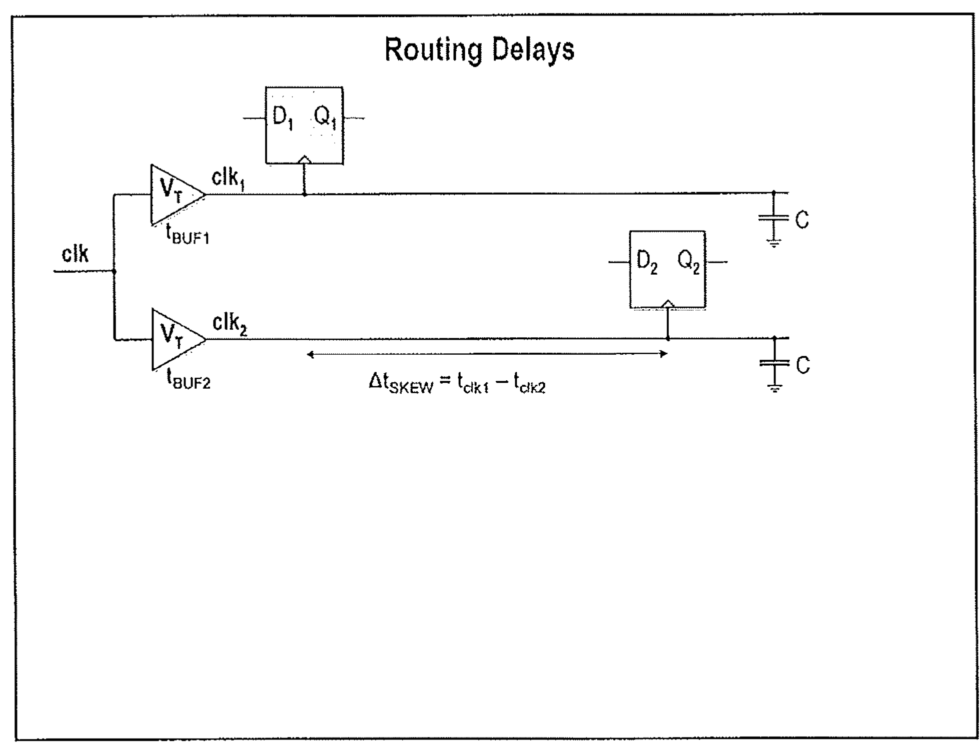

NO SKEW
SKEW
0 D1
Q1
1 D2
Q2 0
C
clk=0
clk1
clk2=0
clk
1 D1
0 D2
clk1=1
clk2
Q2 1
clk=1
clk2 =1

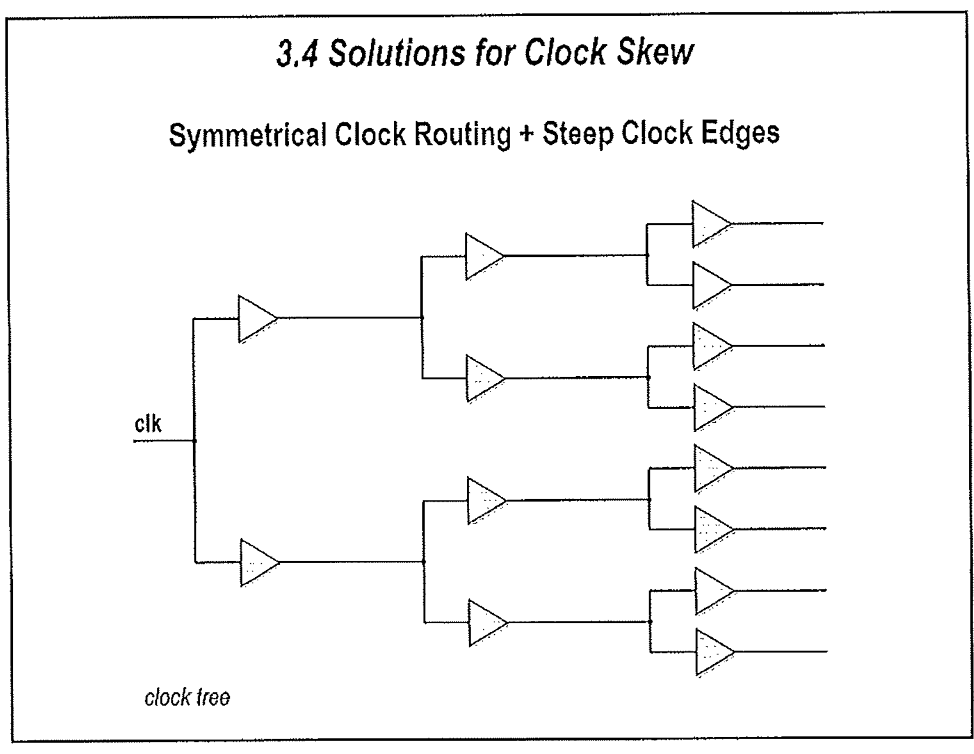

3.2 Timing Constraints
Δt
clk1
TCO
D1
TH
Q1
TCO
Δt < TCO – TH
(independent of the clock frequency)

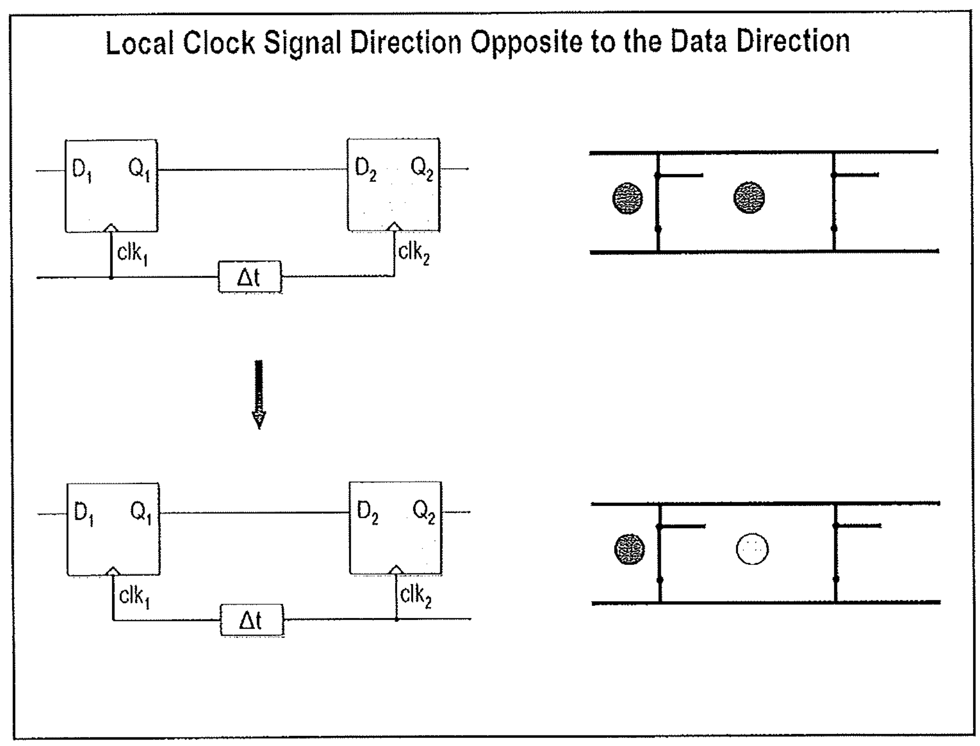

3.3 Cause of Clock Skew: Asymmetry in Clock Distribution
Buffer Delays
different # buffers
clk
t_{BUF}
D_1 Q_1
D_2 Q_2
$\Delta t_{SKEW} = t_{BUF}$

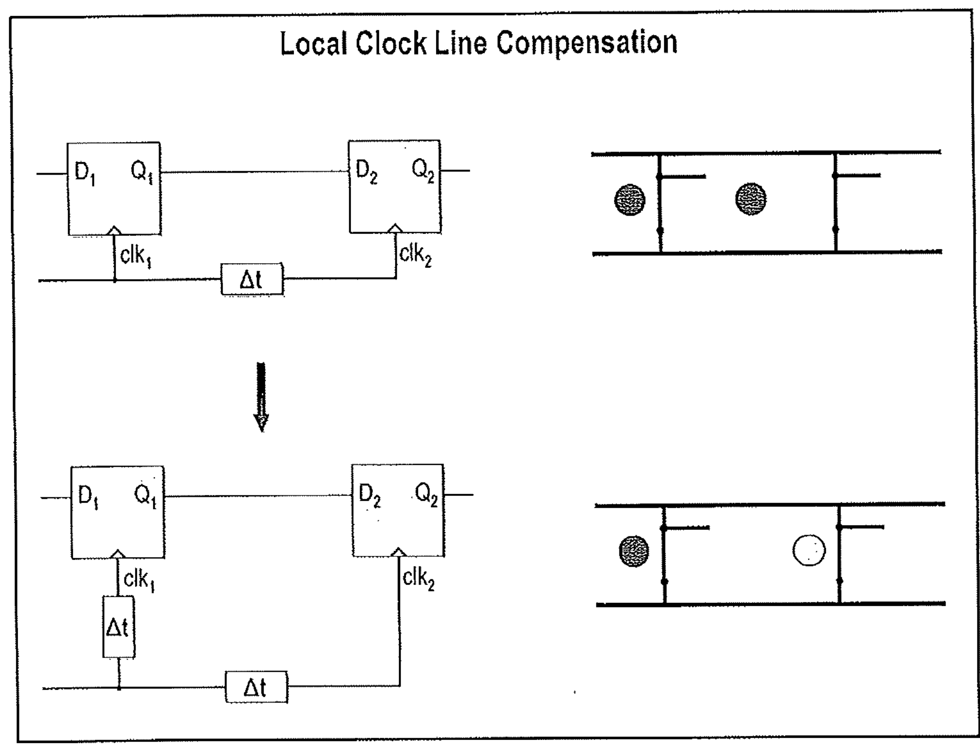

different load (# flip-flops)
D_1 Q_1
D_2 Q_2
D_3 Q_3
D_4 Q_4
clk
t_{BUF1}
D_x Q_x
$\Delta t_{SKEW} = t_{BUF1} - t_{BUF2}$
C_1
t_{BUF2}
C_2
V_{DD}
I_D
V_C
C
I
I_D
Q_2 Q_1
t
t_1 t_2
V_C
V_{DD}
SR_2
SR_1
t
$Q_1 = C_1 . V_{DD} = I_D . t_1$
$Q_2 = C_2 . V_{DD} = I_D . t_2$
$SR_1 = \frac{dV_C}{dt} = \frac{V_{DD}}{t_1} = \frac{I_D}{C}$
$SR_2 = \frac{dV_C}{dt} = \frac{V_{DD}}{t_2} = \frac{I_D}{C}$

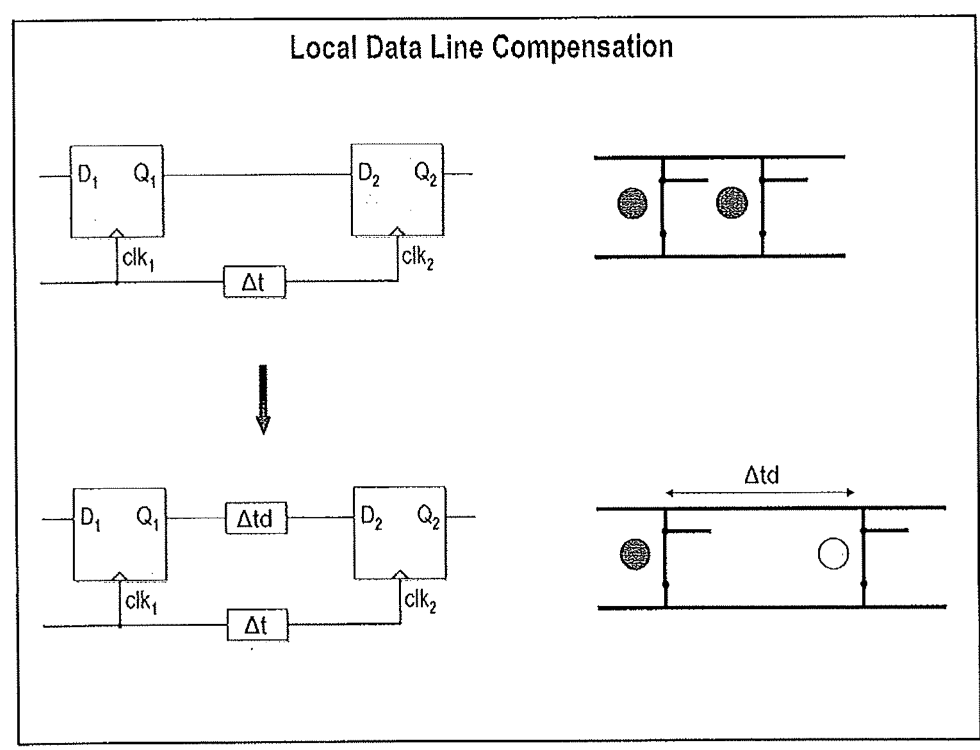

different technology
D1 Q1
D2 Q2
clk1
VT1
tBUF1
clk
D3 Q3
D4 Q4
ΔtSKEW = tBUF1 − tBUF2
C
VT2
clk2
tBUF2
C
Vclk
VDD
VT2
VT1
t
Vclk1,2
ΔtSKEW
clk2
clk1
t

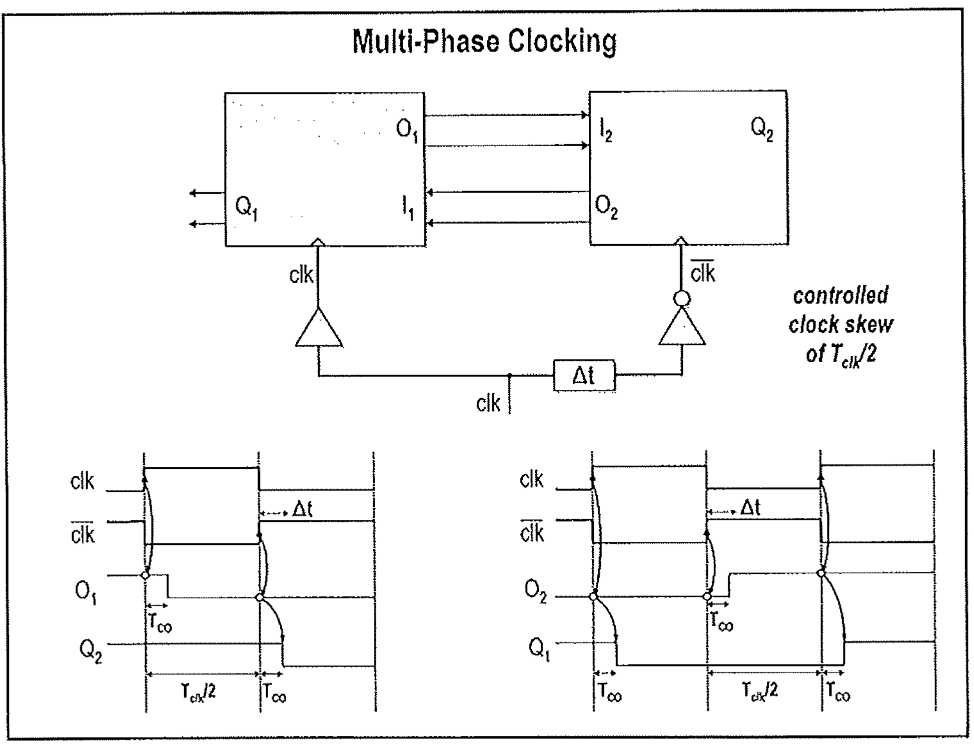

Routing Delays
D1 Q1
VT
clk1
tBUF1
C
clk
D2 Q2
VT
clk2
tBUF2
ΔtSKEW = tclk1 − tclk2
C

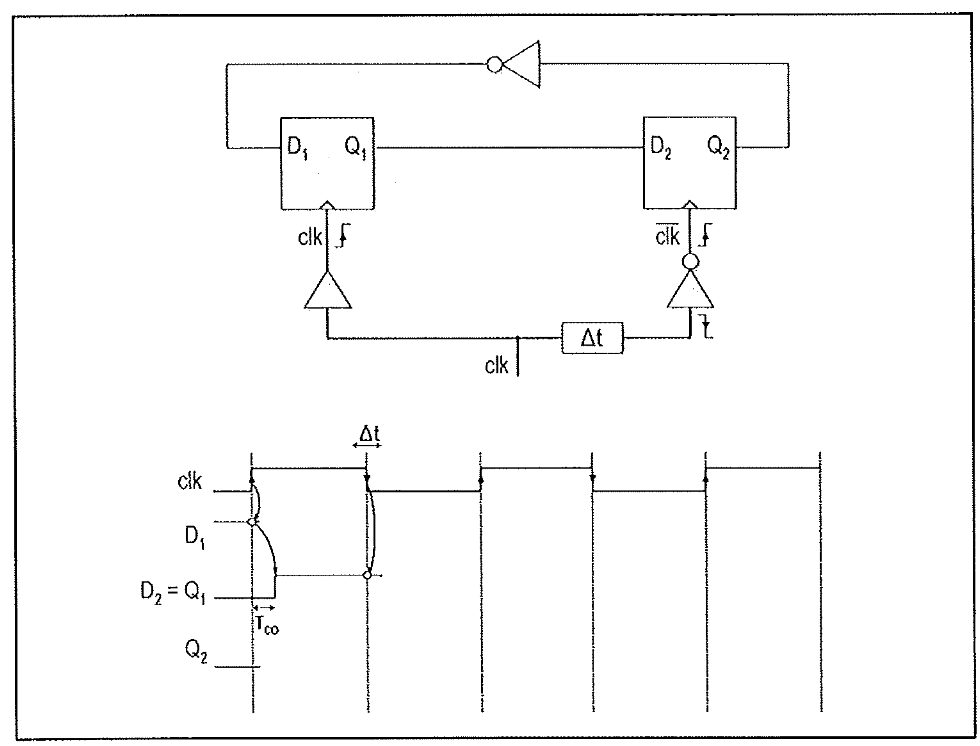

3.4 Solutions for Clock Skew
Symmetrical Clock Routing + Steep Clock Edges
clk
clock tree

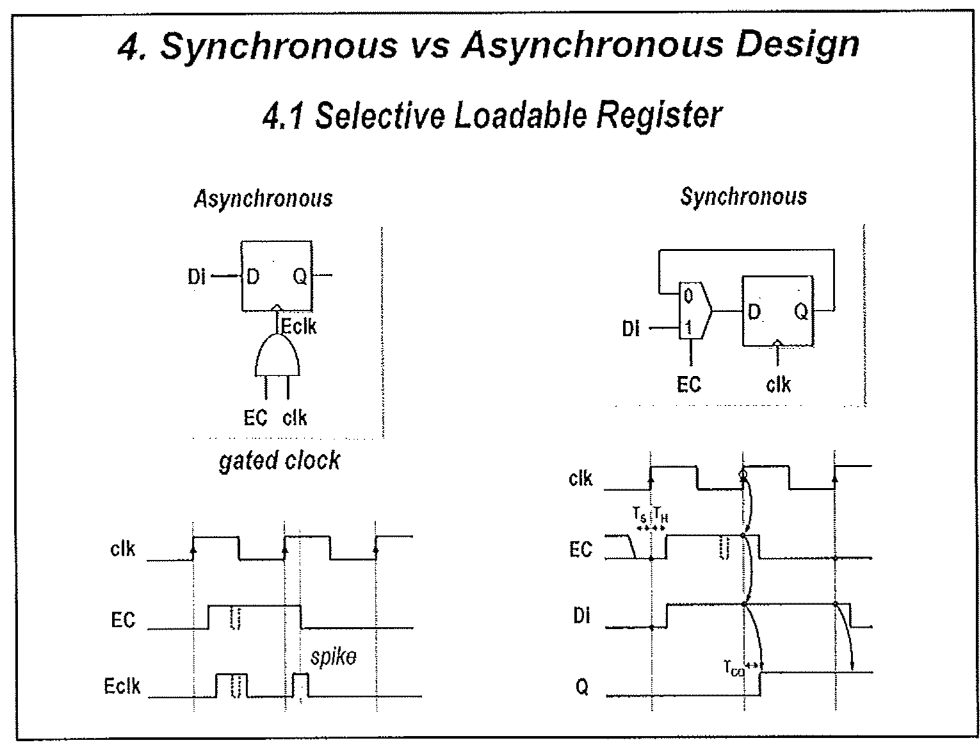

Local Clock Signal Direction Opposite to the Data Direction
D1
Q1
D2
Q2
clk1
clk2
Δt

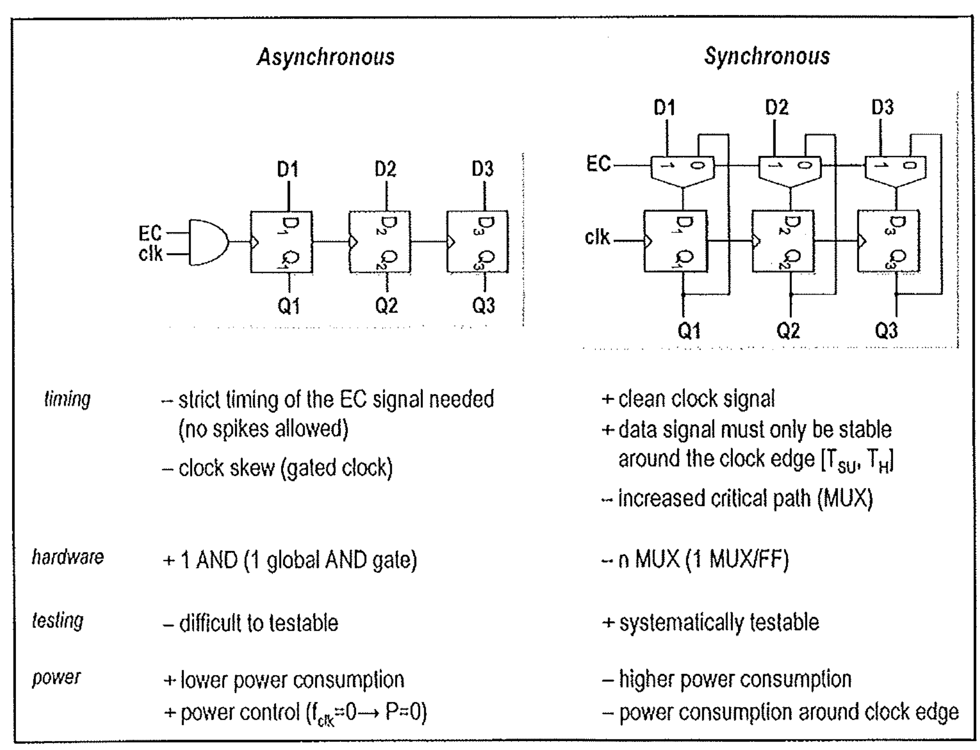

Local Clock Line Compensation
D1 Q1
D2 Q2
clk1
clk2
Δt
D1 Q1
D2 Q2
clk1
clk2
Δt
Δt

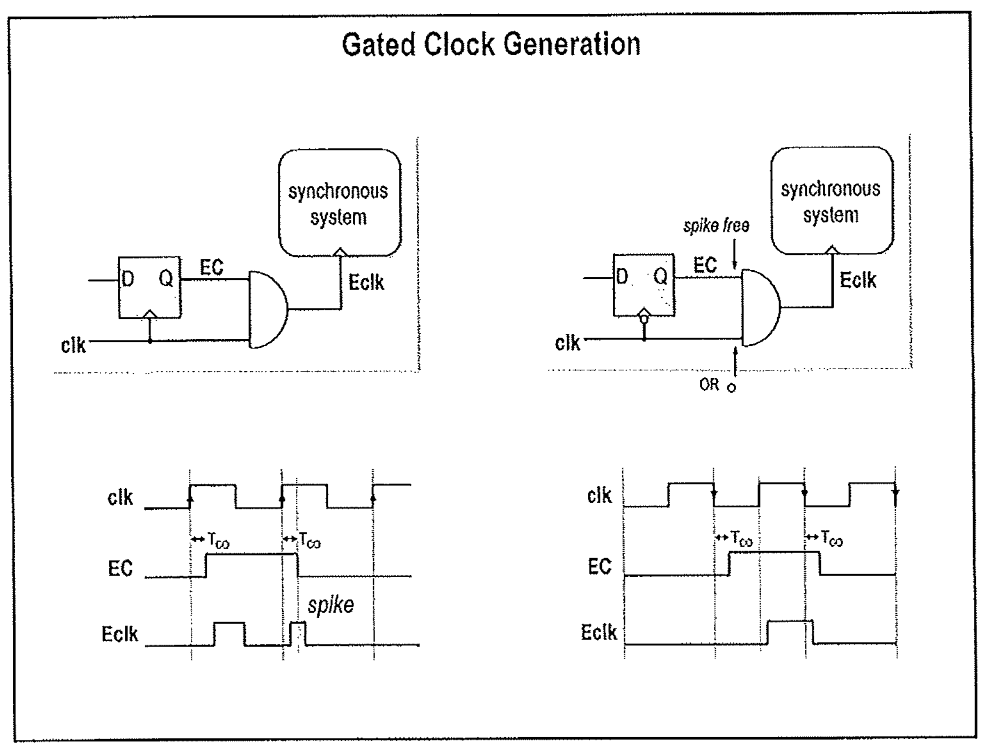

Local Data Line Compensation
D1 Q1
D2 Q2
clk1
clk2
Δt
D1 Q1
Δtd
D2 Q2
clk1
clk2
Δt
Δtd

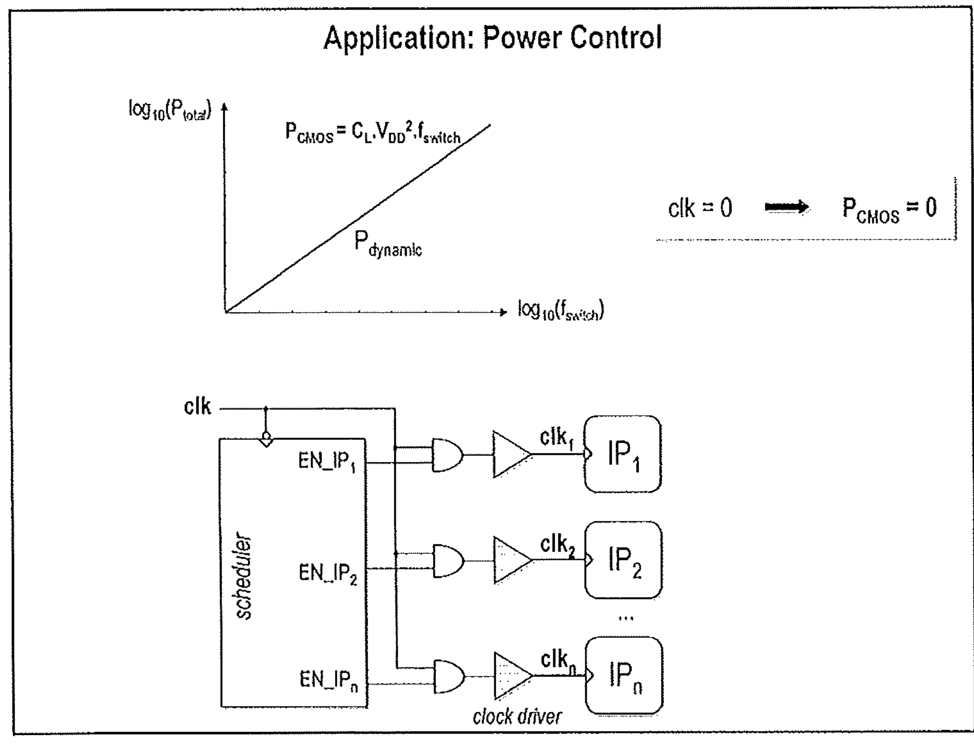

Multi-Phase Clocking
O1
I2
Q2
Q1
I1
O2
clk
clk
controlled
clock skew
of Tclk/2
Δt
clk
clk
clk
O1
Q2
Δt
Tco
Tclk/2
Tco
clk
clk
O2
Q1
Δt
Tco
Tco
Tclk/2
Tco

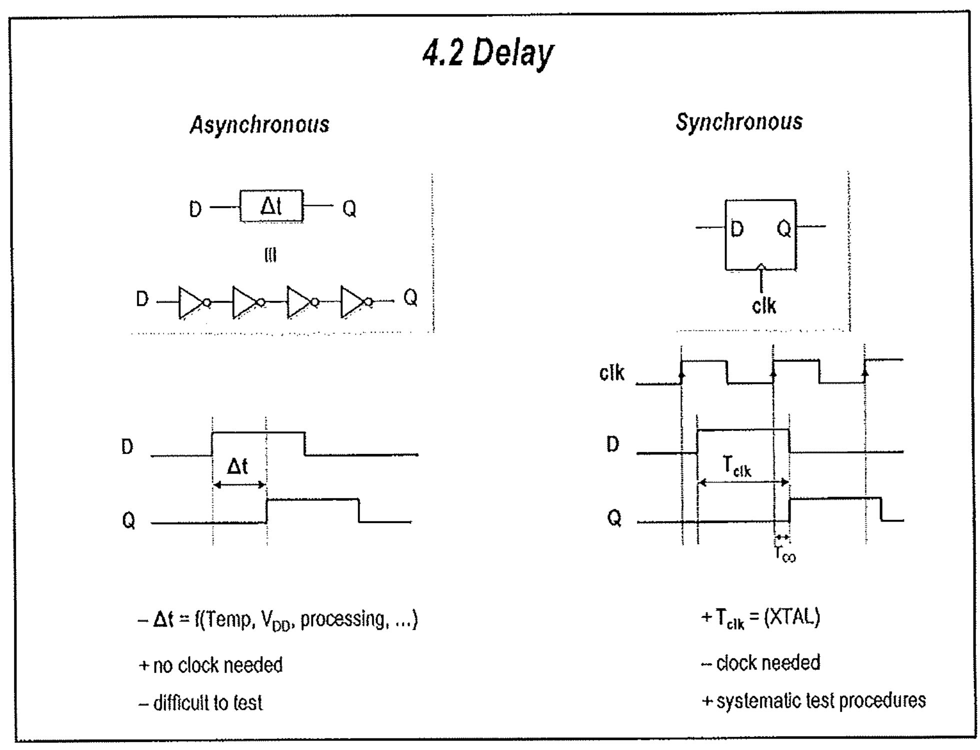

D1
Q1
D2
Q2
clk
clk
Δt
clk
Δt
clk
D1
D2 = Q1
Tco
Q2

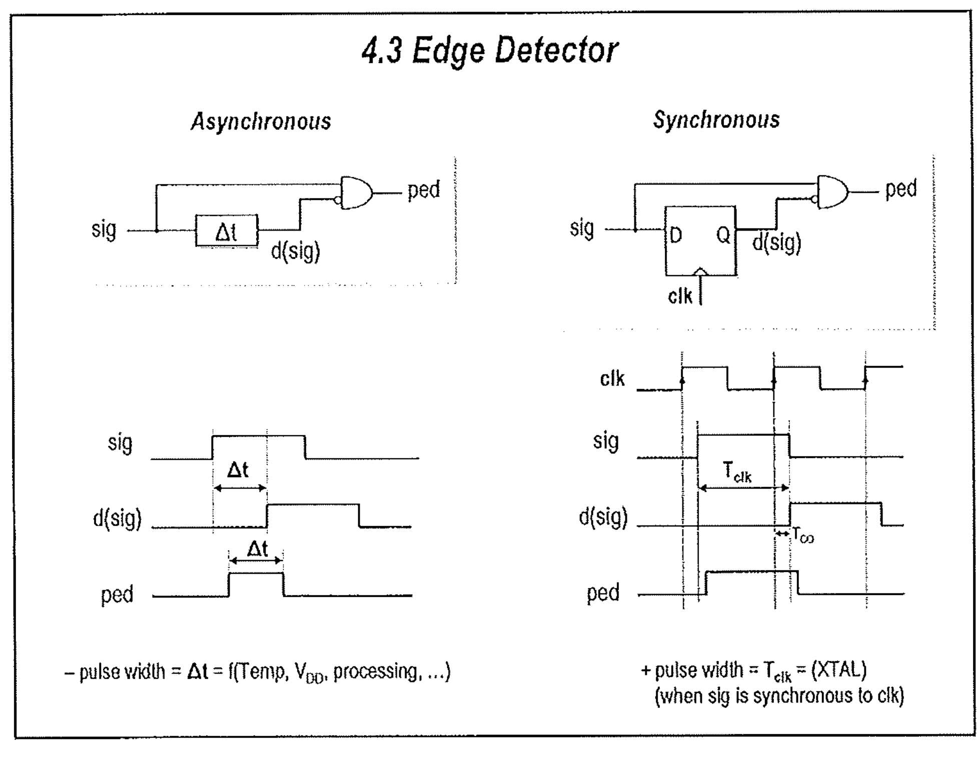

4. Synchronous vs Asynchronous Design
4.1 Selective Loadable Register
Asynchronous
Synchronous
Di
D
Q
Eclk
EC clk
gated clock
0
1
DI
EC
clk
spike
Eclk
Q
T_S T_H
T_{CQ}

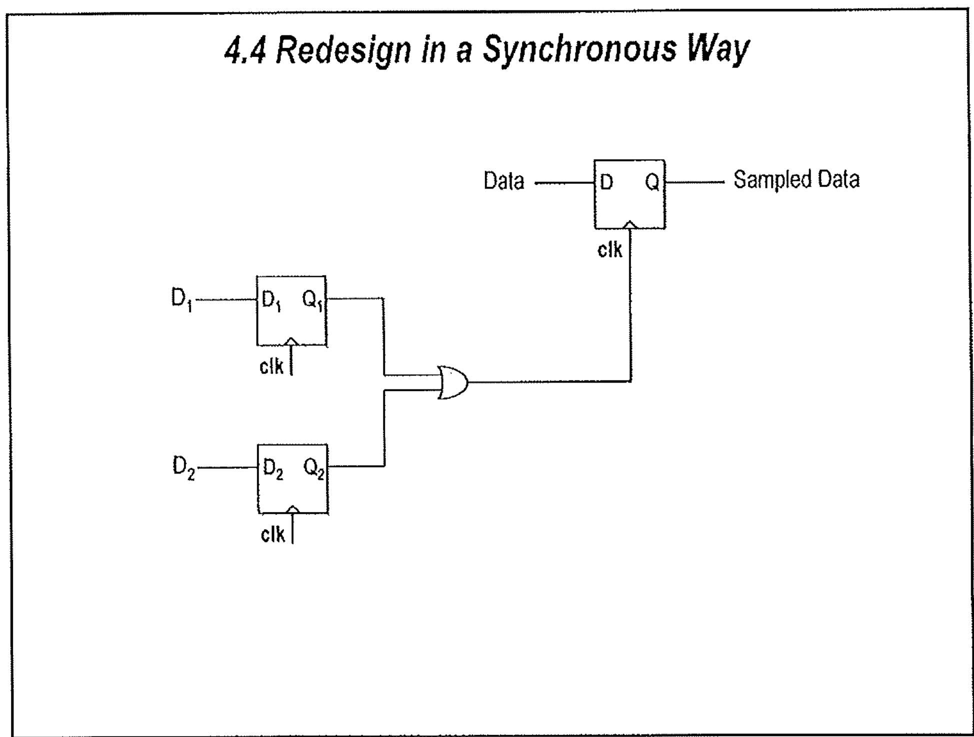

Asynchronous
Synchronous
D1 D2 D3
EC
clk
Q1 Q2 Q3
timing
– strict timing of the EC signal needed (no spikes allowed)
– clock skew (gated clock)
+ clean clock signal
+ data signal must only be stable around the clock edge [T_{SU}, T_H]
– increased critical path (MUX)
hardware
+ 1 AND (1 global AND gate)
– n MUX (1 MUX/FF)
testing
– difficult to testable
+ systematically testable
power
+ lower power consumption
+ power control (f_{clk}=0→ P=0)
– higher power consumption
– power consumption around clock edge


Gated Clock Generation
synchronous system
D Q
EC
Eclk
clk
spike free
OR
T_co
spike


Application: Power Control
log10(Ptotal)
PCMOS = CL.VDD2.fswitch
Pdynamic
log10(fswitch)
clk = 0 → PCMOS = 0
clk
scheduler
EN_IP1
EN_IP2
EN_IPn
clk1
clk2
clkn
IP1
IP2
IPn
clock driver


4.2 Delay
Asynchronous
Synchronous
D
Δt
Q
clk
Tclk
Tco
– Δt = f(Temp, VDD, processing, ...)
+ no clock needed
– difficult to test
+ Tclk = (XTAL)
– clock needed
+ systematic test procedures


4.3 Edge Detector
Asynchronous
Synchronous
sig
Δt
d(sig)
ped
clk
Tclk
Tco
– pulse width = Δt = f(Temp, VDD, processing, ...)
+ pulse width = Tclk = (XTAL)
(when sig is synchronous to clk)


4.4 Redesign in a Synchronous Way
Data
D
Q
Sampled Data
clk
D1
D1
Q1
clk
D2
D2
Q2
clk

The Ten Commandments Of Excellent Design

Instability And Unreliability In Synchronous Systems Can Be Avoided By Rigorously Following These Design Rules.

PETER CHAMBERS VLSI Technology Inc., 8375 South River Pkwy., Tempe, AZ 85284; (602) 752-6395; e-mail: peter.chambers@vlsi.com.

This article is the first of a two-part series written to offer some pointers for designing synchronous circuits that work the first time. Take note of the ten commandments that always should be followed! Part two of the series will appear in an upcoming issue,and will provide VHDL-code examples to support ideas discussed here.

Synchronous digital systems are pervasive in today's designs. Engineers create clocked circuits for every conceivable application, with frequencies ranging from dc to GHz. Every synchronous system has certain common characteristics, and is prone to a group of common faults that can cause instability and unreliability, and may not be uncovered in the typical design process. The net result is a poor product that fails to meet the design criteria.

Poor design results require the engineer to endure costly and time-consuming design modifications and revisions. However, by applying a few simple rules, designers can avoid synchronous faults in designs and achieve consistent first-pass success.

Digital Systems 101

We'll begin by describing a typical synchronous circuit *(Fig. 1)*. Many variations are possible, but a simple, one-clocked element example, showing the circuit and timing, will adequately illustrate the sources of error.

With such problems, why use synchronous logic? Wouldn't asynchronous logic be faster? The answers could fill a book, but here are some reasons to use synchronous design:

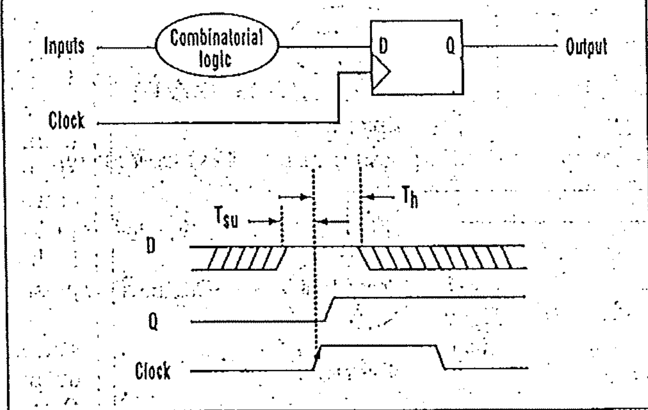
- Synchronous designs eliminate problems associated with speed variations through different paths of logic. By sampling signals at well-defined time intervals, fast paths and slow paths can be handled simply.
- Synchronous designs work well under variations of temperature, voltage, and process. This stability is key for high-volume manufacturing.
- Many designs must be portable. In other words, they must be easily migratable to a new and improved technology (for example, moving from 0.6 µm to 0.35 µm). The deterministic behavior of synchronous designs makes the process of moving to a new technology very straightforward.
- Interfacing between two blocks of logic is simplified by defining standardized synchronous behavior. Asynchronous interfaces demand elaborate handshaking (or token passing) to ensure integrity of information. However, synchronous designs with known timing characteristics can guarantee correct reception of data.

Synchronous circuits are made with a mixture of combinatorial logic and clocked elements, such as flip-flops or registers. The clocked elements share a common clock, and all transition from one state to another on the rising edge of the clock. When the rising edge occurs, the registers propagate logic levels at their D inputs to the Q outputs.

Figure 1, defines two important timing parameters:

- Setup Time (T_{su}): Setup time is the time that a register's D input must be valid before the clock transitions.
- Hold Time (T_h): Hold time is the period that a register's D input must be valid after the clock has transitioned.

If the setup- or hold-time parameters are violated, terrible things happen. We'll discuss this later in the section on synchronization.



1. This simple circuit introduces the notation and timing parameters for a D-type flip-flop, which is the basic element of synchronous circuits.

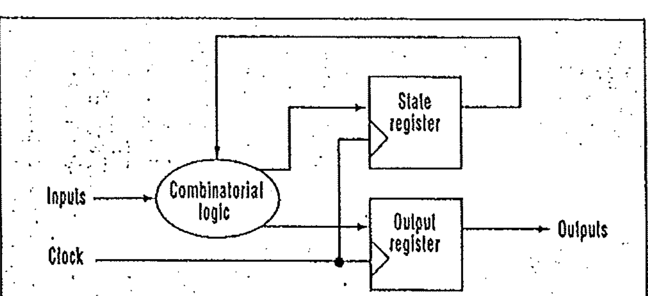
Clock Distribution

The distribution of clocks throughout a design has received considerable attention with the increase in logic speed. Common personal computers have bus speeds of 66 MHz, and processor clocks

run at 150 MHz or greater. In this article, we're concerned more with the possible pitfalls in the synchronous logic itself, not with the production of decent clocks. However, here are the important parameters needed for a good clock distribution system design:

•Skew Minimization: Clock skew is the variation in time of the clock's active transition being detected by different devices within a system. Skew must be kept to a minimum to ensure that setup and hold times are not violated at any one device. Methods for managing skew include equal-length traces, zero-delay PLL-based buffers, and additional logic for extending hold times.

•Clock Fidelity: The clock's waveform must be as clean and deterministic as possible. Techniques used to guarantee consistent clock behavior include transmission-line termination, ground-bounce minimization, and the use of identical clock drivers.



3. A better state machine registers the outputs as well as the current state. That way, it's possible to produce glitch-free outputs for latch control, register loads, and tristate enables.

Good State Machine Design

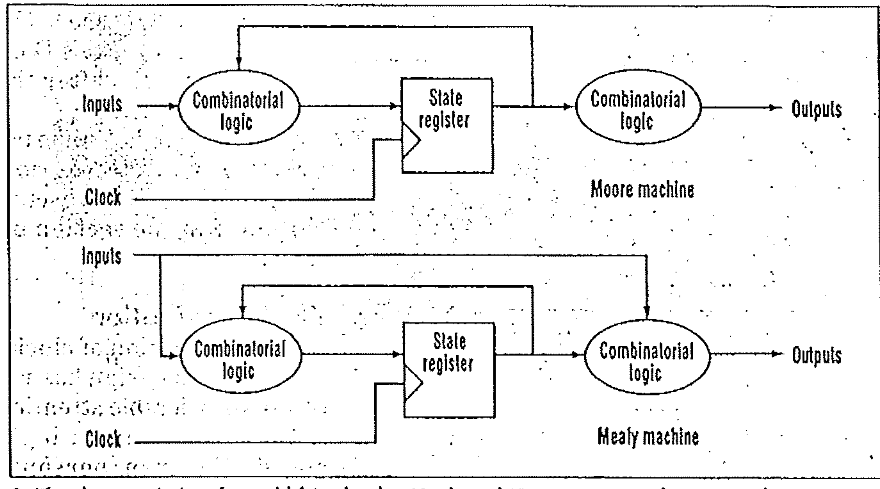
One of the designer's most powerful constructs for synchronous design is the state machine. By combining combinatorial logic and a number of registers, the state machine makes decisions based on its inputs and its current state. The behavior of the state machine is entirely synchronous, with all decisions made at the time of the clock transition. There are two conventional forms of state machines: Mealy and Moore *(Fig. 2)*.

Moore machines are the simpler of the two standard types. Their output is a function only of the current state of the machine. The outputs of Mealy machines are a function of the current state of the machine plus the inputs. This additional path provides more flexibility, but may complicate the understanding of the machine.

Books on high-level design languages (HDLs) expound at great length on the construction of state machines. But the results are frequently disappointing. Defining a state machine in an HDL and running the design through a synthesizer could produce spaghetti logic that no self-respecting designer would ever put together.

Mealy/Moore Drawbacks

Moore state-machine outputs are combinatorial decodes of the current state, and Mealy state-machine outputs are combinatorial decodes of the current state and the inputs *(Fig. 2, again)*. While this arrangement is fine in principle, there are pitfalls here waiting to trap the unwary.



2. The characteristics of our old friends, the Mealy and Moore state machines, are shown in this diagram. Their behavior is entirely synchronous.

The outputs of the state machine may include the following types of functions:

•Latch enables (low- or high-going pulses to open or close latches)

•Tristate enables (signals turning drivers on and off to on-chip or off-chip buses)

•Register enables (enables for synchronously-clocked registers)

•General control signals, such as counter enables, flags, and others.

Most of these signals have one characteristic in common—glitches are absolutely unacceptable at any time. As the state registers and inputs of the Mealy or Moore state machines transition and settle, the combinatorial gates are capable of generating glitches as a consequence of the varying gate propagation delays. These transitory glitches may well contain enough energy to open latches, clock registers, and cause other highly-undesirable effects.

What About Gray Code?

We all learn early on that gray code counters are wonderful because only one bit changes at a time. When fed to an asynchronous decoder, theory suggests that the outputs should settle to their new state without noise. I'm suspicious of this when the implementation is created by synthesized logic; unclocked feed-forward paths might well negate the advantage of gray code.

There is, however, a greater challenge to the use of gray code. The sequence of transitions taken by a state machine as it operates is likely to be quite elaborate. Many state machines are very complex, with numerous branches between the possible states. Because gray-code-driven decodes are only glitch-free when a single bit changes at each clock edge, the designer must ensure that all possible state transitions result in a change in just one bit of the state variable. This technique is practical in only the simplest of state machines.

There is a much better design for a state machine *(Fig. 3)* By adding an output register (with cleanly-clocked D-type flip-flops) that's reloaded at each clock edge, the outputs of the state machine are guaranteed to be

glitch-free. It's suggested that all state machines be implemented in this form, because the quality of the outputs is independent of the number of states or outputs.

Reset signals are traditionally asynchronous, and are routed directly to the clear inputs of state-machine register elements. When the reset is asserted, all registers (state and output bits) are cleared immediately. All well and good, but what happens when the reset is deasserted?

Consider a state machine that transitions from the reset state to another state directly after the reset is deasserted. If reset deasserts close to a clock edge, some of the state bits will assume their new states while others might not. The state machine ends up in an undefined error state.

The solution? Synchronize that darned reset! That way, the reset will be removed well before the clock edge, and all register elements will correctly transition to their new states.

In fact, every input to the state machine must be synchronous. At the very least, be absolutely certain that no input will violate the setup and hold times of the state machine's state and output registers.

State machines with encoded state bits don't always use all possible states. For example, for a 20-state state machine, use a five-bit state register. This design leaves 12 unused state values. Because states are usually counted incrementally from zero, the example would use states 0-19 for normal operation, while states 20-31 wouldn't be used at all, hence the name dead states.

If the state machine ever enters states 20-31, errors are likely. Even worse, the machine may totally lock up, with the state machine forever in one of these illegal states. It may require a hard reset to recover from this condition.

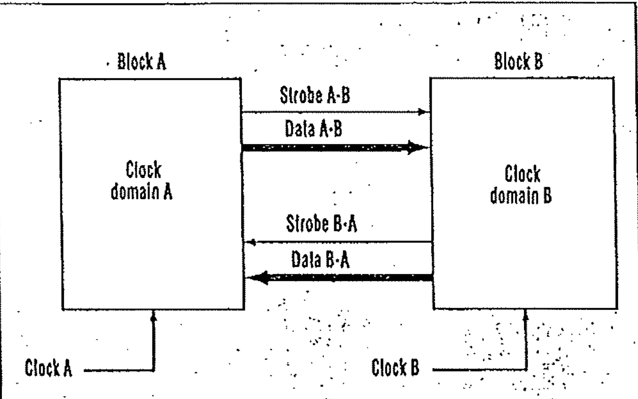
Clearly, it's best to ensure your state machine never reaches a dead state. However, a robust design will at a minimum guarantee that if the state machine does enter a dead state, it will exit the dead state immediately, and then, perhaps enter a quiescent state.

Crossing Clock Domains

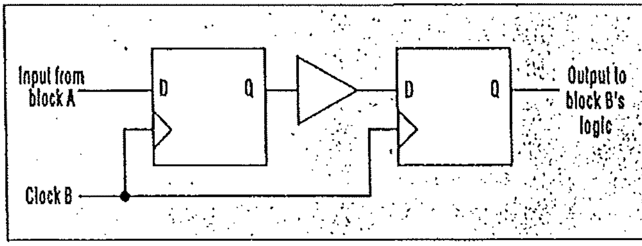
Moving information from one clock domain to another is much like descending into Dante's inferno. All sorts of evils lie in wait to beset the naïve. Set-up and hold violations, metastability conditions, unreliable data, and other perils are manifest when moving from one clock domain to another. In fact, the whole issue of synchronization merits its own article. But these tips might help to resolve the block-to-block synchronization issues.

First, let's define the problem *(Fig. 4)*. Logic Block A operates with Clock A, while logic Block B operates with Clock B. No assumptions have been made about the frequencies of Clock A and Clock B, nor is it assumed that any integer or multiple relationship exists between the two; the two are totally independent.

A strobe is sent from Block A to Block B (Strobe A-B), and some data is sent, Data A-B. In response, Strobe B-A and Data B-A return. The transmission of information between the blocks must be absolutely reliable, raising the issue of cross-domain synchronization.



4. In this environment, signals and data venture from one clock domain into another.



5. Two flip-flops provide the easiest way to synchronize an asynchronous input signal.

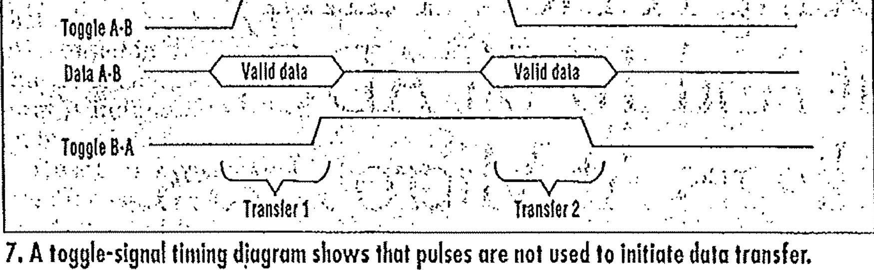
Synchronization 101

Crossing between clock domains is an issue that's similar to managing asynchronous inputs. Because no relationship between the multiple clock domains can be assumed, the inputs from Block A to Block B must be assumed to be asynchronous inputs.

The traditional way of synchronizing an asynchronous input signal is as follows: two D-type flip-flops are used because two synchronization stages are usually sufficient *(Fig. 5)*. Only the rarest applications demand three stages of synchronization. If your silicon library supports metastable-hardened flip-flops, then the first stage should use such a device. Typically, metastable-hardened flip-flops guarantee that their Q outputs will settle after a given maximum time, no matter how close the data transition is to the flip-flop's clock edge.

This method of information interchange has one drawback. If the strobe has the form of a pulse, it may not be seen by the destination block if the pulse width is less than the destination block's clock (sampling) frequency. This design is not a problem if the two blocks exchange levels

6. Toggle signals provide an efficient, unambiguous way to carry information between clock domains. This figure shows how toggle signals qualify the data path between domains.



7. A toggle-signal timing diagram shows that pulses are not used to initiate data transfer. Instead, transitions (rising or falling edges) carry all the information.

instead of pulses. However, it's slow, typically as four level exchanges must occur for a two-way handshake. The toggle method described later is an excellent solution to this problem.

Single-Point Information

Imagine that Block A needs to send two bits of information to Block B. We could simply duplicate the circuit, with one synchronization circuit for each bit *(Fig. 5, again)*. But there's a serious problem that should be clear: occasionally, one bit gets through the two-stage synchronization circuit while the other does not. The result is ambiguous information and errors.

The solution is shown back in Figure 4—use a single strobe from Block A to Block B, and send the rest of the information separately. The single-point strobe from A to B informs the destination block that thé Data A-B is valid, and the originating block ensures that there is adequate setup time.

Toggleo, Toggleas, Toggleat

A nifty way of doing a two-way handshake without worrying about levels and pulse widths is to use a toggle-exchange protocol *(Fig. 6)*. The signal from Block A to Block B, indicating the data (Data A-B) is valid, is a transition of the signal Toggle A-B. This transition may be low-to-high or high-to-low. Both transitions have the same meaning: the Data A-B bus is valid *(Fig. 7)*.

Each transfer is complete with only two events: a toggle of the two Toggle strobes. While each toggle must be synchronized carefully at the receiving end, this method guarantees successful transmission and reception of wide data busses across clock domains of arbitrary frequency. From GHz to KHz, the toggle method is predictable and reliable.

When creating a set of clocked elements, there's often a compelling reason to use latch-based designs. A single-bit register implemented with a latch may use just 60% of the gates that a conventional D-type flip-flop requires. If your design uses great numbers of configuration registers, FIFOs, or has elaborate data paths, the savings from using latches might be considerable. And the latch control might be the same signal as the clock enable to a D-type flip-flop, so why not use latches?

The latch's Q output is stable while the latch is closed *(Fig. 8)*. When the latch is open, the input is continuously copied to the output. Two potential pitfalls exist with latches:

• Noisy Inputs—Any glitches on the latch's D input are propagated directly through to the output. This design is, of course, manageable by ensuring that there aren't any glitches on the input. However, in a synchronous system, busses tend to switch states at clock edges, and the latch enable typically straddles a clock edge, requiring that the D input be perfectly clean right through the same clock edge. This is the worst time for switching noise, particularly on wide busses.

What's more, the latch needs the D input to be stable for two clock periods (so it's clean through the clock edge). Changing the D input with the same edge that closes the latch, produces a race that you're bound to lose (Murphy and his law).

• Noisy Latch Enable—Perhaps worse than noise on latch inputs is noise on the enable line. If a latch enable glitches as a result of an asynchronous decode, your design is toast.

The first part of this article discussed how to eliminate glitches on decoded signals. However, if you get it wrong, a register-based design is still likely to be robust because glitches on clock enables don't matter except when the clock transitions. But glitches on latch enables always mean instant death when they occur.

Registers Rule!

Register-based designs suffer from none of the disadvantages listed above. Race conditions are rare to nonexistent, glitches on the control or D signals are unlikely to cause harm, and signals can be reliably latched in one clock period. A register-based design may be larger than its latch-based equivalent, but it will be more robust and will contribute toward first-silicon success. Bottom line: If you absolutely have to use latches, beware!

The classic example of a race condition shows that the transition as the output of the first flip-flop changes might well violate the hold time on the D input of the second flip-flop *(Fig. 9)*. This situation can worsen if there is skew between the clocks to each of the two flip-flops. If flip-flop B's clock lags A's, then B's output might actually replicate the output of A, rather than

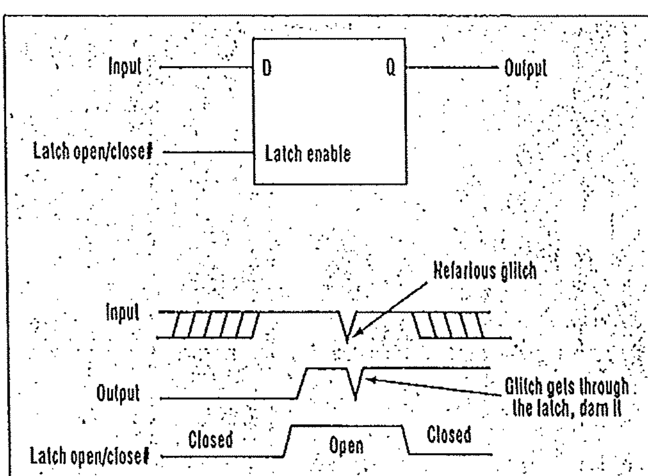
The Ten Commandments Of Excellent Design

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.

add the extra clock delay that is required.

A simple delay solves the fast-path problem *(Fig. 10)*. The delay element ensures that there is sufficient time for flip-flop B to complete its transition before the result of A's transition reaches B.

Some synthesizer tools have a fix hold option that claims to take care of this situation. But if your design fails, who gets the blame, the designer or some well-hidden option in a synthesizer? Check carefully for fast paths.



8. This diagram illustrates the workings of a basic, transparent latch. Propagation of glitches is a less-desirable feature of latch circuits.

If all goes well, the chip will enter mass production and the world (or at least the shareholders) will rejoice. To get to mass production, the design must be testable. Because testability is a much-neglected aspect of many designs, these tips might help test engineers sleep better at night:

Sympathy For The Test Engineer

•Break long counters into bite-size chunks. Counters require lots of test vectors to ensure that all bits toggle correctly, and that carry bits are generated as they should be. To keep the number of test vectors to a reasonable number, provide the ability to partition a counter into multiple, smaller (for example, four bits each) counters.

Provide visibility of the most significant bit of each stage. That way, the test sequence can verify that every counter stage works by observing the most significant bit's low-to-high and high-to-low transitions, and can reasonably conclude that the counter will work as a unit. You can use the partition-and-provide-visibility method to increase testability of any long, sequential piece of logic. Counters are discussed here because they're the most common example of this type of design.

•Asynchronous feedback paths are a federal offense. Even without considering the effect they have on a test engineer's disposition, logic that uses asynchronous feedback is generally bad for a number of reasons. It's hard to simulate; it may be dependent on voltage, temperature, and process. It also may be very susceptible to transients. Just as bad, it may be impossible to test on a fixed-frequency tester.

If there are unclocked feedback paths in your design, make sure that they can be broken and analyzed from the tester. Better still, get rid of them altogether.

It's tempting to say, "I'll just design it quickly, then find the bugs in simulation." But it's a bad idea that's doomed from the start. Simulators are notorious for hiding the quirky details of a design. Examples include:

•Clock Synchronization—Synchronizing flip-flops constantly battle metastability and glitching inputs. Their behavior is not even closely approximated by an average simulator. All you see is a clean transition at the clock edge. Crossing clock domains must always be correct-by-design from the earliest stages.

•Asynchronous Logic—Similarly, asynchronous logic is often simulated poorly. Certainly, fast paths and race conditions may be hidden. Some situations will determine (and optionally correct) hold-time violations, but this is not a universal panacea for correct asynchronous logic.

Correct-By-Design

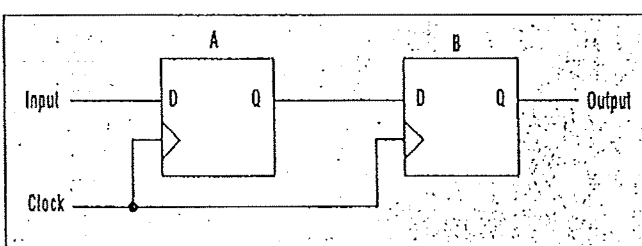
When designing logic that is outside the protected realm of clock-to-clock register-to-register implementations, the only solution for robust design is to do it right from the start. Logic must be:

•Correct by Design—Each gate, each line of VHDL, or each line of Verilog must be understood completely. Don't hope that simulation will find the bugs because you may neglect to test a part of the design. If it was designed sloppily, it will fail.

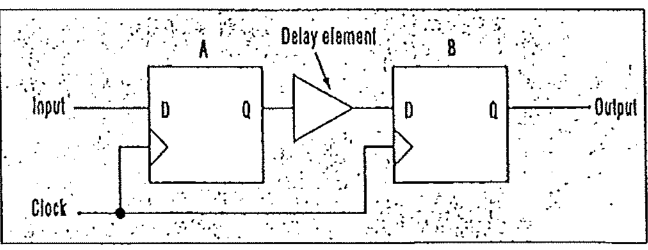
•Correct by Inspection—Disciplined layout also will make your design more robust, comprehensible, and maintainable. It should not be necessary to sort through a mass of ugly code or spaghetti gates to understand the operation of the function. Organized gates, commented code, and thorough accompanying documentation will provide a basis for a reliable design.

Peter Chambers is an Engineering Fellow in the Computer Products Division at VLSI Technology, Tempe, Ariz. He holds a BSc degree from the University of Exeter, England, and an MS from Arizona State University.

HOW VALUABLE	CIRCLE
HIGHLY	540
MODERATELY	541
SLIGHTLY	542



9. A simple circuit with two flip-flops shows the classic race condition in which the switching of flip-flop A violates the hold time of flip-flop B. If B's clock lags A's clock, then B's output may be incorrect.



10. The fast-path problem is easily solved with a delay element. Many synthesizers will insert the delays automatically, but they'll only work if the constraints are set up correctly.

HOGESCHOOL VOOR WETENSCHAP & KUNST | DE NAYER INSTITUUT
SINT-KATELIJNE-WAVER

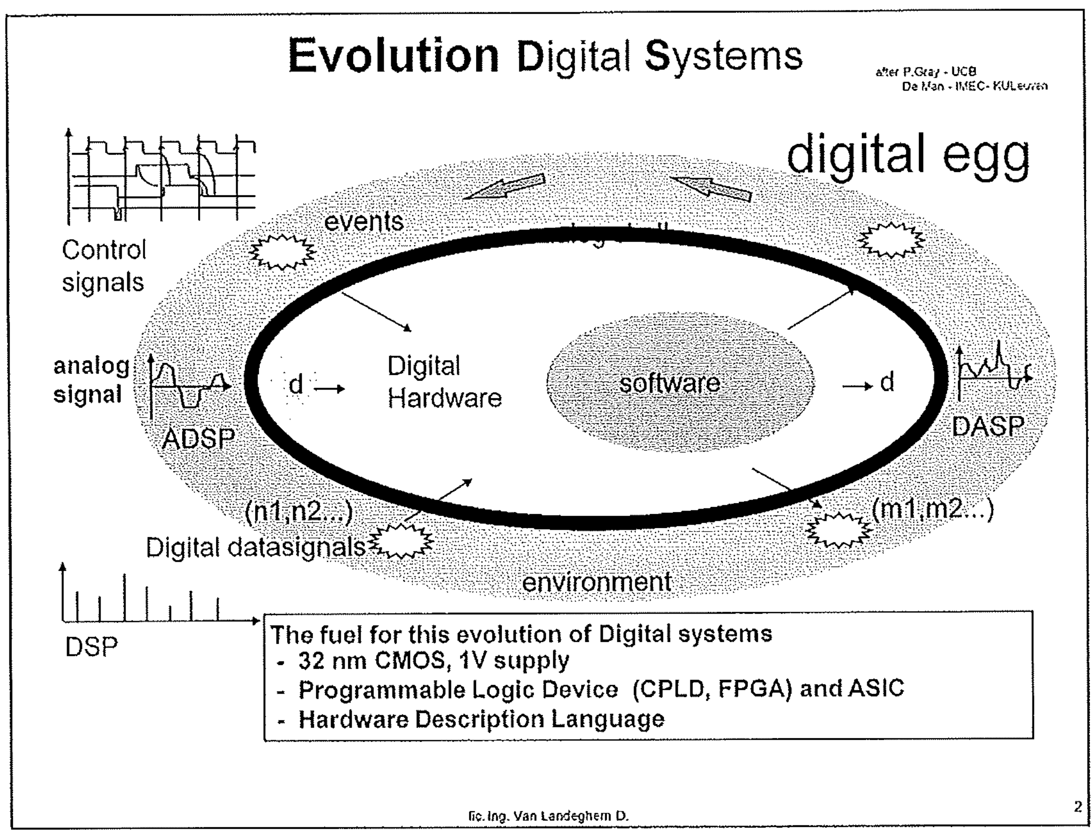
Digitale Elektronica

VHDL: Hardware Description Language

lic. ing. D. Van Landeghem

revision: ir. J. Meel

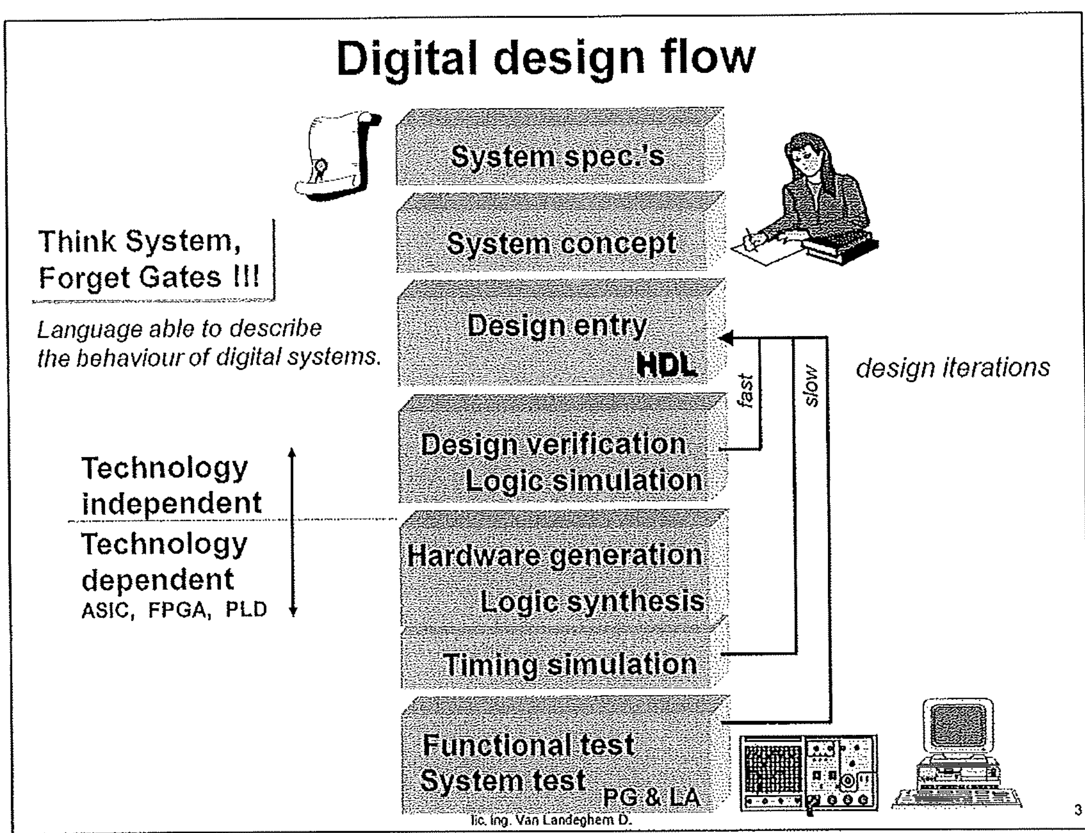
feb 2010



In the first half of the 1990s, the electronics industry experienced an explosion in the demand for personal computers, cellular telephones and high-speed data communications devices. Vendors built products with increasingly greater functionality, higher performance, lower cost, lower power consumption and smaller dimensions. To do this, vendors created highly integrated, complex systems with fewer ICs and less PCB area. This situation fostered the need for widespread adoption of modern methodologies in design and test. Both high density programmable logic devices (PLDs) and VHDL, the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language, became key elements in these methodologies.

High density programmable logic devices, including complex PLDs (CPLDs) and Field Programmable Gate Arrays (FPGAs), can be used to integrate large amounts of logic in a single IC. Semicustom and full-custom Application Specific Integrated Circuits (ASICs) are also used for integrating large amounts of digital logic, but CPLDs and FPGAs provide additional flexibility. They can be used with tighter schedules, for low volumes, for first production runs and even for high volumes.

VHDL provides high-level language constructs that enable designers to describe large circuits and bring products to market rapidly. It supports the creation of design libraries in which to store components for reuse in subsequent designs. Because it is a standard language (IEEE standard 1076), VHDL provides portability of code between synthesis and simulation tools, as well as device independent design. It also facilitates converting a design from a programmable logic to an ASIC implementation.



Ideally, the design process will be performed top down. This means that the circuits function is specified typically by text, plus constraints on cost, performance, and reliability.

The circuit is then repeatedly divided into blocks as necessary to complete the design. In order to obtain reusability and to make maximum use of predefined modules, it is often necessary to perform portions of the design bottom up. In addition, a particular circuit design for a given specification may violate one of the constraints in the initial specification. In this case, it is necessary to backtrack upward through the hierarchy until a level is reached at which the violation can be eliminated.

In modern design, Hardware Description Languages have become essential to the design process. HDL's have a resemblance to programming languages, but are specifically oriented to describing hardware structures and behaviour. They differ markedly from the typical programming language in that they represent extensive parallel operation whereas most programming languages represent serial operation. An obvious use for a HDL is to provide an alternative to schematic entry of the circuit design

Logic synthesis

A major reason for the growth of the use of HDLs is *logic synthesis*. A logic synthesis tool with an accompanying library of components can convert a HDL description into an interconnection of primitive components that implements the circuit in the target technology (CPLD, FPGA, ASIC).

Simulation

With simulation the behaviour of the design can be predicted for given inputs. All the hardware descriptions, no matter what hierarchy level they are, can be simulated. Since they represent the same system, in terms of functionality, but not necessarily timing, they should respond by giving the same results for the same applied inputs. This simulation property supports design verification and is a principal reason for the use of HDL's.

- Logic simulation approximates the hardware behaviour by assuming that logic modules do not have a delay.

- Timing simulation predicts the exact behaviour (based on the delay of building blocks).

Think System, Forget Gates !!!

- Design entry by Hardware Description Language
- Verification of function and timing spec's in the **HDL** (Simulation of the description)
- Automatic hardware generation (Logic Synthesis)
- Verification of function performance (Simulation of the hardware)

- Time to design decrease
  *** HDL (entry/simulation)**
  *** Logic Synthesis**

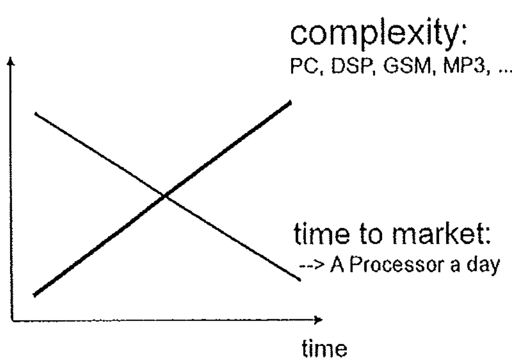
- Time to process decrease
  *** CPLD's**
  *** FPGA's**

lic. ing. Van Landeghem D. 4

Using a HDL, makes it possible to create a device-independent design. The designer does not need to know the details the internal architecture of the device in order to generally optimize the function of his design. Instead, he can concentrate on the functionality of his design. Resource utilization or performance still is dependent on the used architecture of the devices.

During the design entry en optimization, the simulation of the design is done on the HDL description in stead of on the synthesized design. Simulating a complex design description before synthesis can save considerable time. A design bug discovered at this stage can be corrected before the design implementation stage.

- Size and complexity of digital systems increase
- Communication among designers and designs
- Part obsolescence
  - * Design takes-too much time
  - * Time of use can be small
  - * Parts obsolete before system is ready



lic. ing. Van Landeghem D. 5

Every design engineer in the electronics industry should use a hardware description language to keep pace with the productivity of competitors. With VHDL, the time to describe the functionality of the design is reduced. Synthesis is done automatically.

VHDL has powerful language constructs with which to write succinct code descriptions of complex control logic.

VHDL makes it possible to create a design without having to first choose a device for implementation. With one design description, many device architectures can be targeted.

Because VHDL is a standard, a design description can be taken from one synthesis tool to another, and one platform to another. This means that VHDL design descriptions can be used in multiple projects, and that designer skills in using one EDA tool are useful with another.

A VHDL description can first be used synthesize the design for a CPLD or FPGA generates, to hit the market quickly. When production volumes reach appropriate levels, VHDL facilitates the development of an ASIC. Sometimes, the exact code used with the FPGA can be used to generate the layout of the ASIC. VHDL and programmable logic pair well together to facilitate a speedy design process. They bring products to market in record time.

Hardware Description Language

VHDL: VHSIC (Very High Speed Integrated Circuit) HDL

IEEE Standard

- IEEE Std 1076-1987 (first standard) → bit, bit_vector (2-value)
- IEEE Std 1076-1993 (extensions)
- ...
- IEEE Std 1076.6-1999 (Register Transfer Language [RTL] synthesis)
- ...
- IEEE Std 1076-2008 (productivity enhancements)

  → **VHDL: Verification HDL** (ABV = Assertion Based Verification)

- IEEE Std 1164 (hardware related types and conversion functions)
  → std_ulogic, std_ulogic_vector (9-value)

Verilog HDL

IEEE Standard 1364 - 1995

lic. Ing. Van Landeghem D. 6

VHDL is a programming language that has been designed and optimized for describing the behavior of digital systems.

VHDL is a standard language. This virtually guarantees that design descriptions can be reused.

VHDL, which stands for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, was developed in the early 1980s as a spin-off of a high-speed integrated circuit research project funded by the U.S. Department of Defense. Researchers were confronted with the daunting task of describing circuits of enormous scale and of managing very large circuit design problems that involved multiple teams of engineers. With only gate-level design tools available, it soon became clear that better, more structured design methods and tools would be needed.

The first publicly available version of VHDL, version 7.2, was released in 1985. In 1986, the Institute of Electrical and Electronics Engineers, Inc. (IEEE) was presented with a proposal to standardize the language, which it did in 1987 after substantial enhancements and modifications were made by a team of commercial, government and academic representatives. The resulting standard, IEEE 1076-1987, is the basis for virtually every simulation and synthesis product sold today. An enhanced and updated version of the language, IEEE 1076-1993, was released in 1994, and VHDL tool vendors have been responding by adding these new language features to their products.

To get around the problem of nonstandard data types, another standard was developed by an IEEE committee. This standard, numbered 1164, defines a standard package (a VHDL feature that allows commonly used declarations to be collected into an external library) containing definitions for a standard nine-valued data type. This standard data type is called std_logic.

Hardware abstraction levels
(architecture styles)

- Behavioral

It describes the **function** of the design in a **software-like** procedural form, and provides **no detail** as to how the design is **to be implemented.**

- Dataflow

Is a **concurrent representation** of the **flow of control** and **movement of data.**
This imply an architecture and a **unique hardware.**

- Structural

Is the **lowest** and **most detailed level** of description.
It's the hardware description at **gate-level (netlist).**

lic. ing. Van Landeghem D. 7

Entity and architecture

If the entity declaration is viewed as the engineer's 'black box', for which the inputs and outputs are known but the details of what is inside the box are not, then the architecture body is the internal view of the black box. Every architecture body is associated with an entity declaration. An architecture describes the contents of an entity.

Architecture style

VHDL allows the designer to write designs using various styles of architecture, and to mix these styles. The styles are behavioral, dataflow and structural descriptions, or a combination thereof. These styles allow the designer to describe a design at different levels of abstraction, from algorithmic level to gate level primitives. The name given to an architecture style is not important. However, the terms will give us a common vocabulary.

Different design descriptions (styles) however produce different, but functionally equivalent, design equations resulting in different circuit implementations.

- *Behavioral style*

  Describes a system in terms of what it does (or how it behaves)

  Process statements and procedures with sequential statements (IF, CASE, FOR, ...)

- *Dataflow style*

  Specifies the relationship between the input and output signals

  Concurrent signal assignments with operators (and, or, not, xor, ...)

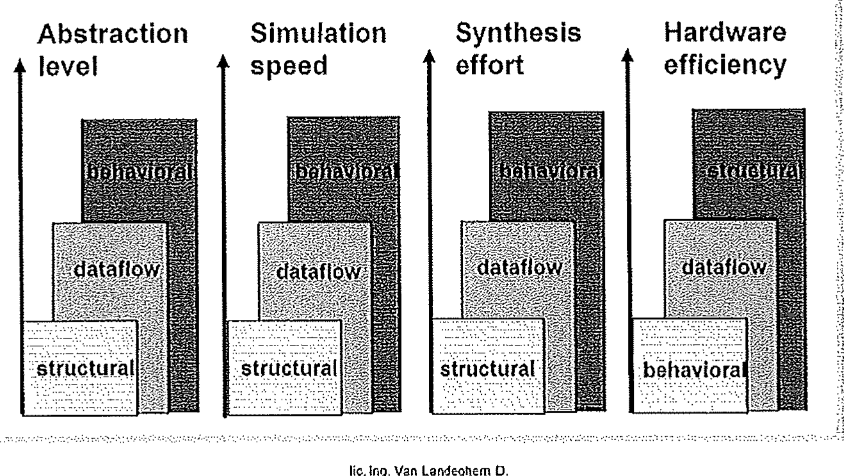
- *Structural style*

  Describes a system as an interconnection of predefined components.

  Hierarchical design: modules & interconnections (Component, Port Map, Generate, ...)

The Synthesizer

- The synthesizer is used for automatic hardware generation.
- Descriptions of hardware can be classified by the abstraction levels of hardware at ***behavioral, dataflow or structural*** levels.
- The consequences of abstraction levels:

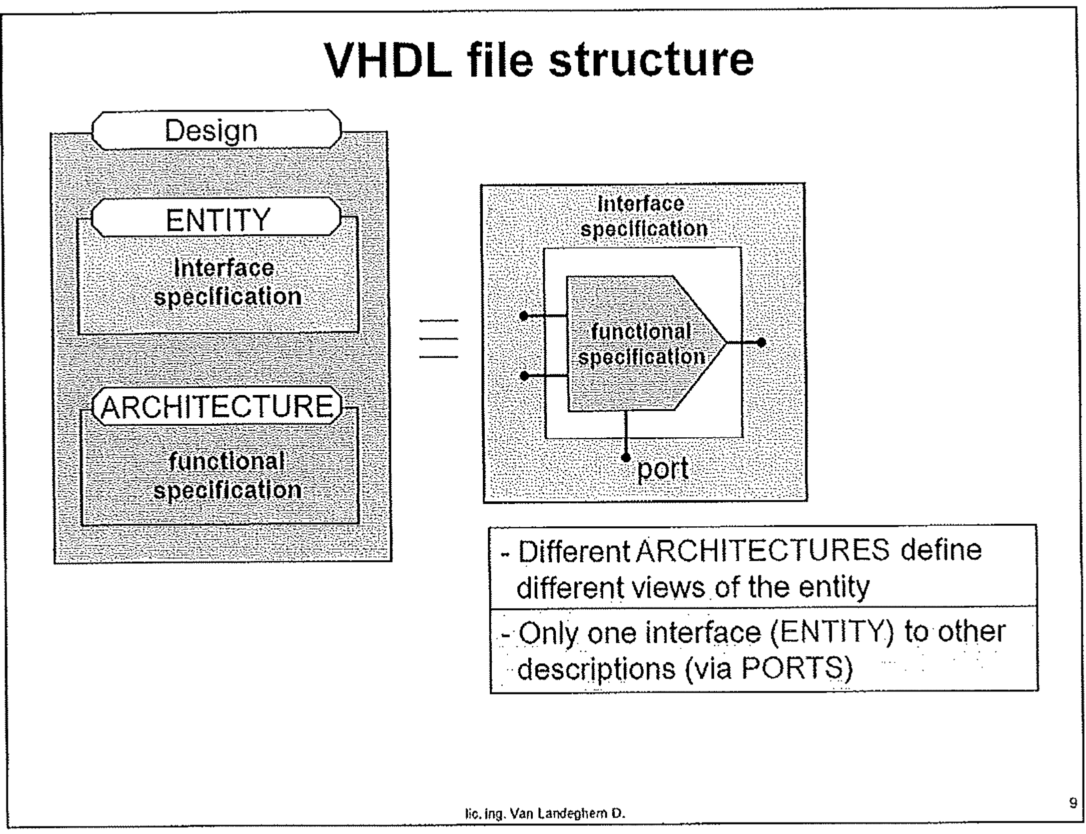


lic. Ing. Van Landeghem D. 8

Different design descriptions of hardware abstraction levels can produce different, but functionally equivalent, design equations resulting in different circuit implementations. Fortunately, for simple design descriptions, almost any description style will most assuredly be realized with the same device resources. However, this is not true of more complex design descriptions. Synthesis software must interpret complex design descriptions and attempt to minimize the logic of the circuit. The synthesis software and fitter- or place and route software must then determine how to implement that logic to make the best use of available resources in the target logic device (CPLD, FPGA). A VHDL description of the behavioral style requires mature synthesis tools to generate efficient hardware. These synthesis tools will have built-in algorithms that will find the optimal, or near optimal solution, regardless of the form of the description. VHDL descriptions, that more closely resembles a netlist of components representative of device resources, 'can' synthesize to a more optimal implementation. The synthesis tools have nowadays a high performance.

Using a behavioral design description (lacking the implementation details), will reduce the design time. This is because the designer doesn't have to go into circuit details. The required simulation time of a behavioral description is also short. The simulation of an algorithm (behavioral description) is much faster then a structural circuit description because of the volume of events, signals, variables etc. that must be calculated during the simulation. The simulation software as a second advantage will be executed often to verify the functionality of the design based on iteration.

The only disadvantage using a behavioral description is the efficiency of the hardware result, but this will be compensated by superior time to market.



Even the most basic VHDL model has two parts: an entity and an architecture.

Entity

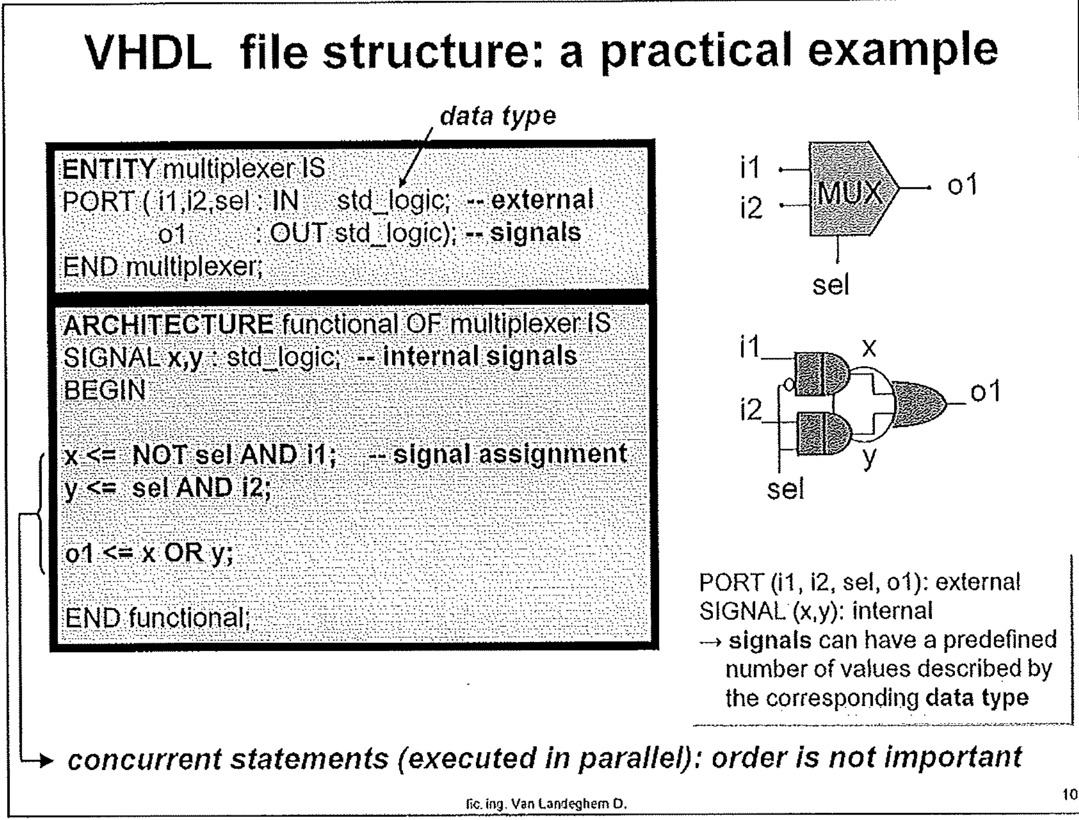
An entity is gives a simple declaration of the module's inputs and outputs. The entity part describes a black box. It is an abstraction of a design that could represent a complete system, board, chip, small function or logic gate. An entity describes the name of the module, the inputs and outputs, but nothing is known of the functionality of the internals of the circuit.

Ports are user-defined identifiers to name external-interface signal (input and outputs).

Architecture

An architecture is a detailed description of the module's internal structure or behavior. So, the architecture body describes the functionality of the design entity. It can contain any combination of behavioral, structural or dataflow descriptions (= level of abstraction of the description) to define an entity's function.

The reason for having this split is that it is possible to have more than one architecture for an entity and perhaps describing alternative implementations or different levels of description.



Signals

The *ports* i1, i2, sel (IN) and o1 (OUT) are the signals by which the entity communicates with its external environment.

(Note: In the PORT declaration, the semi-colon is used as separator. No semi-colon after the last port declaration).

The *internal signals* x and y are used to describe the behavior of the multiplexer in the architecture part. They cannot be seen by the external environment.

Each signal has a predetermined *data type* which limits the amount op possible values for this signal. Synthesizable data types, like std_logic, offer only a limited number of values (std_logic: 9 values).

Signals represent wires and storage elements.

Description of functionality

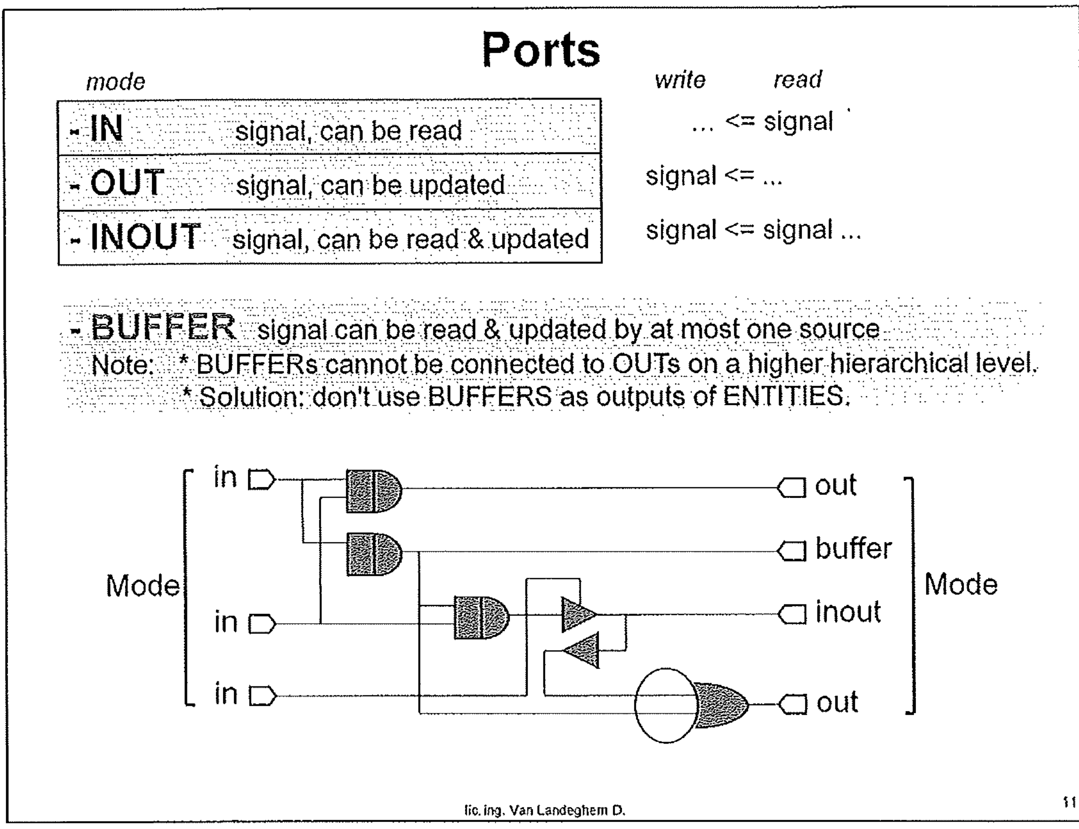
Hardware is concurrent or parallel, and executes simultaneously. It can be modeled by a series of concurrent statements that define how outputs react to inputs.

The concurrent signal assignment:

```
o1 <= x OR y;
```

means: on each moment o1 takes the value of the logic OR of signals x and y.

The order of the concurrent statements is not important, they are executed in parallel.



The entity contains a port declaration. Each declared port (pin in a schematic symbol) must have a name (self-explanatory), a mode (direction) and a data type. The mode can be one of 4 values:

• IN: Data flows only into the entity. The driver for a port of mode in is external to the entity. Mode in is used primarily for clock inputs, control inputs (like load, reset and enable) and unidirectional data inputs.

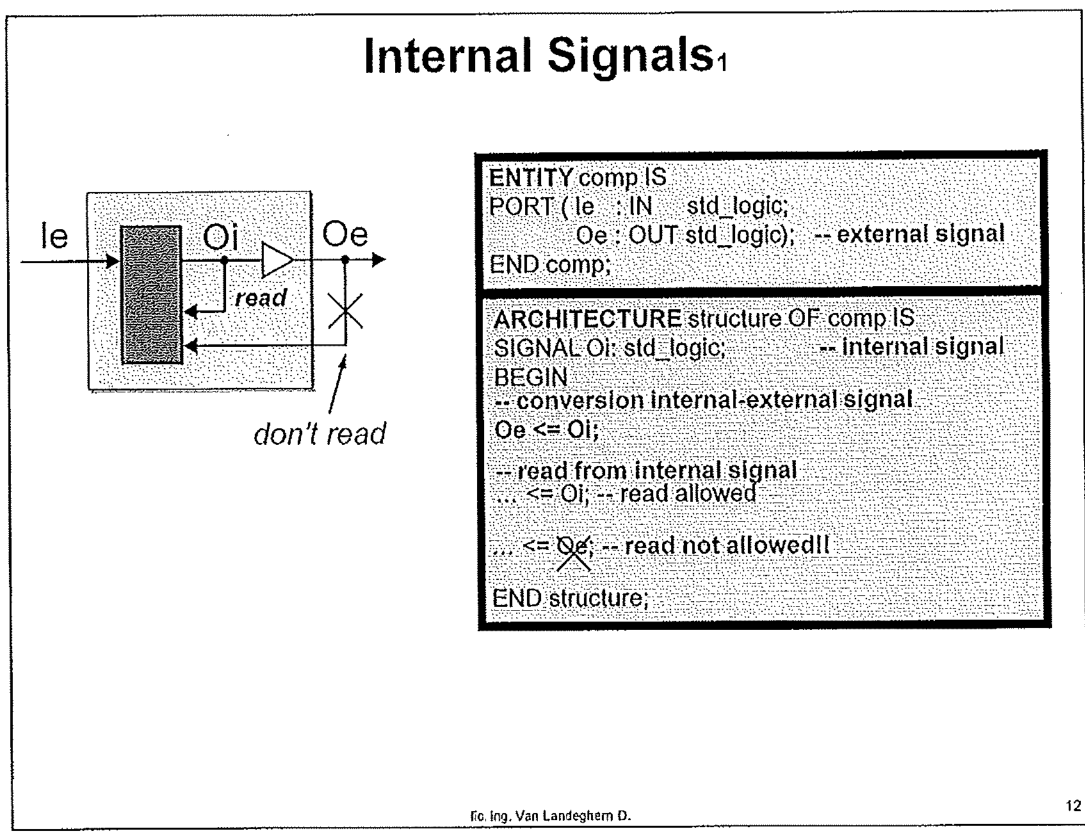
• OUT: Data flows only from its source to the output port of the entity. The driver for a port of mode out is inside the entity. Mode out does not allow for feedback because such a port is not considered readable within the entity.

Mode out is used for outputs such as a terminal count output (a terminal count is asserted when the value of a counter reaches a predefined value).

• BUFFER: For internal feedback (that is, to use a port also as a driver within the architecture), the port can be declared as mode buffer. More often a separate signal is declared within the architecture body (an internal signal). A port that is declared as mode buffer is similar to a port that is declared as mode out, except that it does allow for internal feedback. Mode buffer does not allow for bidirectional ports because it does not permit the port to be driven from outside of the entity. Attention: (1) a port of mode buffer may not be multiple driven. (2) It may only be connected to an internal signal or to a port of mode buffer of another entity (not to out or inout of another entity). Therefore it is not recommended to use buffer.

• INOUT: For bidirectional signals, a port must be declared as mode inout, which allows data to flow into or out of the entity. In other words, the signal driver can be inside or outside of the entity. Mode inout also allows for internal feedback. Mode inout can replace any of the other modes. Although using only mode inout for all ports would be legal, it would reduce the readability of code, making it difficult to discern the source of signals. A more appropriate use for mode inout is for ports that are truly bidirectional, such as the data bus of a processor.

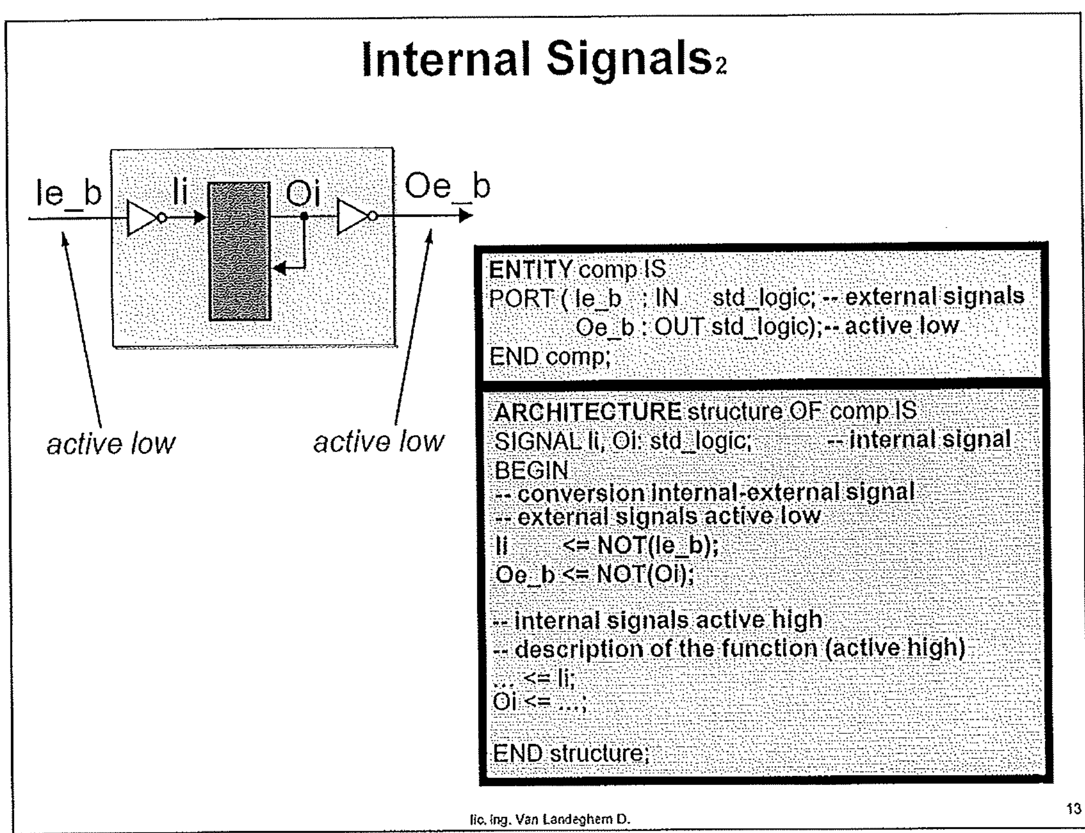
The port modes most commonly used are: IN and OUT.



An output port (OUT) of an entity can only be driven. The driver for a port of mode out is inside the entity. Data flows from its source to the output port of the entity. Data cannot be read from an output port!

Solution

Internal signals can be driven en read. Define an internal signal that copies its value to the output port. From this internal signal the value can be read.



A description of a logic function or system is often done with active high signals. The input coming from a switch or the output driving a LED is often active low.

Solution

Internal signals can be used the describe the functionality active high. A conversion from/to the active low input/output ports can be done with concurrent assignments (as a shell around the active high functional description).

Standard data types

- The **PACKAGE standard (library STD)** contains standard data types.

- Example :

```
PACKAGE standard IS
    TYPE boolean     IS (false, true);
    TYPE bit         IS ('0', '1');
    TYPE bit_vector  IS ARRAY (NATURAL RANGE <>) OF bit;
    TYPE character   IS (  ...  );
```

bit values enclosed in '.'
vector values enclosed in "..."

- **Type BIT** is built-in in standard VHDL.
  It is an enumeration type, all the possible values are enumerated.

- **Array type** compatibility : Array values must be of the correct base to be assigned.

Example :

```
SIGNAL a,b : bit_vector(7 DOWNTO 0);
SIGNAL c   : bit_vector(5 DOWNTO 0);
SIGNAL d   : bit_vector(0 TO 7);

a <= b;
a(5 DOWNTO 0) <= c;
a <= d;
a(3) <= c(5);
```

vector = collection of signals of the same type

a(7) ← d(0)
...
a(0) ← d(7)

assignment according to position, not number

- **Integer** is a built-in type. Ranges of integers can be specified as a new type:

```
TYPE test_integer IS RANGE -5 TO 4;
```

VHDL = strongly-typed language *(only explicit type conversions)*

lic. ing. Van Landeghem D. 14

In addition to specifying identifiers and modes for ports, also the data types for ports must be declared.

The types provided by the IEEE 1076-1993 standard that are most useful and well-supported and that are applicable to synthesis are the data types boolean, bit, bit_vector and integer.

The definitions of the standard data types are contained in the PACKAGE standard.

VHDL is a strongly-typed language:

- Explicit type conversion is supported
- Implicit type conversion is not supported

Libraries & Packages

- A **LIBRARY** clause is used to define the library the user wants to have access to.

example : LIBRARY ieee;

-The **LIBRARY work** contains compiled objects (entities, architectures, packages, ...), and is (like the library standard) **always accessible.**

-The **USE** clauses are used to define the **part of a PACKAGE within a LIBRARY** that the user wants to have **access to.**

```
example : LIBRARY ieee;
          USE ieee.std_logic_1164.ALL;     -- everything in the package std_logic_1164
          USE ieee.arith_1164.multiply;    -- only function multiply in package arith_1164
```

- The scope of libraries and use clauses :

USE clause makes a PACKAGE or part of a package **visible** to the end of the enclosing declarative region.

It is necessary to **repeat** the library and USE statement **for each ENTITY and PACKAGE.**
Even if they are in the same file !

lic. ing. Van Landeghem D. 15

Library

Libraries provide a set of packages, components, functions, data types that simplify the task of designing hardware.

The LIBRARY work is the current working library that contains all compiled objects (entities, architectures, packages, ...), and is (like the library standard) always accessible. It must not be declared.

Package

A package is a collection of definitions of related data types, constants, subprograms, ... that serve a common purpose.

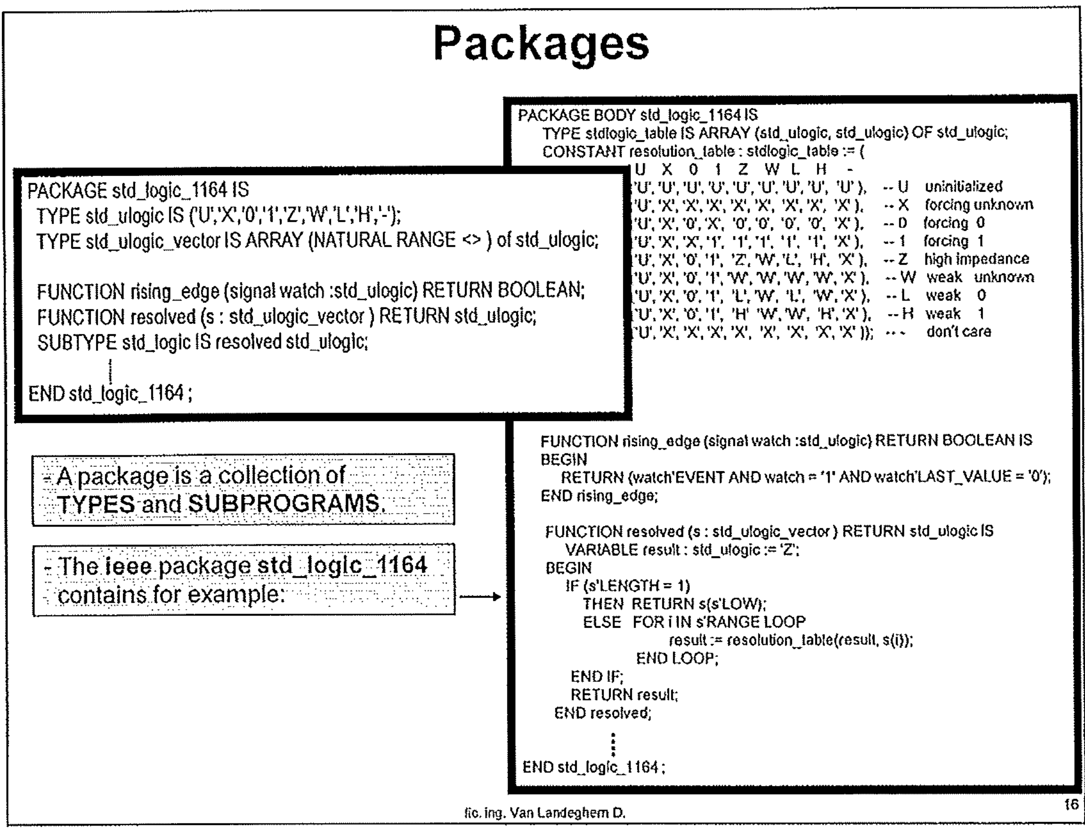
A package is used for declarations that shall be shared by several design units.

Std_logic_1164

The most useful and well-supported types for synthesis provided by the IEEE std_logic_1164 package are the types std_ulogic, std_logic and arrays of these types. As the names imply, 'standard logic' is intended to be a standard type used to describe circuits for logic synthesis and simulation. For simulation and synthesis tools to process these types, their declarations must be made visible to the entity by way of LIBRARY and USE clauses.

The use of the ieee library and its std_logic_1164 package:

```
LIBRARY ieee;
USE ieee.st_logic_1164.ALL
```

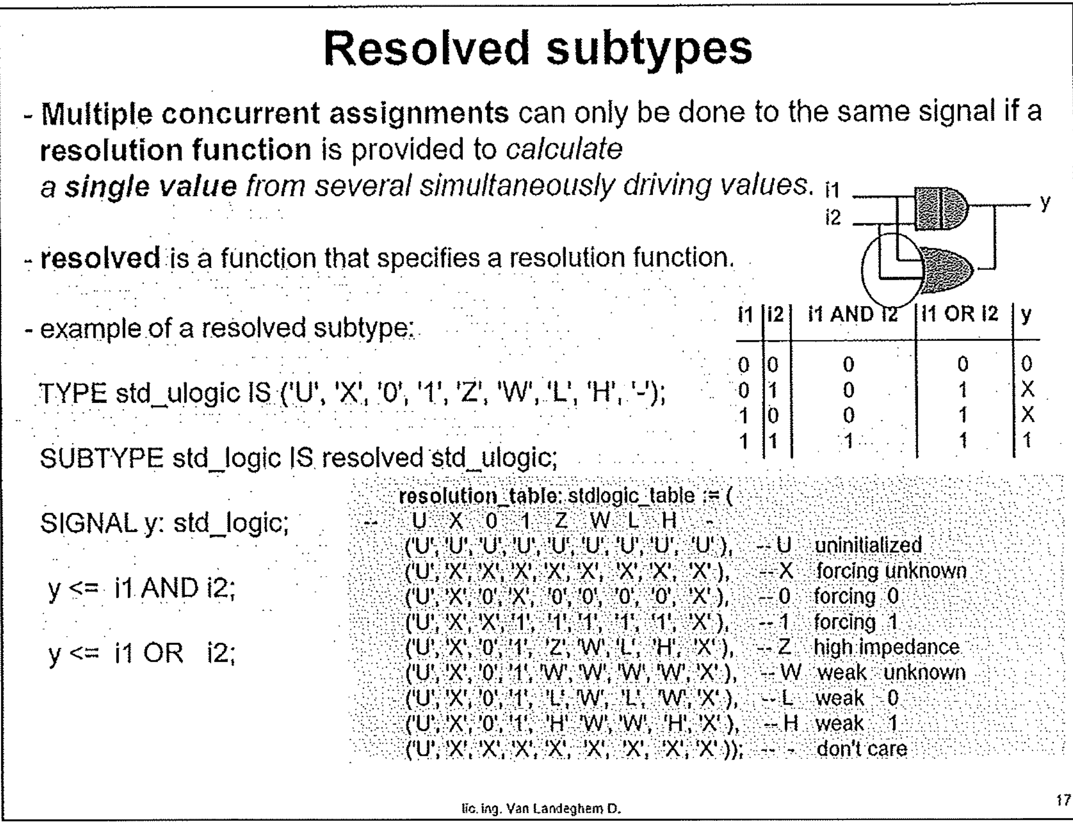


The most useful and well-supported types for synthesis provided by the IEEE std_logic_1164 package are the types std_ulogic, std_logic and arrays of these types. As the names imply, « standard logic » is intended to be a standard type used to describe circuits for synthesis and simulation. The type std_ulogic defines a 9-value ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-') logic system. The subtype std_logic has std_ulogic as its base type, and the set of values for std_logic is the same.

The values '0', '1', 'L' and 'H' are logic values that are supported by synthesis. The values 'Z' and '-' are also supported by synthesis for three-state drivers and don 't-care values. The values 'U', 'X' and 'W' are not supported by synthesis.

In this course the data type std_logic will be used for most examples. That is because, as its name implies, it is a standard data type useful for synthesis. It is more versatile than bit because it provides the high-impedance value ' Z ' and the don 't-care value '-'.

The IEEE 1164 standard defines arrays of std_ulogics and std_logics as std_ulogic_vector and std_logic_vector.



The subtype std_logic has std_ulogic as its base type, and the set of values for std_logic is the same. But the subtype std_logic also has a *resolution function*, named **resolved**, associated with it.

VHDL does not permit a signal to have more than one driver unless it has an associated resolution function. A resolution function is used to compute a value for a signal based on multiple drivers.

Depending on the current driving values of the drivers (in this case two logic gates), the logic value for y could be '0', '1' or 'X'. In this example, two concurrent signal-assignment statements cause two drivers to be created for y. A resolution function must be associated with the signal y. This function is used to compute a value for y based on the values of the drivers.

Resolved types (like std_logic) use resolution functions to determine the value on a signal when conflicting values are driven on the signal at the same time.

It is not likely that a designer would ever intentionally write code such as the slide example in which the drivers are placed in conflict. But one of the more common applications for which resolution functions are used is *busing*.

In this course std_logic is used for most examples. That is because, as its name implies, it is a standard data type useful for synthesis. It is more versatile than bit because it provides the high-impedance value 'Z' (interesting for simulation of tri-state busses) and the don 't-care value '-' (interesting for synthesis). However, writers of simulation models may find std_logic too restrictive for higher-level modeling. Writers of large system models may avoid using a resolved type in order to increase simulation performance. Use of a resolved type requires a call to the resolution function for each signal assignment, thereby slowing down the simulation.

Predefined Operators

Logical Operators

and	AND
or	OR
nand	NAND
nor	NOR
xor	exclusive OR
xnor	exclusive nor
not	complementation
&	concatenation

Numeric Operators

+	addition / sign
–	subtraction / sign
*	multiplication
/	division
mod	modulo division
rem	modulo remainder
abs	absolute value
**	exponentiation

Relational Operators

=	equal
/=	not equal
<	smaller
<=	smaller equal (+assign)
>	greater
>=	greater equal

Shift Operators

sll n	shift left logical (fill 0)
srl n	shift right logical (fill 0)
sla n	shift left arithmetic (fill LSB)
sra n	shift right arithmetic (fill MSB)
rol n	rotate left (wrap around)
ror n	rotate right (wrap around)

ing. Van Landeghem D. 18

VHDL Operators

Highest precedence first, left to right within same precedence group, use parenthesis to control order. Unary operators take an operand on the right. "result same" means the result is the same as the right operand. Binary operators take an operand on the left and right. "result same" means the result is the same as the left operand.

	operator name	use	result
**	exponentiation	numeric ** integer	numeric
abs	absolute value	**abs** numeric	numeric
not	complement	**not** logic or boolean	same
*	multiplication	numeric * numeric	numeric
/	division	numeric /numeric	numeric
mod	modulo division	integer **mod** integer	integer
rem	modulo remainder	integer **rem** integer	integer
+	unary plus	+ numeric	numeric
–	unary minus	– numeric	numeric
+	addition	numeric + numeric	numeric
–	subtraction	numeric – numeric	numeric
&	concatenation	array & array	array
		element & element	array

Operators: precedence

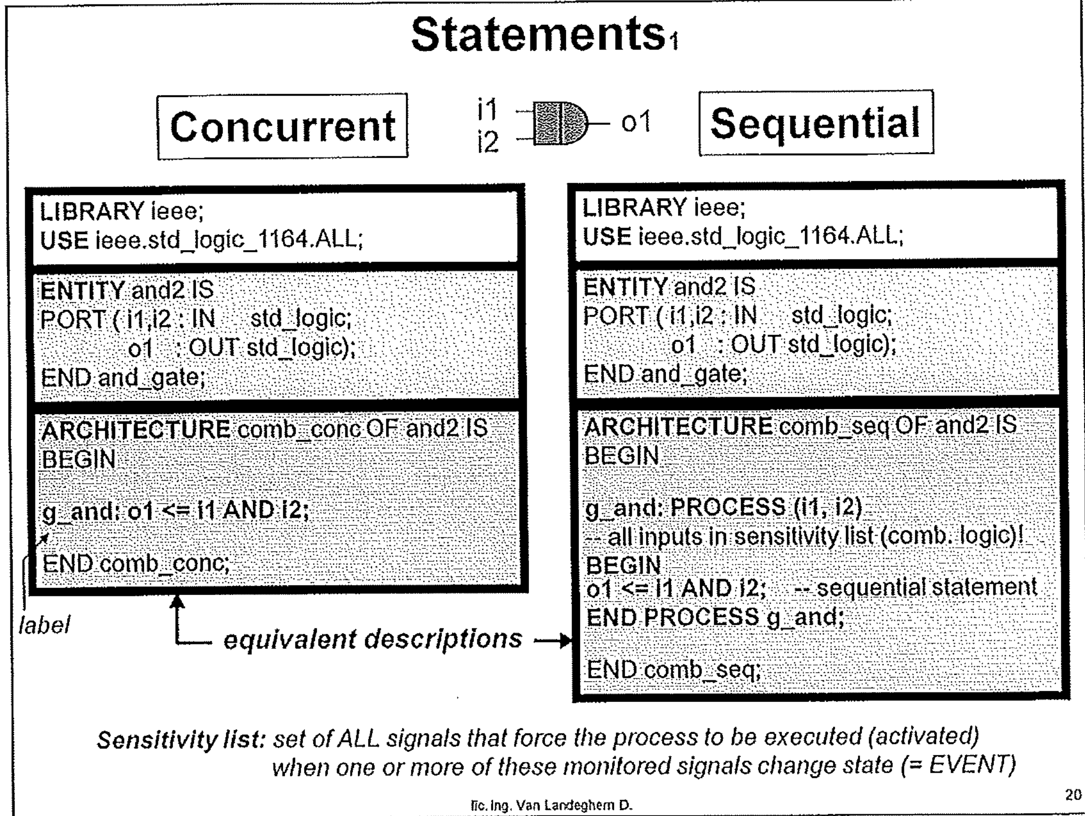
Class	Operators
Miscellaneous	**, abs, not
Multiplying	*, /, mod, rem
Sign	+, –
Adding	+, –, &
Shift	sll, srl, sla, sra, rol, ror
Relational	=, /=, <, <=, >, >=
Logical	and, or, nand, nor, xor, xnor

↑ increasing precedence

*not synthesisable: **, /, mod, rem*

lic. ing. Van Landeghem D. 19

operator name		*use*	*result*
sll	shift left logical,	logical array **sll** integer	same
srl	shift right logical,	logical array **srl** integer	same
sla	shift left arithmetic,	logical array **sla** integer	same
sra	shift right arithmetic,	logical array **sra** integer	same
rol	rotate left,	logical array **rol** integer	same
ror	rotate right,	logical array **ror** integer	same
=	test for equality	all types	boolean
/=	test for inequality	all types	boolean
<	less than	numeric or logical array	boolean
<=	less than or equal	numeric or logical array	boolean
>	greater than	numeric or logical array	boolean
>=	greater than or equal	numeric or logical array	boolean
and	logical and	logical array or boolean	same
or	logical or,	logical array or boolean	same
nand	logical nand	logical array or boolean	same
nor	logical nor	logical array or boolean	same
xor	logical exclusive or	logical array or boolean	same
xnor	logical exclusive nor	logical array or boolean	same



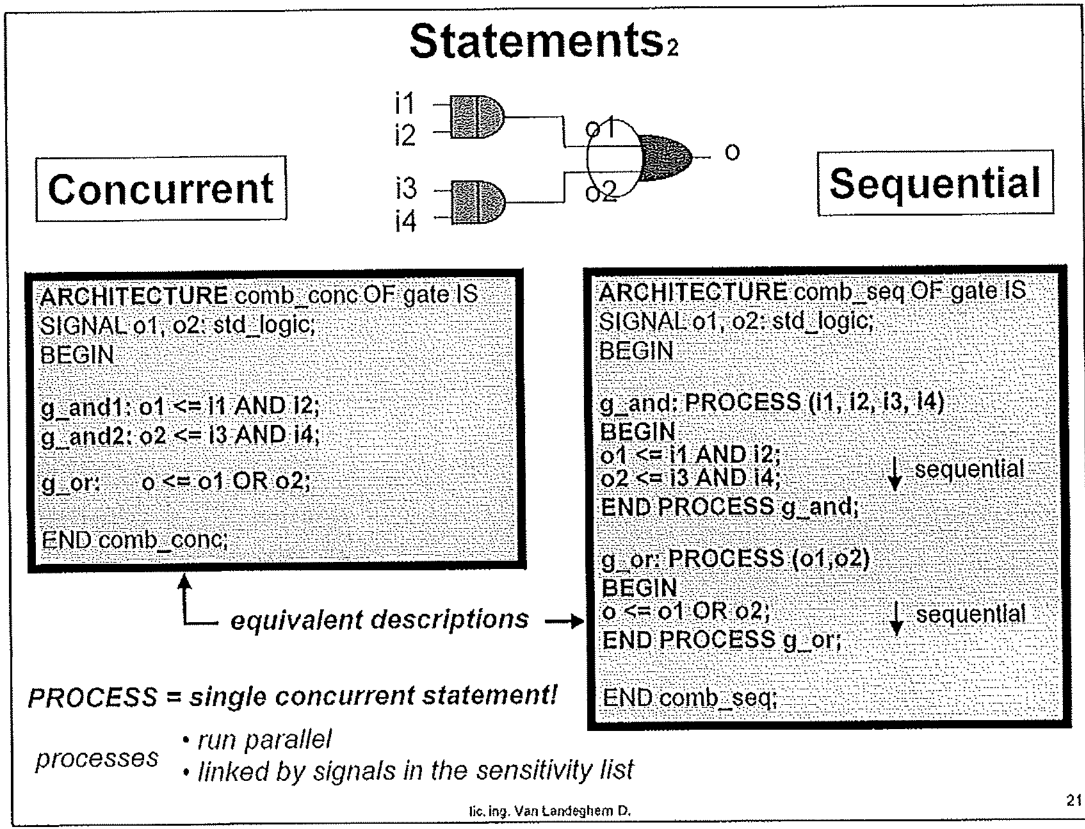
A logic circuit description in an architecture body consists of concurrent statements and sequential statements.

Concurrent statements lie outside of a process. Conceptually, they execute concurrently. Therefore, their order is not important. Concurrent statements are included within architecture definitions and within "block" statements, representing concurrent behavior within the modeled design unit. These statements are executed in an asynchronous manner, with no defined order, modeling the behavior of independent hardware elements within a system.

Sequential statements are those statements contained in a process, function or procedure. A process contains several sequential statements. An independent sequential process represents the behavior of some portion of a design. The body of a process is a list of sequential statements.

Electronic systems are 'concurrent' and so are processes. If a design has multiple processes, then those processes are concurrent with respect to one another and to other concurrent statements within the architecture. The collection of statements that make up a process constitutes a concurrent statement: the process itself. Inside a process, however, signal assignment is sequential from a simulation standpoint and the order in which signal assignments are listed, does affect how logic is synthesized.

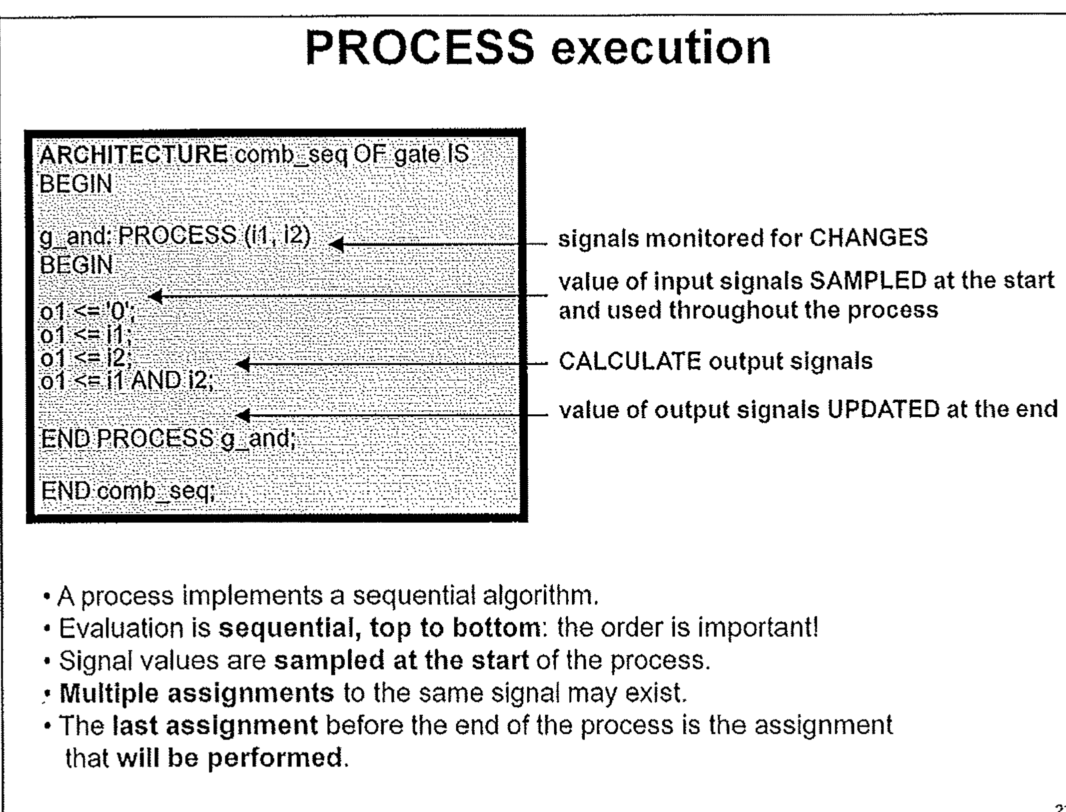
Do not confuse sequential statements with sequential logic (logic with memory or clocked logic)!!!



Each concurrent statement and each set of concurrent statements can be mapped on an equivalent process description with the same functionality.

Concurrent statements are executed in parallel. The order of their definition is not important.

Processes are concurrent statements, while the statements within each process are executed sequentially, i.e. one after another. The order of the definition of the processes is not important, but the order of the statements within a process is important.



• A PROCESS holds a number of sequential statements and executes parallel towards its environment.

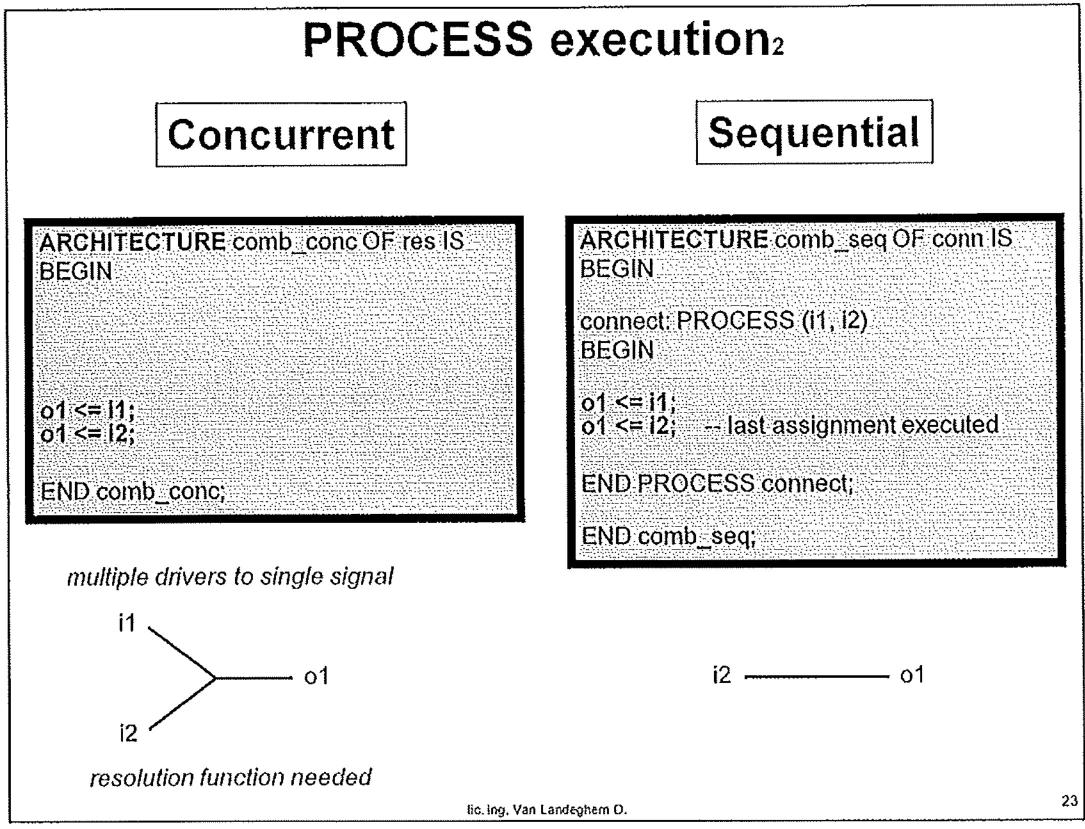
• A PROCESS functions as an eternal loop. It must include at least a *sensitivity list* that specifies the signals that start the execution of the sequential statements of the PROCESS each time value changes (= event) are detected on one of this signals. (A *sensitivity list* is equivalent to a WAIT ON statement placed as the final line in the PROCESS.)

• All sequential statements will execute once at simulation startup in such processes and after that the processes will suspend.

The sequential statements in the process are executed in order, commencing with the beginning of simulation. After the last statement of a process has been executed, the process is repeated from the first statement, and continues to repeat until suspended. If the optional sensitivity list is given, a wait on ... statement is inserted after the last sequential statement, causing the process to be suspended at that point until there is an event on one of the signals in the list, at which time processing resumes with the first statement in the process.

The execution of a process can be broken up in three steps:

1. SAMPLE input signals when a signal in the sensitivity list changes value
2. CALCULATE output signals based upon the sequential statements (order is important)
3. UPDATE the output signals at the end of the process.



Signal driver

. A source which determines a value of each signal

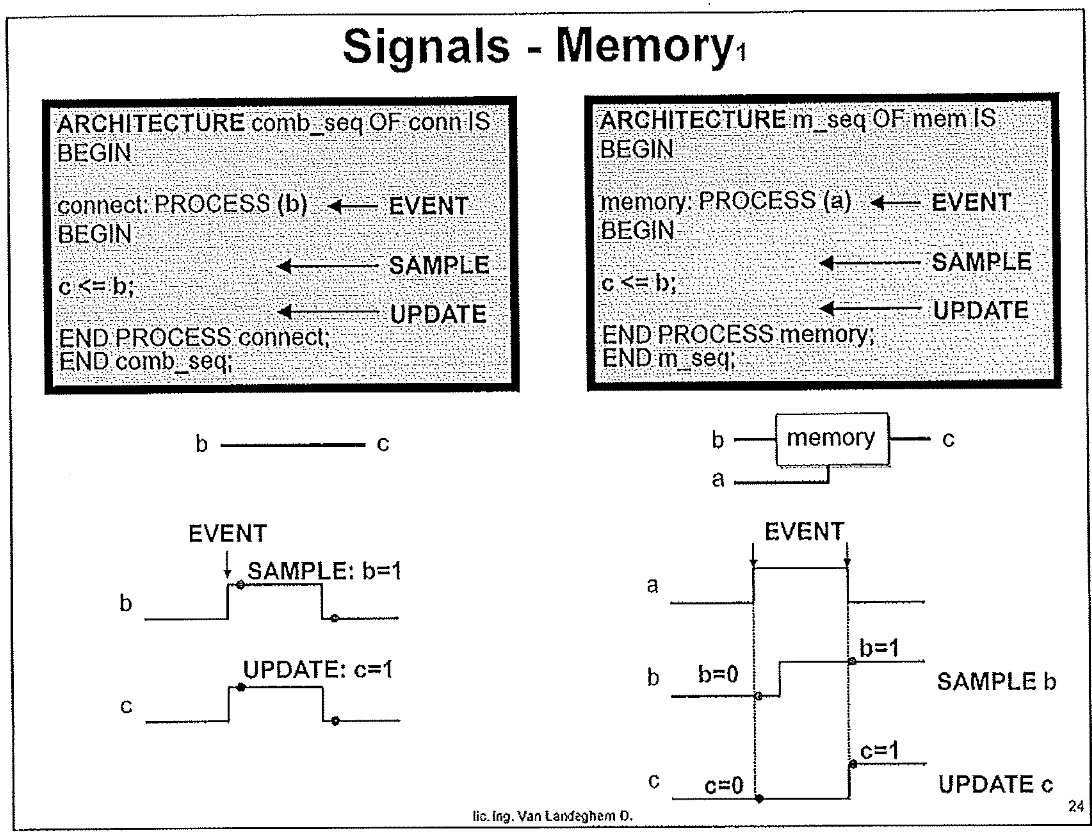
. A signal is updated by the driver at every source update

Multiple driver

. = several signal assignments to a single signal

. This is not allowed in concurrent statements, when no resolution function (example: wired-or bus) is used. Two or more concurrent statements trying to assign a vlaue to the same signal results in conflicts (two sources with different values try to assign their value to the same signal). Processes are also concurrent statements. The assignment of the same signal by two or more processes (when no resolution function is used), is also not allowed.

. Within a process, multiple assignments to the same signal are allowed. The last assignment is executed (sequential execution!).

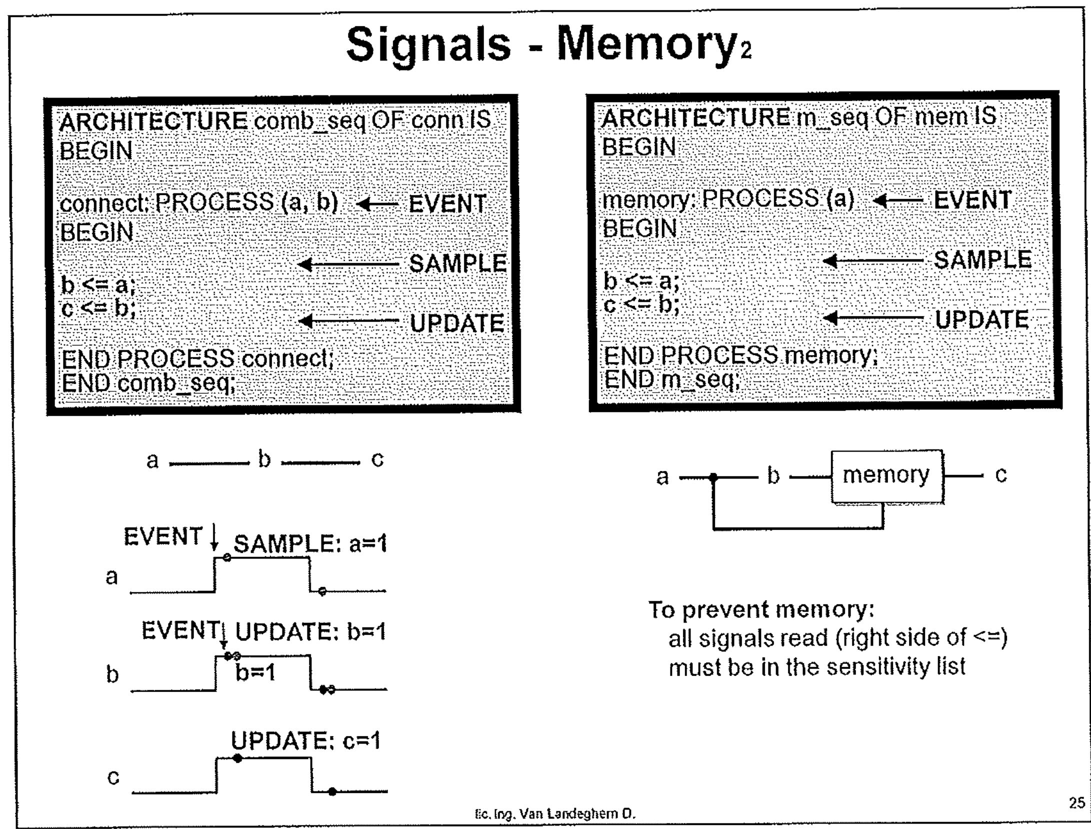


All input signals in the sensitivity list

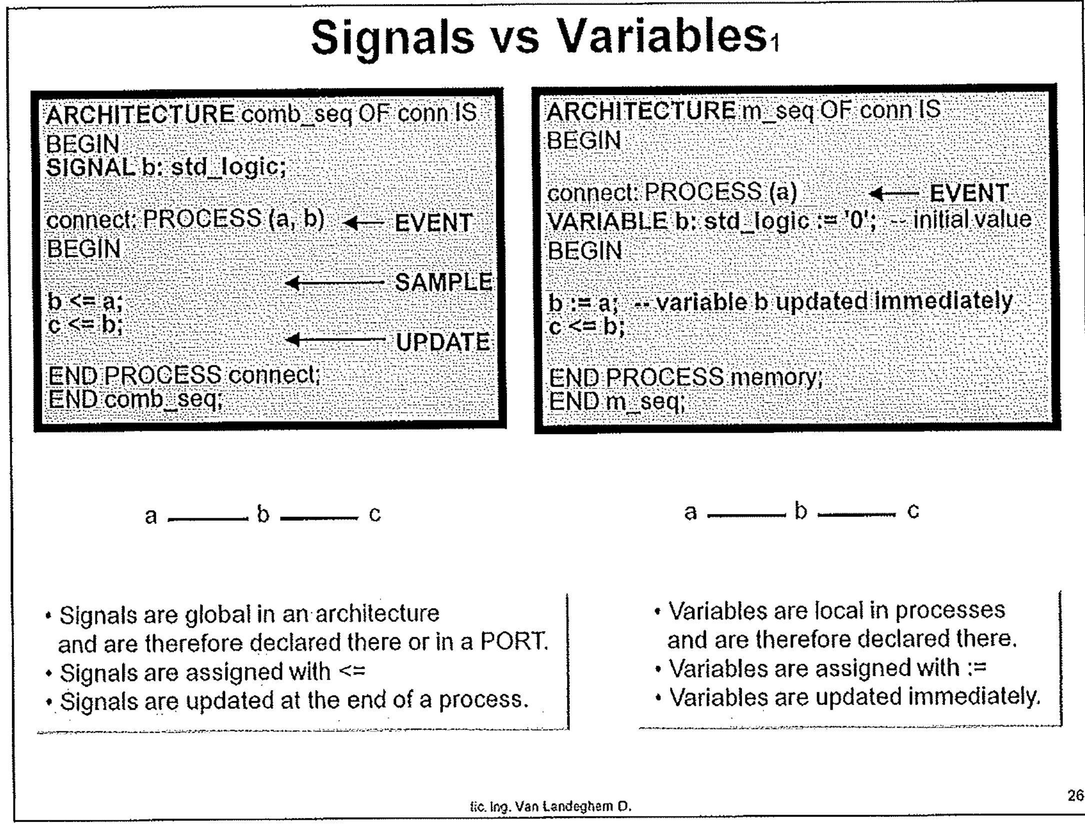
Whenever an input signal changes value, the corresponding outputs are calculated based on the actual inputs at that event. This results in the description of combinatorial logic.

Not all input signals in the sensitivity list

When an input changes that is not included in the sensitivity list, then the output does not change. The previous valued is 'remembered'. The output is only updated when the 'sensitive' input signals change value, as is the case with flip-flops and latches.



To correctly describe combinatorial logic, preventing the synthesis of unwanted latches, it is necessary that all input signals are included in the sensitivity list!!!

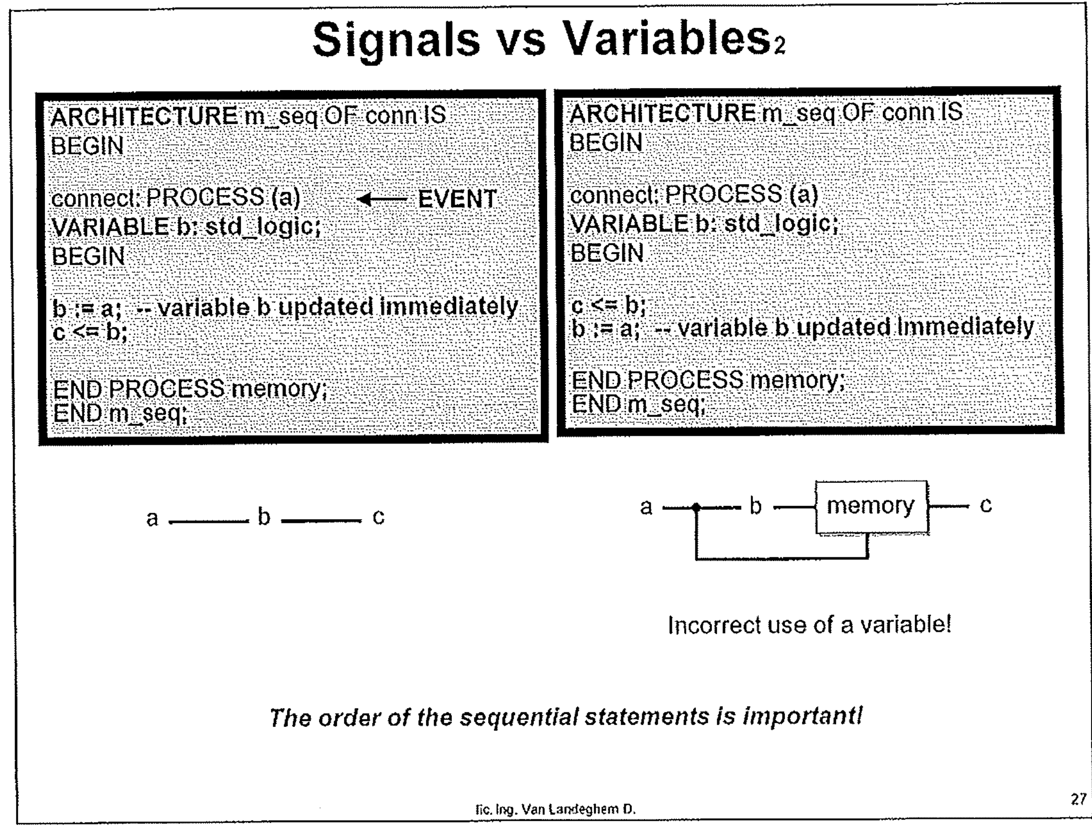


There are two fundamental types of objects used in a VHDL design description: signals and variables. Variables can be used to simplify sequential statements (within processes, procedures and functions), but signals must be used to carry information between concurrent elements of the design (such as between two independent processes).

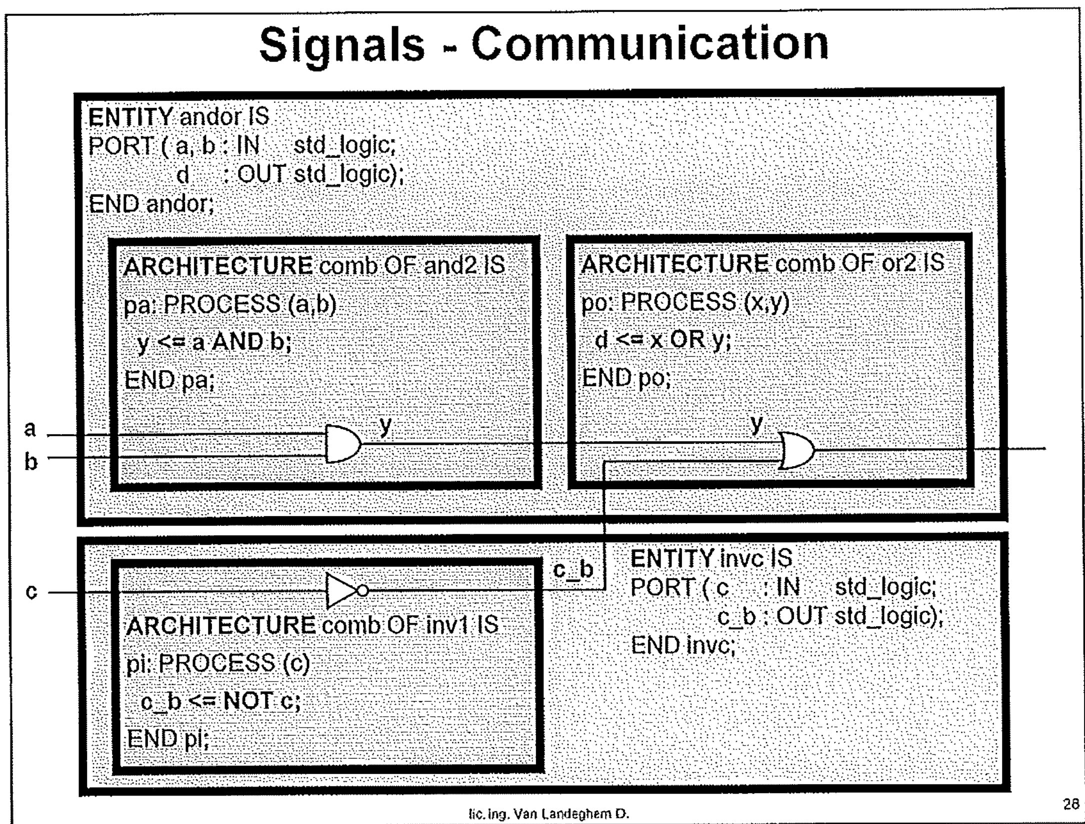
A *signal* is an object with a history of values (related to "event" times, i.e. times at which the signal value changes). Signals are declared via signal declaration statements or entity port definitions, and may be of any data type.

A *variable* is declared within a blocks, process, procedure, or function, and is updated immediately when an assignment statement is executed. A variable can be of any scalar or aggregate data type, and is utilized primarily in behavioral descriptions. It can optionally be assigned initial values (done only once prior to simulation). Variables are sometimes used to hold the results of computations and for the index variables in loops.

Variables can be declared (only) inside the process. Any variable declared can be used only by the code within the process; the *scope* of the variable is limited to the process. To use the value of such a variable outside the process, the variable's value can be assigned to a signal.



When the execution of a process is started, the values of the input signals are sampled. These values cannot be changed during the execution of the process. A variable that is assigned a value is updated immediately. The new value will be used for the 'following' statements. The previous assignments use the previous value of the variable. The order of the sequential statements is therefore important!



All processes of a VHDL design run in parallel, no matter in which entity or hierarchical level they are located. They communicate with each other via signals.

Within an entity internal signals are used for communication between the processes.

When processes from different entities/architectures depend from another, the signals need to be ports of the entities.

Statements3

Concurrent	Sequential
Signal Assignment	Signal Assignment
BLOCK	NULL
PROCESS	FOR...LOOP
...WHEN...ELSE	IF...THEN...ELSE...END
WITH...SELECT...WHEN	CASE...IS
Component Instantiation	NEXT...WHEN
FOR...GENERATE	EXIT...WHEN
GUARDED	RETURN
TRANSPORT	WAIT
ASSERT...REPORT...SEVERITY	ASSERT...REPORT...SEVERITY
Procedure Call	Procedure Call

lic. ing. Van Landeghem D. 29

Concurrent statements

This statements lie outside of a process. Conceptually, they execute concurrently. Therefore, their order is not important. There are three types of concurrent statements used in dataflow descriptions:

. concurrent signal assignment statements with Boolean equations

. selective signal assignment (with-select-when) statements

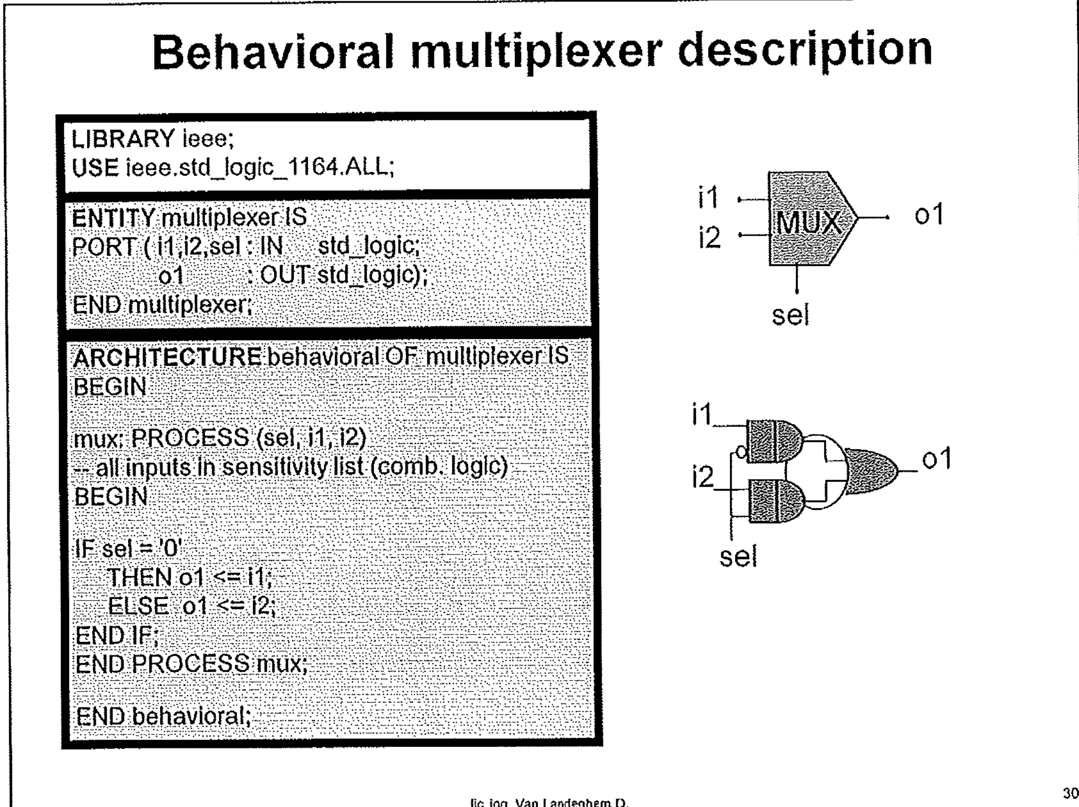
. conditional signal assignment (when-else) statements.

Electronic systems are 'concurrent' and so are processes. If a design has multiple processes, then those processes are concurrent with respect to one another and to other concurrent statements within the architecture. The collection of statements that make up a process constitutes a concurrent statement: the process itself.

Sequential statements

This statements are contained in a process, function or procedure. Inside a process, signal assignment is sequential from a simulation standpoint and the order in which signal assignments are listed does affect how logic is synthesized.

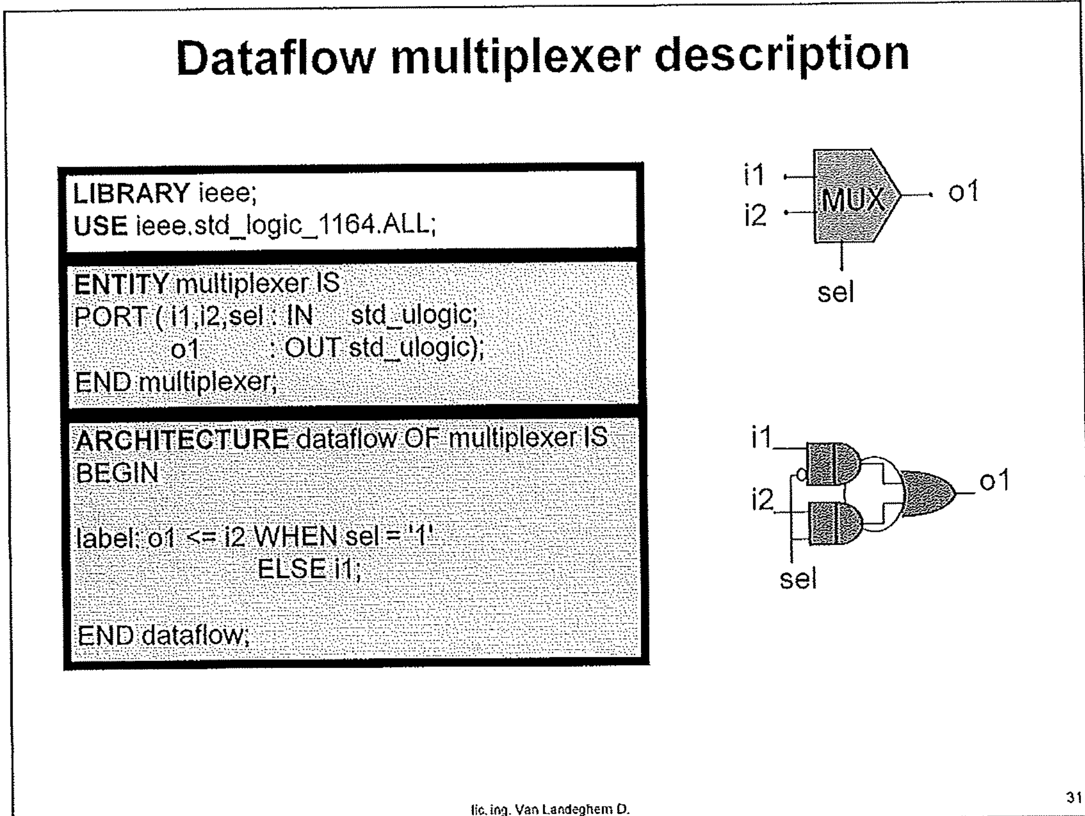
Do not confuse sequential statements with sequential logic (logic with memory or clocked logic)!!!



This is an example of a behavioral description. The architecture is described in an algorithmic way. Behavioral descriptions are sometimes referred to as high-level descriptions, because of their resemblance to high-level programming languages. Rather than specifying the structure or netlist of a circuit, a set of statements is specified that, when executed in sequence, model the function or behavior, of the entity (or part of the entity). The advantage to high-level descriptions is that the designer don 't need to focus on the gate-level implementation of a design. Instead, he can focus on accurately modeling its function.

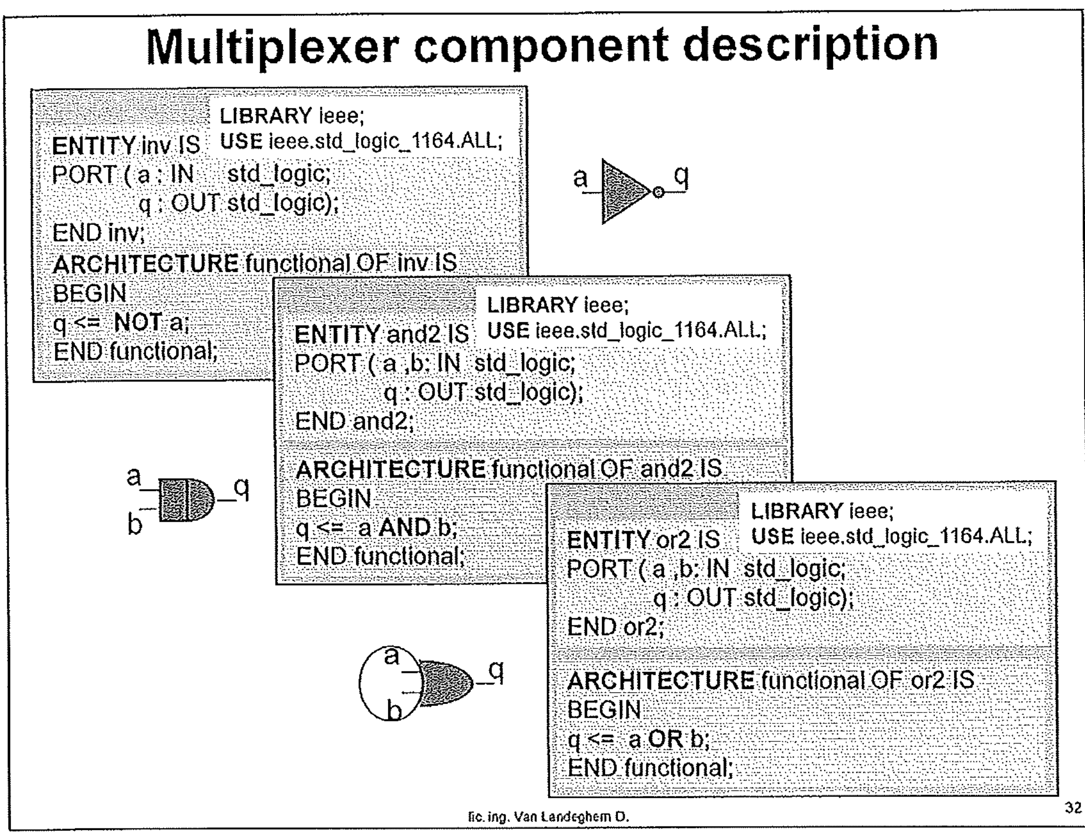
The LIBRARY and USE statements are required to make the std_logic type visible to this entity. The ENTIY part contains the entity declaration for this single bit 2 channel multiplexer. The ARCHITECTURE part contains the architecture body which uses an algorithm to describe the design function. A process statement is one of VHDL's design constructs for embodying algorithms. A process statement begins with an optional label (mux in this case), followed immediately by a colon (:), then the reserved word process and a sensitivity list. A sensitivity list identifiers which signals will cause the process to execute. Thus in our example, a change in the input signals i1 or i2, or in the selection signal sel will cause the process to be executed. The process body includes sequential statements that when executed, model a single bit 2 channel multiplexer. An if statement is used to decide which channel i1 or i2 must be connected to the output o1 by using the signal sel. The process statement is completed with the reserved words end process and optionally the process label.

Although hardware is concurrent or parallel, and executing simultaneously, it can be modeled by a series of sequential statements that define how outputs react to inputs. An architecture body can contain more than one process, and each process is concurrent with the others.

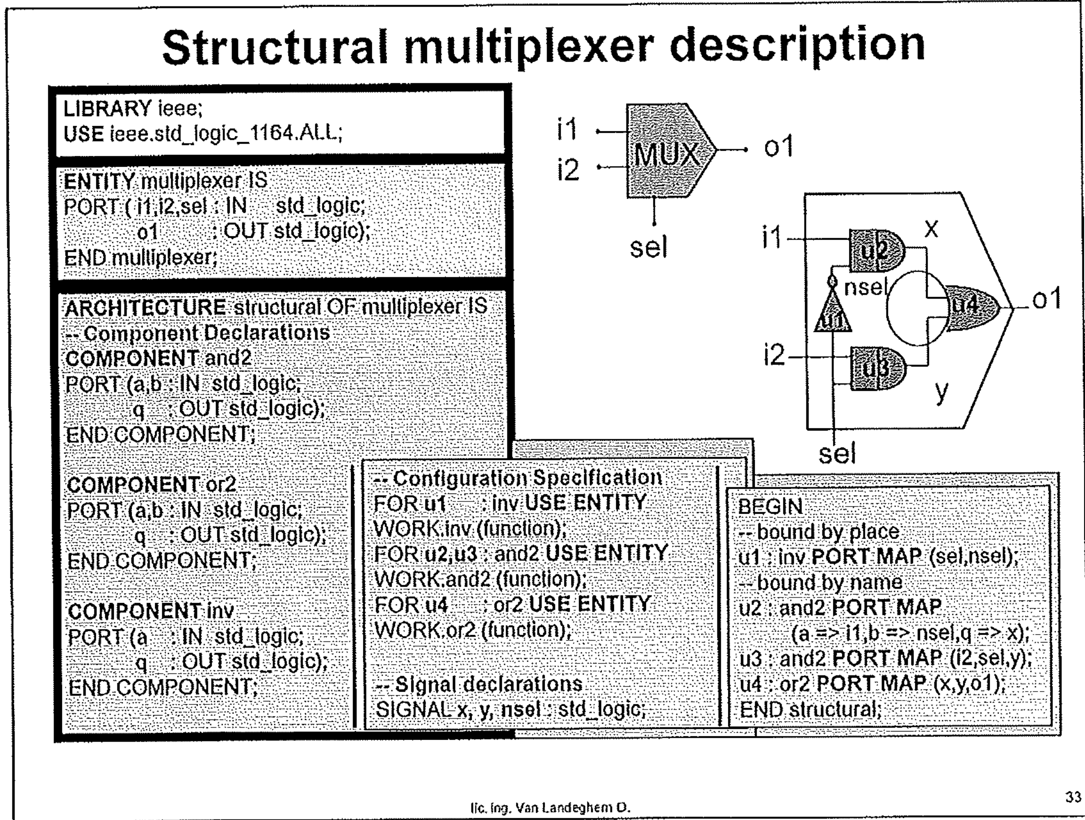


This is an example of a dataflow architecture because it specifies how data will be transferred from signal to signal and input to output without using sequential statements. Dataflow descriptions will likely be used in cases where it is more succinct to write simple equations, conditional signal assignment (when-else) statements, or selected signal assignment (with-select-when) statements rather than a complete algorithm. On the other hand, when structures have to be nested, sequential statements are preferable.

Dataflow architectures use concurrent signal assignment statements rather than processes and their sequential statements. The signal assignment inside a process statement is sequential. Concurrent statements lie outside of process statements. Whereas the order of sequential signal assignment statements in a process can have a significant effect on the logic that is described, the order of concurrent signal assignment statements doesn't matter. An architecture can have multiple signal assignment statements, and the statements will execute concurrently.

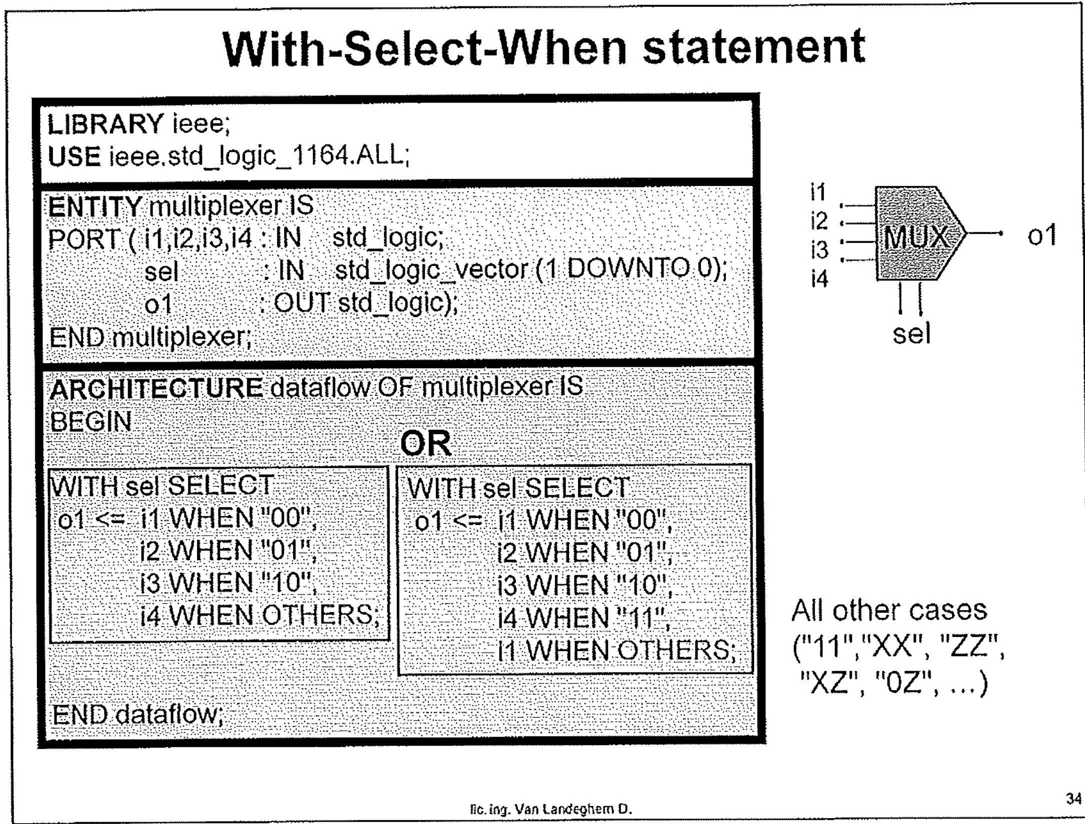


This structural design description requires that inv, and2 and or2 components are defined in a package (and that this package is compiled into a library). Several VHDL files with a single component description or multiple component descriptions in a single VHDL file are both possible. Placing these files in the same directory as the structural description and compiling them, makes the compiled components available in the WORK library. These components can be instantiated in a netlist description style (structural description).



Structural descriptions consist of VHDL netlists. These netlists are very much like schematic netlists. Components are instantiated and connected together with signals. To instantiate a component is to place it in a hierarchical design. Structural designs are hierarchical. Separate entity declarations and architecture body pairs are created for inv, and2, or2 and multiplexer. The multiplexer design contains instances of the inv, and2 and or2 components (when an entity is used inside of another entity, it is referred to as a component). The architecture bodies for inv, and2 and or2 are not contained in the same design file for our multiplexer entity. They are accessed (made visible) by way of a use clause (see configuration specification in the example).

A structural description for a single bit 2 channel multiplexer probably is not an appropriate use of structural descriptions, because it is more cumbersome than necessary. Large designs, however, are best decomposed into manageable subcomponents. Multiple levels of hierarchy may be called for, with the underlying components netlisted (connected) at each level of the hierarchy. Hierarchical design allows the logical decomposition of a design to be clearly defined. It also allows each of the subcomponents to be easily and individually simulated.

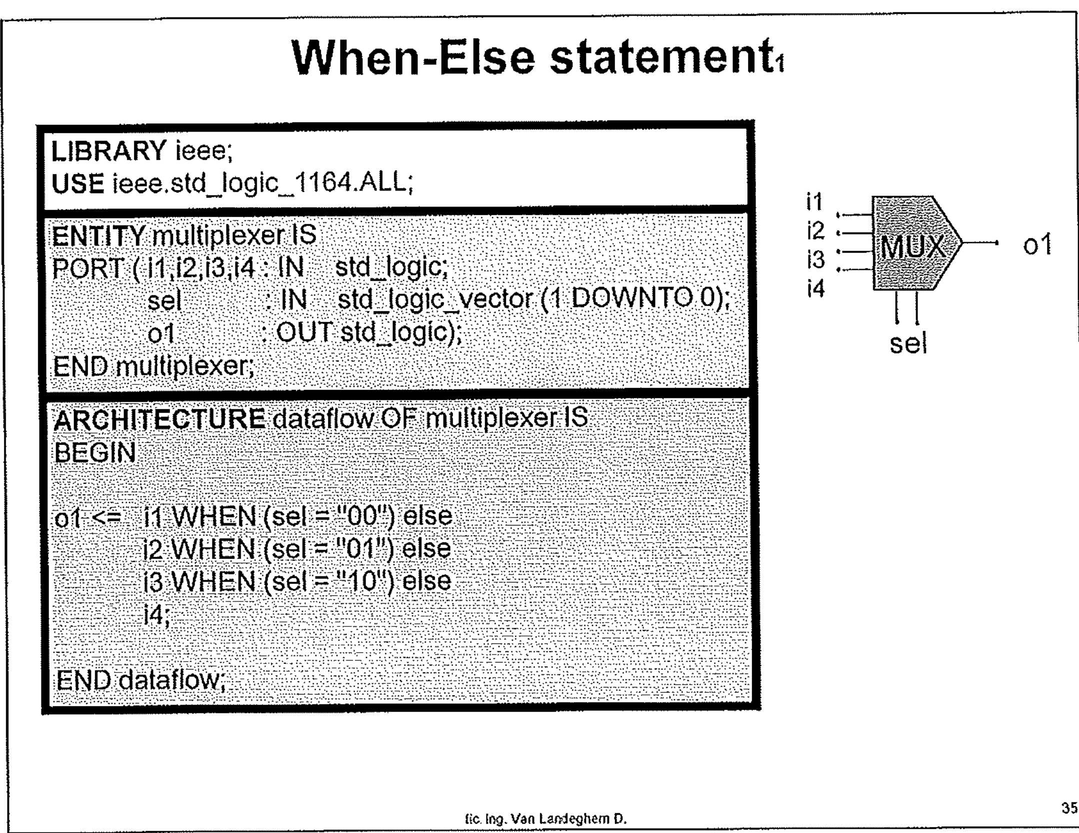


The with-select-when statement provides selective signal assignment, which means that a signal is assigned a value based on the value of a selection signal. In the example above a selective signal assignment statement is used to describe a four to one multiplexer. Based on the value of signal sel, signal o1 is assigned one of four values (i1, i2, i3 or i4). Three values of sel are explicitly enumerated ("00", "01" and "10"). The reserved word OTHERS is used to indicate all other possible values for sel. Others is specified instead of "11" for the following reason: sel is of type std_logic_vector, and there are nine possible values for a data object of type std_logic. If "11" were specified instead of others, only four of the 81 values would be covered in the with-select-when statement. The values "1X", "Z0", "U-", "UU" and "LX" are just a few of the other possible values for sel that include metalogic values. For hardware and synthesis tools, "11" is the only other meaningful value, but the code must be made VHDL-compliant. In simulation, there are indeed 77 other values that sel can have. The value "11" can explicitly be specified as one of the values of sel; however, OTHERS is still required to specify all other possible values of sel.

For synthesis, the results are the same for any of the two versions of the architecture. Results of synthesizing this design to a sum-of-products architecture (a CPLD, for instance) results in an equation similar to the following:

o1= !sel1 & !sel0 & i1 # !sel1 & sel0 & i2 # sel1 & !sel0 & i3 # sel1 & sel0 & i4

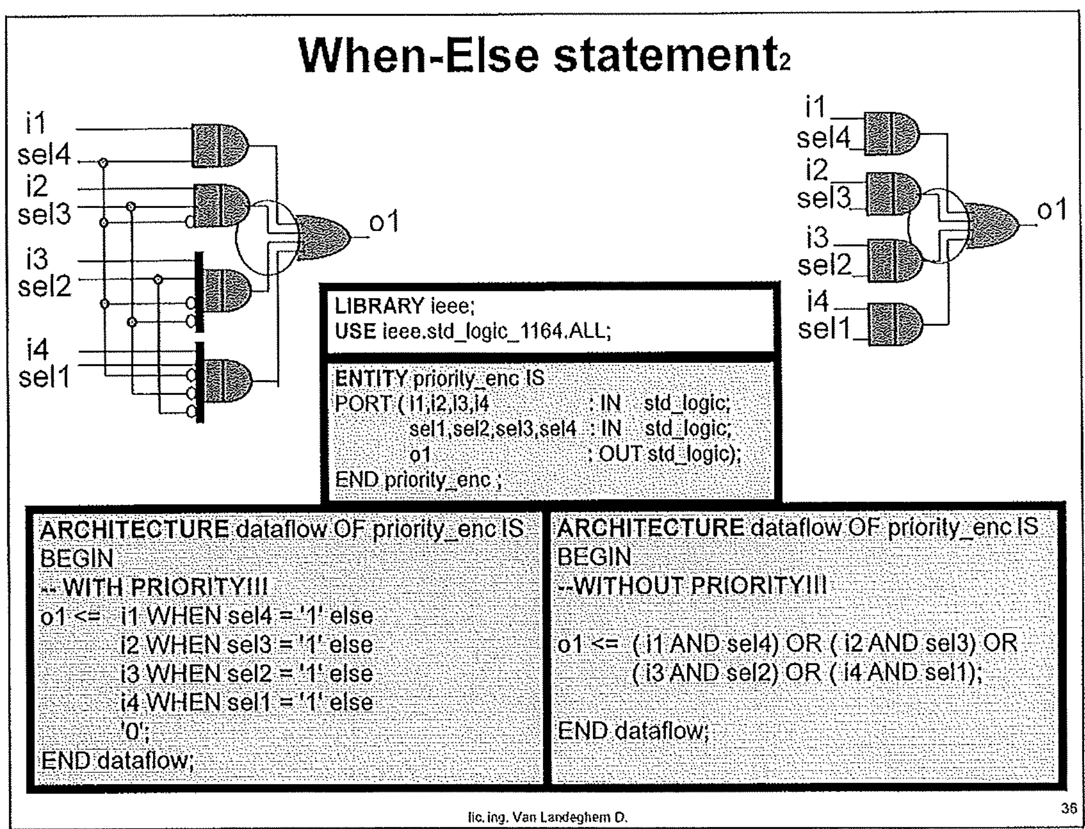
A selective signal assignment describes logic based on mutually exclusive combinations of values of the selection signal. That is, the when conditions can only specify possible values of the selection signal. This is not necessarily true for conditional signal-assignment (when-else) statements.



The when-else statement provides for conditional signal assignment, which means that a signal is assigned a value based on a condition. In the example above a conditional signal-assignment statement is used to describe the four to one multiplexer. Whereas the when conditions in a with-select-when statement must specify mutually exclusive values of the selection signal, the when conditions in a when-else statement can specify any simple expression. However in this example, all of the conditions listed in the when-else statement are mutually exclusive. Thus the Boolean transfer equation produced by synthesizing this design to a sum-of-products architecture results to the following expression:

o1= !sel1 & !sel0 & i1 # !sel1 & sel0 & i2 # sel1 & !sel0 & i3 # sel1 & sel0 & i4

Because the when conditions were mutually exclusive, the equation for this design match the one produced for the with-select-when description.



However, if the conditions in a when-else statement are not mutually exclusive, logic ensures that the highest priority goes to the first when condition listed. Priorities to subsequent when conditions are based on order of appearance.

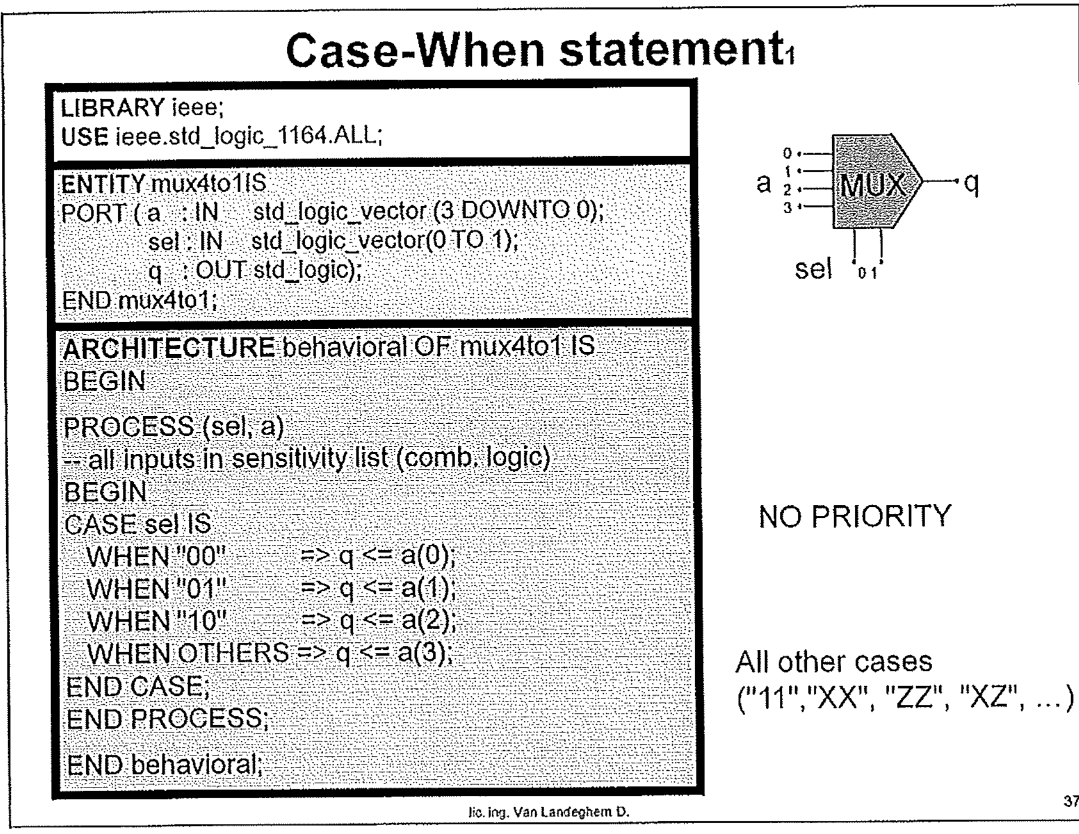
If used with mutually exclusive values of one signal, the when-else construct is slightly more verbose than the with-select-when construct. But this construct can also helps to succinctly describe priority encoders with a Boolean equation similar to the following: o1 = sel4 & i1 #

!sel4 & sel3 & i2 #

!sel4 & !sel3 & sel2 & i3 #

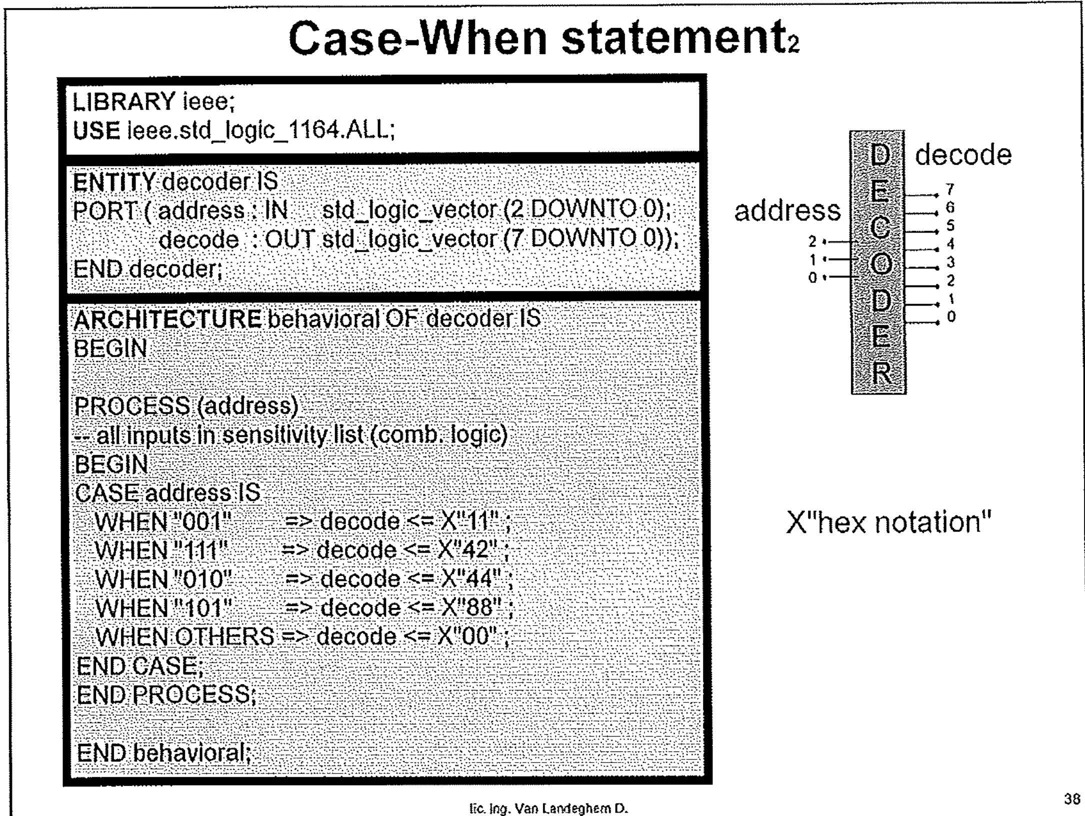
!sel4 & !sel3 & !sel2 & sel1 & i4

This equation and the when-else statement indicate the priority that o1 is assigned the value of i1 when sel4 is asserted, even if sel3, sel2 or sel1 is asserted. Signal sel3 holds priority over sel2 and sel1, and signal sel2 holds priority over sel1. If, however, sel4, sel3, sel2 and sel1 are mutually exclusive (that is, if it is known that only one will be asserted at a time), then the code of the second example above is more appropriate. This second design ensures that the logic produced by synthesis, requires AND gates with fewer inputs than in the first example. Although using wider AND gates in a CPLD does not usually require additional resources, the wider AND gates could have an impact on the implementation in an FPGA.. Wider AND gates could require additional logic cells and levels of logic. These designs are not functionally equivalent, however.



Case statements are used to specify a set of statements to execute based on the value of a given selection signal. A case-when statement can be used for example, as the equivalent of a with-select-when statement. The case-when statement describes how the output q is driven based on the value of sel (of the four to one multiplexer). The reserved word others is used to completely define the behavior of the output q for all possible values of sel. When others covers the sel input combination "11" as well as all of the metalogical values. Although multiple statements may be specified (and executed) for each when condition, this example has just one signal assignment statement per condition.

Because the case-when statement belongs to the class of sequential statements, the case statement must be packed into a process statement. The process sensitivity list contains the signals sel and a, which will cause the process to execute.



The case-when statement in this example describes how decode is driven based on the value of the input address. The reserved word others is used to completely define the behavior of the output decode for all possible values of address. When others covers the address input combinations "000", "011", "100" and "110", as well as all of the metalogical values. The equations generated by synthesizing this design are:

decode7 = address2 & !address1 & address0

decode6 = !address2 & address1 & !address0 # address2 & address1 & address0

decode5 = GND

decode4 = !address2 & !address1 & address0

decode3 = address2 & !address1 & address0

decode2 = !address2 & address1 & !address0

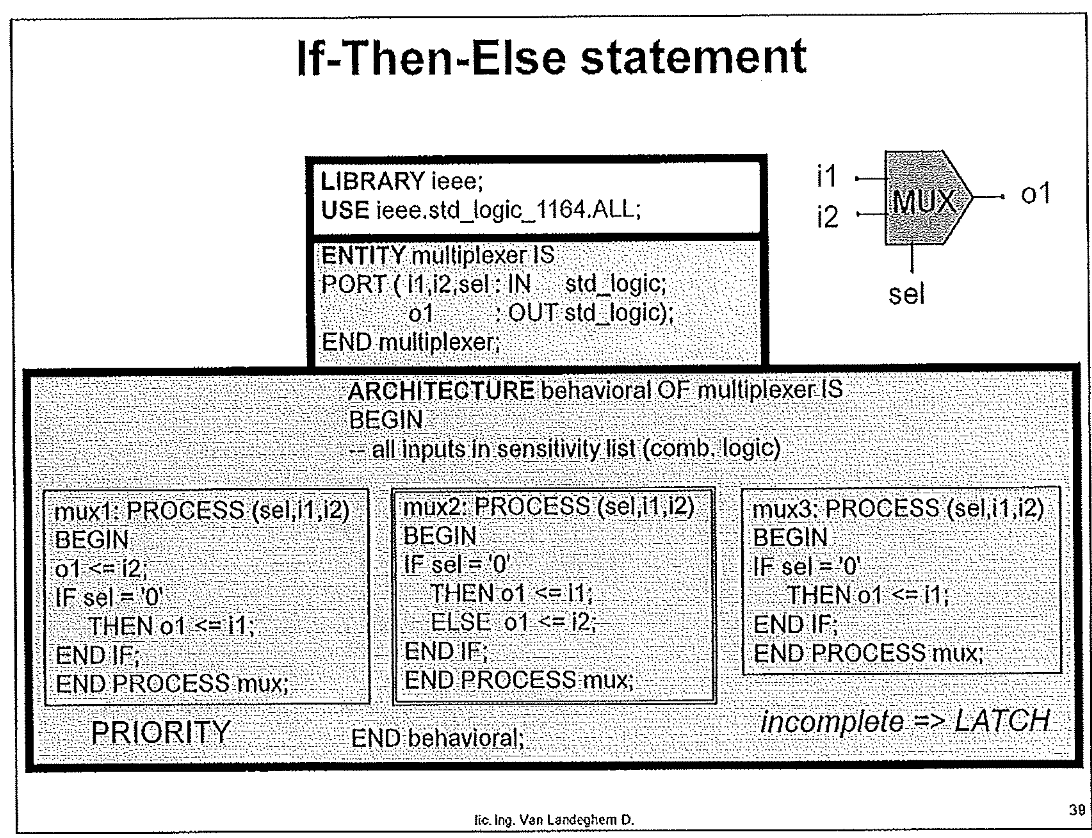
decode1 = address2 & address1 & address0

decode0 = !address2 & !address1 & address0

Many of these equations turn out to have common product terms. The bits 7 and 3 are equivalent, as well as the bits 4 and 0. The bits 6 and 2 share a common product term, as well as the bits 6 and 1.

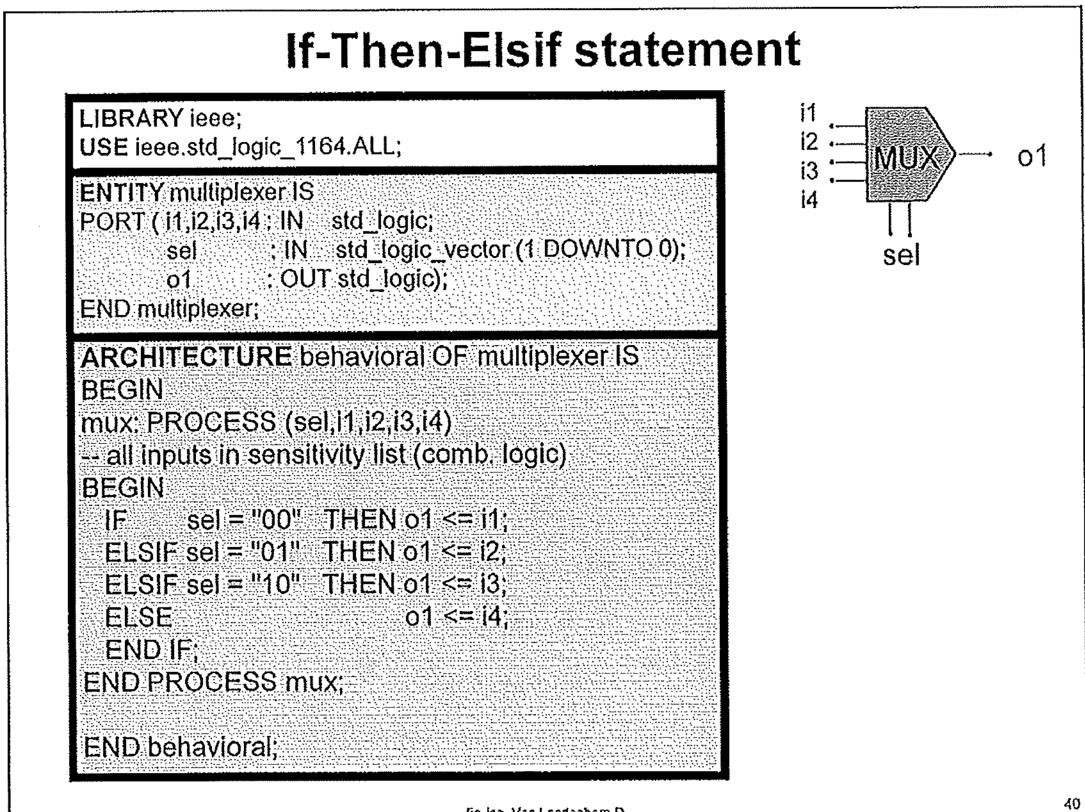
In a CPLD, these common product terms may share resources. Some architectures produce an incremental delay when shared product terms are used, in which case, depending upon the performance requirements of this design, it is not preferable to use these shared product terms.

An implementation in an FPGA could take advantage of equivalent signals, the output of one logic cell could drive two I/O cells, such as decode4 and decode0. The shared gates between the bits 6, 2 and 1 probably would not provide an advantage in most FPGAs because an additional level of logic would likely be required.



The if-then-else construct is used to select a set of statements to execute based on a Boolean evaluation (true or false) of a condition(s). Because sequential statements are executed in order of appearance, the processes mux1 and mux2 are functionally equivalent. With either of these processes, o1 will assume the value of i1 if sel is equal to '0' and i2 if sel is equal to '1'.

The process mux3 does not describe the same logic because neither a default value nor an else assignment is specified for o1. The process mux3 implies that o1 should retain its value if sel is not equal to '0'. This is referred to as implied, or implicit memory. This must be avoided because synthesis of this design results in a circuit with a latch, what is not the wanted functionality. To prevent the synthesis of a latch on o1, be sure to include a default value, or preferable, complete the if-then with an else.



The if-then-else can expanded further to include an elsif to allow for further conditions to be specified and prioritized. Because the conditions in this example are mutually exclusive values of sel, the logic described here synthesizes to the same for the previous four to one multiplexer implementations. However, because there is priority in the conditions (as in a when-else statement), an if-then statement is not the best choice when the conditions involve multiple signals that are mutually exclusive. Using an if-then-else statement in this case may cause additional logic to be synthesized to ensure that previous conditions are not true, as with the when-else statement. Instead, a Boolean equation, or case-when statement, may be more appropriate.

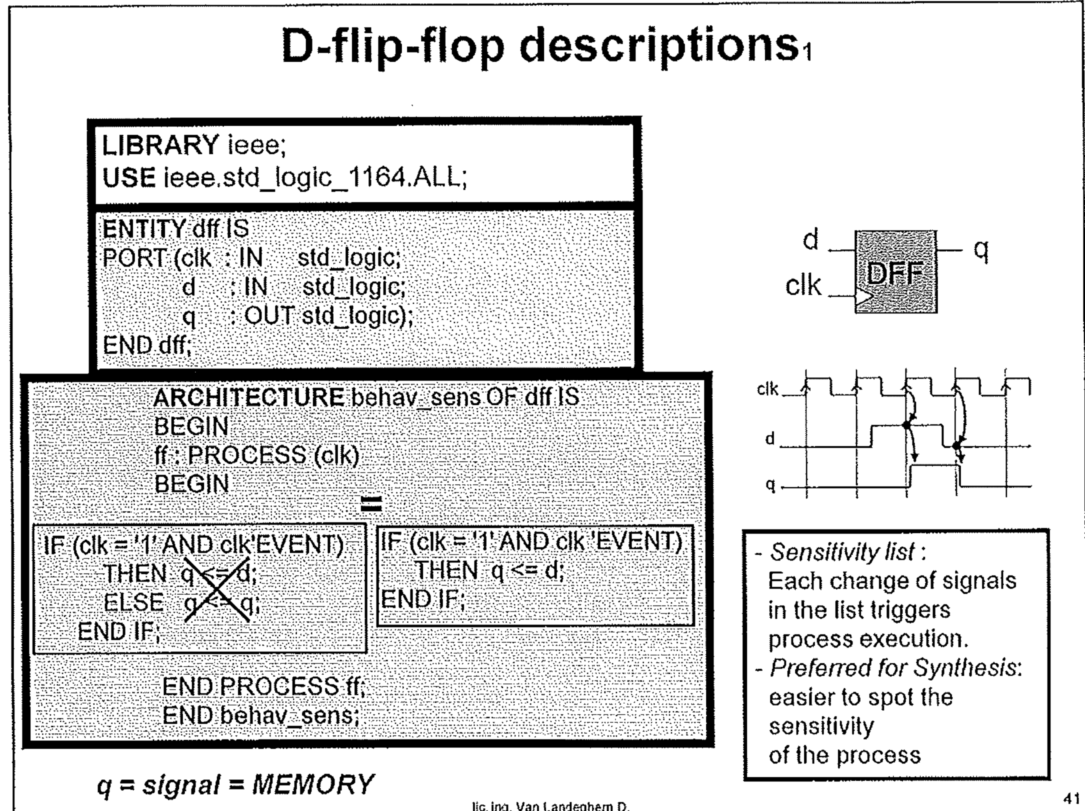
Signals i1, i2, i3, i4 and sel are included in the process sensitivity list because a change in any one of them should cause a change in (or evaluation of) o1. If sel were omitted from the sensitivity list, then o1 would not change with a change in sel, only changes in i1..i4 would lead to a change in o1, and this would not describe a multiplexer. The design equation produced by synthesizing this design description

results to the following expression:

o1= !sel1 & !sel0 & i1 # !sel1 & sel0 & i2 # sel1 & !sel0 & i3 # sel1 & sel0 & i4

If mapped to a PLD-like architecture, sel can easily be implemented with one macrocell and four product terms.

If mapped to an FPGA, it is up to the synthesis and optimization tools to optimally map the equation to the device architecture. For device architectures with multiplexers, this design should fit nicely.



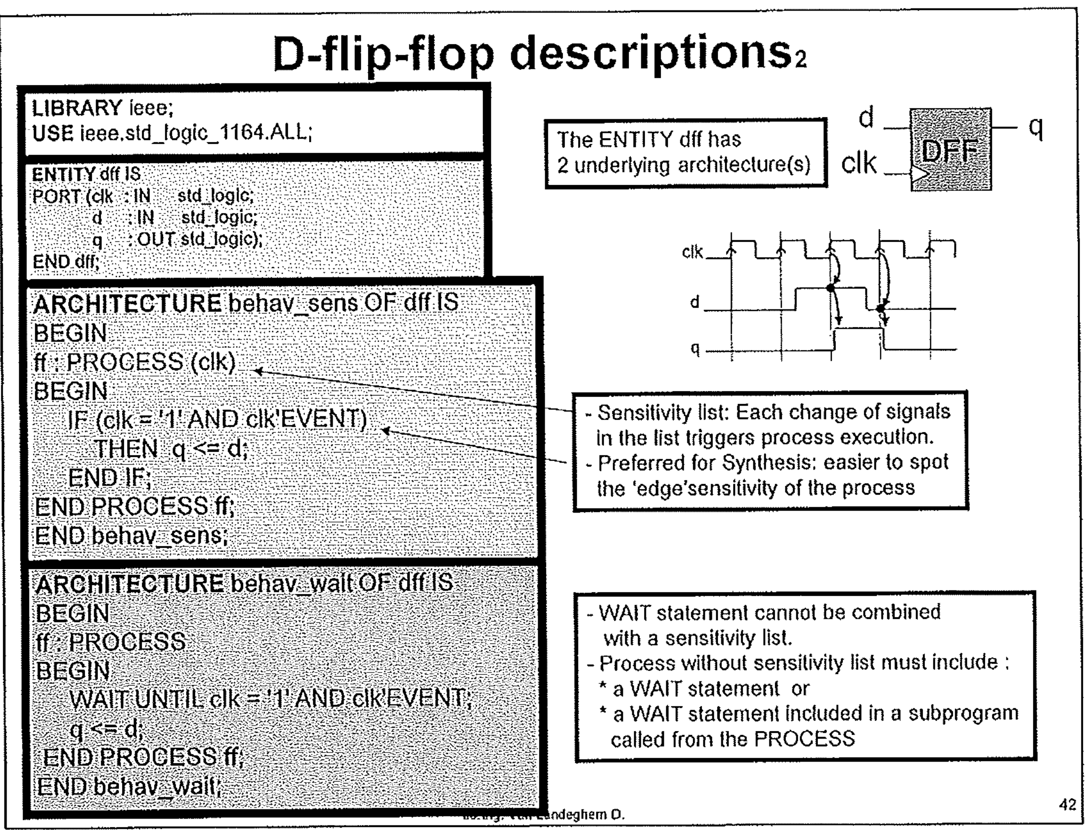
Programmable logic devices lend themselves well to synchronous applications. Most device architectures have blocks of combinational logic connected to the inputs of flip-flops as the basic building block for a CPLD macrocell or an FPGA logic cell.

The listed code above describes a simple DFF. This process is sensitive only for changes in clk. Thus, a VHDL simulator activates this process only when there are clk transitions. A transition in d does not cause a sequencing of this process. The if clk'EVENT condition is true only when there is a change in value, that is an event of the signal clk (remark that 'EVENT is an attribute that when combined with a signal forms a Boolean expression that indicates when the signal transitions).

The clk'EVENT expression and the process sensitivity list are redundant, they both detect changes in clk. Because some synthesis tools ignore sensitivity lists, the clk'EVENT expression should be included to describe edge triggered events. Because the change in clk can be either from '0' to '1' (a rising edge) or '1' to '0' (a falling edge), the additional condition clk = '1' is used to define a rising edge event. The statements inside the if statement are executed only when there is a change in the state of the clk from '0' to '1' which is a synchronous event, allowing synthesis software to infer from the code a positive edge triggered DFF.

Without an else, there is implied memory (that q will keep its value), which is consistent with the operation of the DFF. In other words, the example on the left side has the same meaning for simulation. But, most synthesis tools will not handle an else expression following an if (clk'EVENT and clk='1') because it may describe logic for which the implementation is ambiguous. For example, it is unclear how the following description should be synthesized:

```
if (clk 'EVENT and clk='1')
      then q <= d;
      else  q<= a;
end if;
```



The behavior of this registered circuit can also be described by using the WAIT UNTIL statement instead of the if (clk'event and clk='1') statement. The process in the second architecture behav_wait does not use a sensitivity list, but begins with a wait statement. A process that has a wait statement cannot have a sensitivity list (the wait statement implicitly defines the sensitivity list). For descriptions that are to be synthesized, the wait until statement must be the first in the process. Because of this synchronous logic described with a wait statement cannot be asynchronously reset. Interpreting the code fragment of architecture behav_wait in terms of simulation, this process is suspended until the condition following the wait until statement is true. Once true, the signal assignments that follow the wait statement are made, after which the process once again waits for the clock to be asserted. So, in this case, once the clk signal becomes equal to '1' (that is, on the rising edge of clk), q will be assigned the value of d, thereby describing a DFF without an asynchronous reset or preset.

Despites both architectures results after synthesis in an equal circuit, most of the designers will prefer a process with sensitivity list. This is because it gives a better overview of the sensitivity of the process, specially when an architecture contains more communicating processes to describe the circuit.

Predefined Attributes

- Attributes describe a characteristic of:
  entities, architectures, types, signals
- Predefined attributes useful in synthesis :
  * **value attributes** : 'LEFT, 'RIGHT, 'HIGH, 'LOW, 'LENGTH, ...

Example:
TYPE count IS integer RANGE 0 TO 127;
TYPE states IS (idle, even, odd, error);
TYPE reg IS ARRAY(15 DOWNTO 0) OF std_logic;

count 'LEFT = 0	states 'LEFT = idle	reg 'LEFT = 15
count 'RIGHT = 127	states 'RIGHT = error	reg 'RIGHT = 0
count 'HIGH = 127	states 'HIGH = error	reg 'HIGH = 15
count 'LOW = 0	states 'LOW = idle	reg 'LOW = 0
count 'LENGTH = 128	states 'LENGTH = 4	reg 'LENGTH = 16

  * **signals attributes** : 'EVENT, 'LAST_VALUE...
  * **ranges attributes** : 'RANGE...

Example:
SIGNAL register : std_logic_vector (15 DOWNTO 0);

register 'RANGE = 15 DOWNTO 0

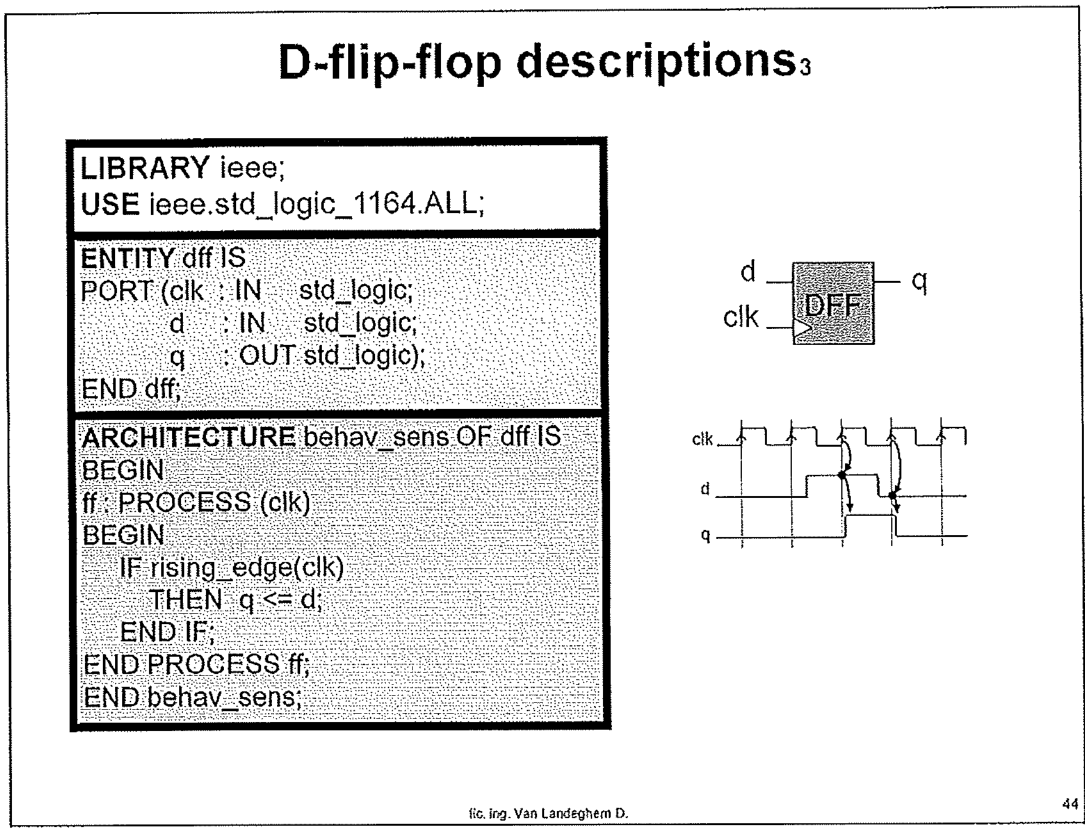
lic. Ing. Van Landeghem D. 43

An attribute provides information about items such as entities, architectures, types and signals. There are several predefined value, signal and range attributes that are useful in synthesis.

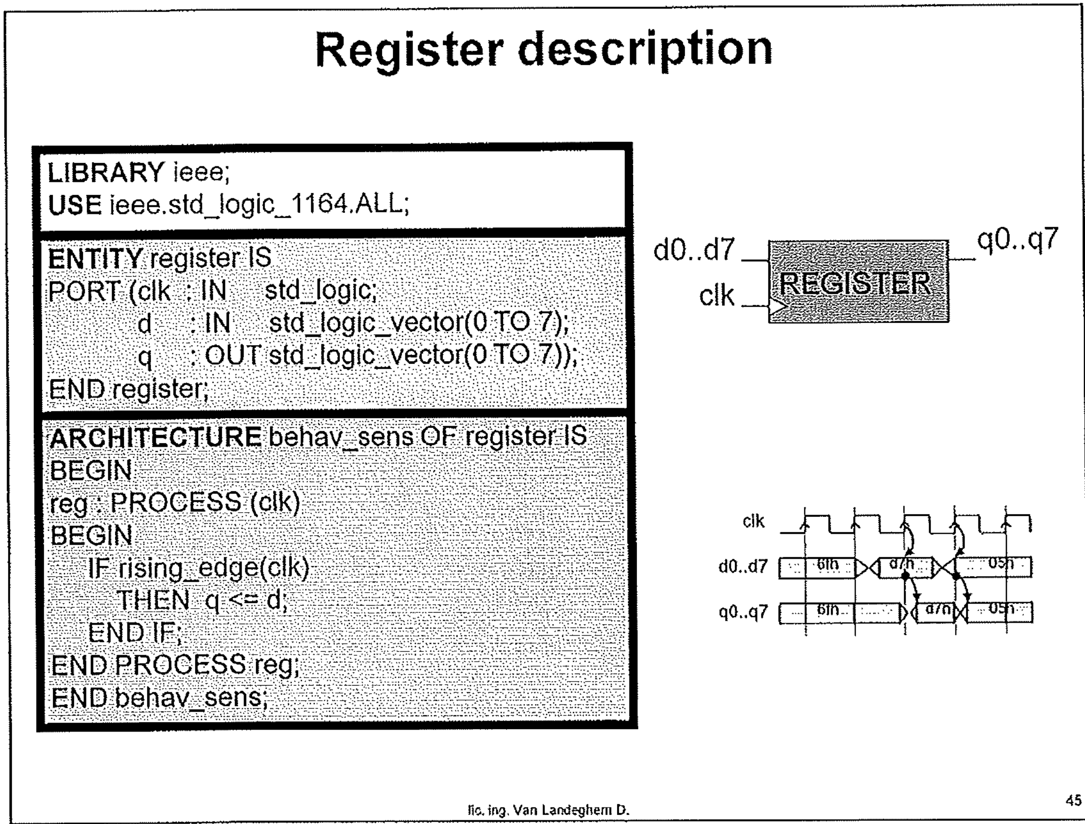
Scalar types have value attributes. The value attributes are ' left, ' right, 'high, ' low and ' length. The attribute ' left yields the leftmost value of a type and ' right the rightmost. The attribute ' high yields the greatest value of a type. For enumerated types, this value is the same as ' right. For integer ranges, the attribute ' high yields the greatest integer in the range. For other ranges, ' high yields the value to the right of the keyword to or to the left of the keyword downto. The attribute ' low yields the value to the left of to or the right of downto. The attribute ' length yields the number of elements in a constrained array.

An important signal attribute useful for both synthesis and simulation is the ' event attribute. This yields a Boolean value of true if an event has just occurred on the signal for which the attribute is applied. It is used primarily to determine if a clock has transitioned.

A useful range attribute is the ' range attribute, which yields the range of a constrained object.

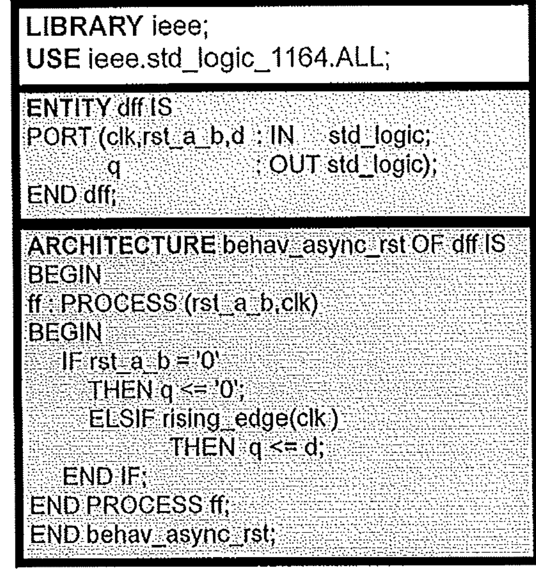


The std_logic_1164 package defines the functions rising_edge and falling_edge to detect rising and falling edges of signals. One of these functions can be used in place of the (clk 'event and clk='1') expression if the clk signal is of type std_logic. These functions are preferred by some designers because in simulation the rising_edge function will ensure that the transition is from '0' to '1' and not some other transition such as 'U' to '1'. However, the (clk 'event and clk='1') and rising_edge(clk) expressions will be used interchangeable.

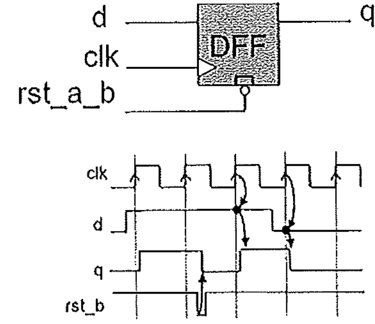


This architecture is similar to that of the previous one, except in this design, q and d are vectors, not bits. So this design describes an 8-bit register.

D-flip-flop with asynchronous reset



```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dff IS
PORT (clk,rst_a_b,d : IN   std_logic;
      q             : OUT std_logic);
END dff;

ARCHITECTURE behav_async_rst OF dff IS
BEGIN
ff : PROCESS (rst_a_b,clk)
BEGIN
   IF rst_a_b = '0'
     THEN q <= '0';
     ELSIF rising_edge(clk)
            THEN q <= d;
   END IF;
END PROCESS ff;
END behav_async_rst;
```



-The rst_a_b condition has to be described BEFORE the clk condition.
-The rst_a_b signal must be PART of the sensitivity list.

None of the previous examples make reference to resets or initial conditions. The VHDL standard does not require to be reset or initialize a circuit. The standard specifies that for simulation, unless a signal is explicitly initialized, it gets initialized to the 'left value of its type. So a signal of type std_logic will get initialized to 'U', the uninitialized state and a bit will get initialized to '0'. In the hardware world, however, this is not always true, not all devices power up in the reset state and the uninitialized state is physically meaningless. What if global and local resets are required to place the logic in a known state? Resets end presets can be described with simple modifications to the previous code of the DFF.

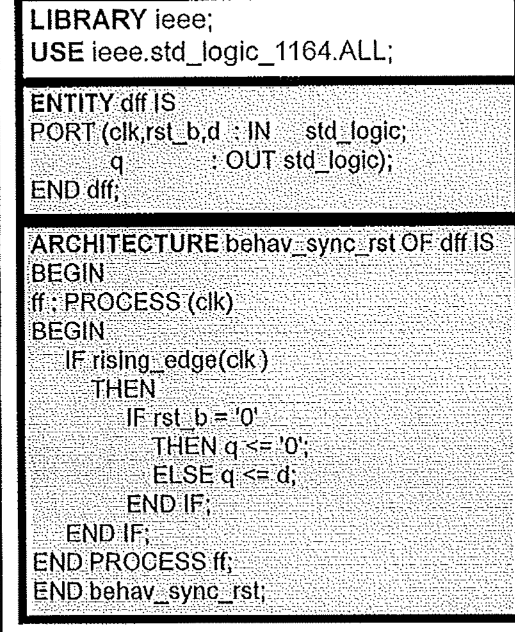
For an *asynchronous* reset the signal must be included in the sensitivity list of the process describing the flip-flop. The sensitivity list indicates that this process is sensitive to changes in clk and reset. A transition in either of these signals will cause a simulator to sequence through the process. This code accurately describes a DFF with an asynchronous reset. If reset gets the value '0', then signal q will be assigned '0', regardless of the value of clk. If reset is not asserted and the clock transition is from '0' to '1', then the signal q will be assigned the value of signal d. This process cause synthesis software to infer an asynchronous active low reset.

A preset can be described in a similar way as a reset:

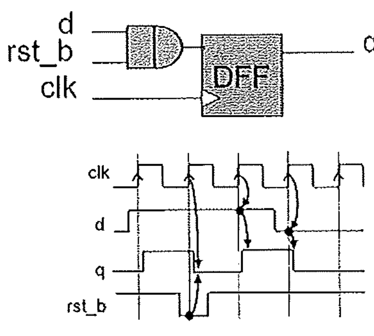
IF preset = '1'	instead	IF rst = '0'
THEN q <= '1';	of	THEN q <= '0';
ELSIF rising_edge(clk)…		ELSIF rising_edge(clk)…

This cause synthesis software to infer an asynchronous active high preset.

D-flip-flop with synchronous reset



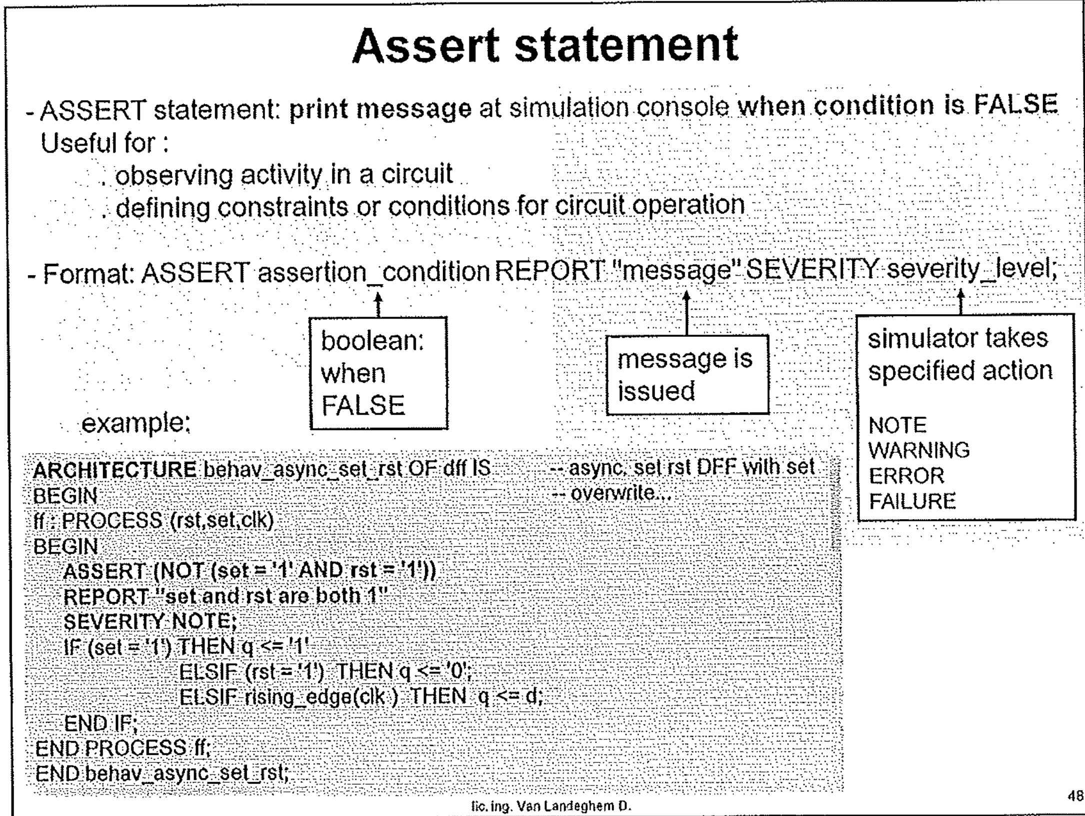
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dff IS
PORT (clk,rst_b,d  : IN    std_logic;
        q          : OUT std_logic);
END dff;

ARCHITECTURE behav_sync_rst OF dff IS
BEGIN
ff : PROCESS (clk)
BEGIN
   IF rising_edge(clk )
      THEN
         IF rst_b = '0'
            THEN q <= '0';
            ELSE q <= d;
         END IF;
   END IF;
END PROCESS ff;
END behav_sync_rst;
```



-The rst_b condition has to be described AFTER the clk condition.
-The rst_b signal is NOT A PART of the sensitivity list.

lic. ing. Van Landeghem D. 47

A *synchronous* reset or preset of a DFF is realized by putting the reset or preset condition inside the portion of the process that describes logic that is synchronous to the clock. The code listed above describes a process that is sensitive only to changes in the clock. The synthesis tool will produce a DFF that is synchronously reset whenever the reset signal is asserted and is sampled by the rising edge of the clock. For CPLDs without synchronous resets the implementation of synchronous resets requires additional product terms to use. However, it is important to know that in the case of an asynchronous reset command, the signal must be hazard free. This means that the generating combinational circuit require a hazard free design. For this reason and also a good design for testability strategy the preferred reset solution is a synchronous one.



The assert statement is used for observing circuit activity. During simulation the assert statement indicates constraints or conditions overflow.

• With an ASSERT statement a logical expression is claimed to be true. If it is false the rest of the ASSERT statement is executed.

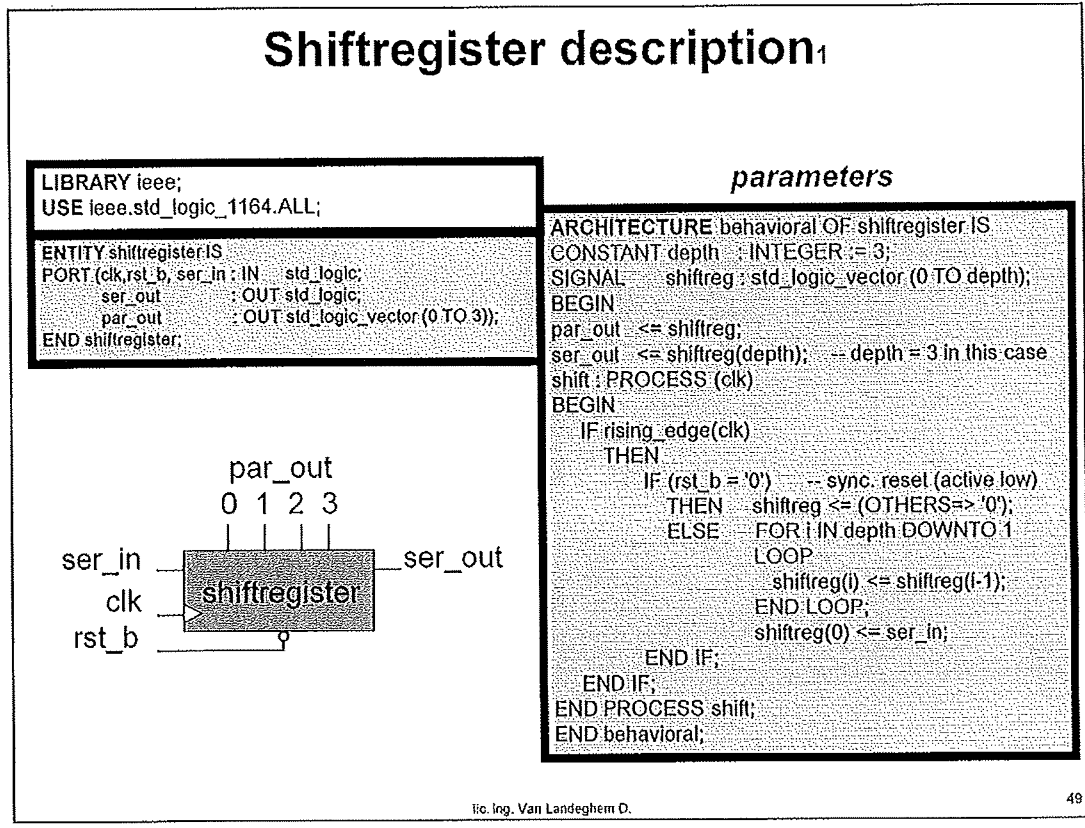
• SEVERITY has four possible values – NOTE, WARNING, ERROR and FAILURE

- NOTE: specific conditions
- WARNING: not expected conditions
- ERROR: conditions that will cause the model to work incorrectly
- FAILURE: conditions that are catastrophic

• ASSERT has default severity level ERROR while REPORT has NOTE.

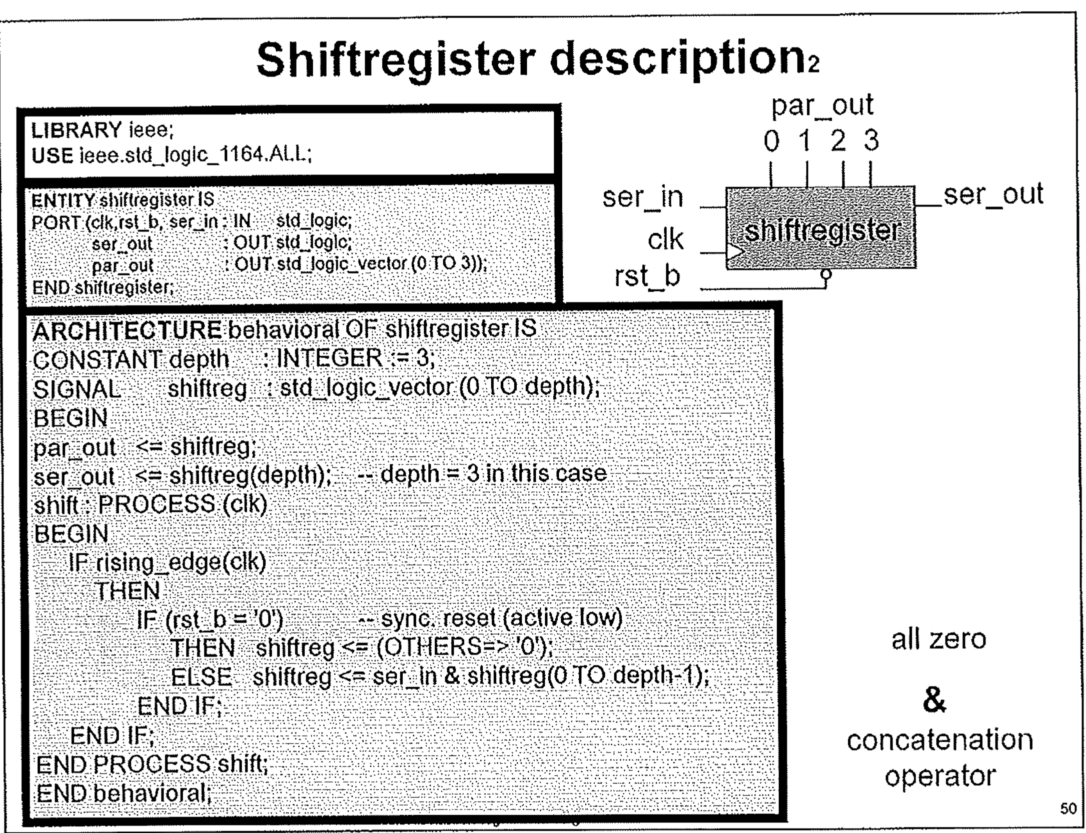
• ASSERT is both a sequential and a concurrent statement while REPORT only is a sequential statement. However a concurrent ASSERT statement may include a REPORT statement.

In the given example the message "set and rst are both 1" will appear when the Boolean expression « NOT(set='1' AND rst='1') » is false. Because the severity level is not 'failure', the simulator can continue analyzing the design. Since in this design description the set signal has priority to the rst, the assert statement offers an efficient method to check if both signals are '1' at the same time. Notice that the assert statement doesn't has any effect on the result of the synthesis tool.

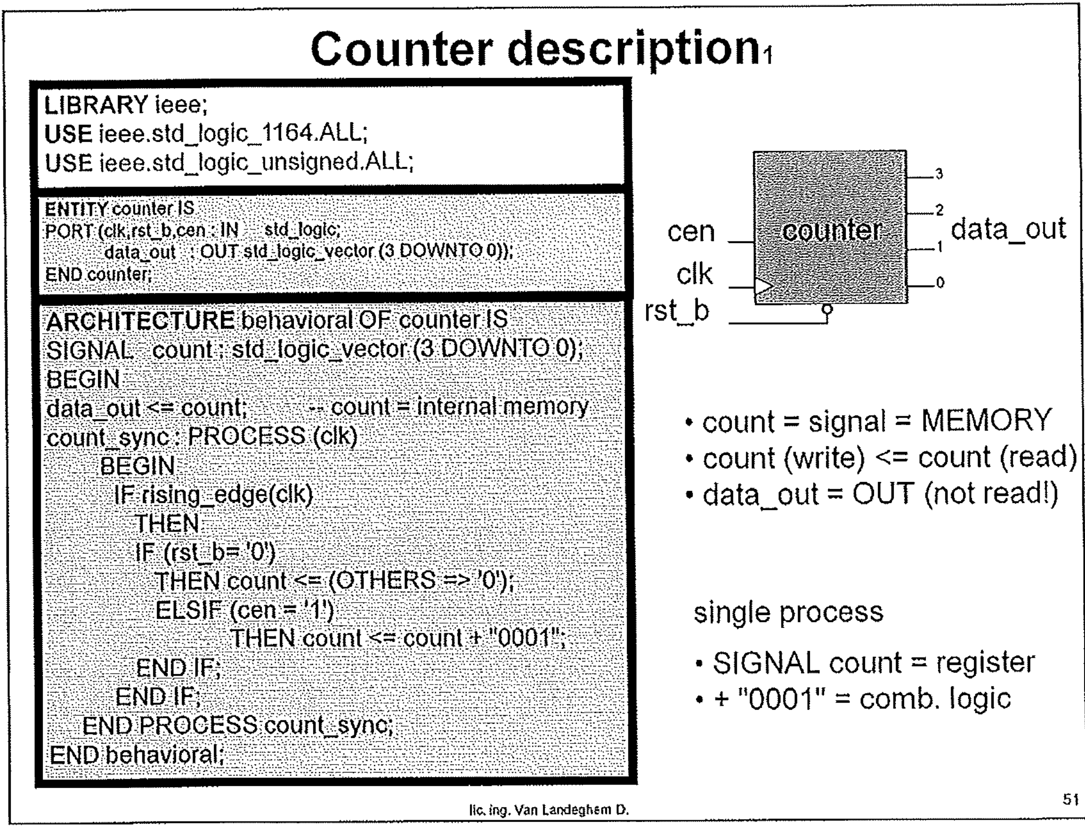


Loop statements are used to implement repetitive operations and consist of either for loops or while loops. The for statement will execute for a specific number of iterations based on a controlling value. The while statement will continue to execute an operation as long as a controlling logic condition evaluates true. An extra step is required to initialize the controlling variable of a while statement. Take for instance the loop used in the above shift process. This loop sequences 3 std_logic values to describe a shift operation during one clk cycle: shiftreg(2) assigned to shiftreg(3), shiftreg(1) assigned to shiftreg(2) and shiftreg(0) assigned to shiftreg(1). Outside the loop shiftreg(0) gets the incoming signal ser_in. In a for loop, the loop variable is automatically declared. A while loop can be used here instead of a for loop, but it requires the additional overhead of declaring, initializing and incrementing the loop variable. For example: shift : PROCESS (clk)

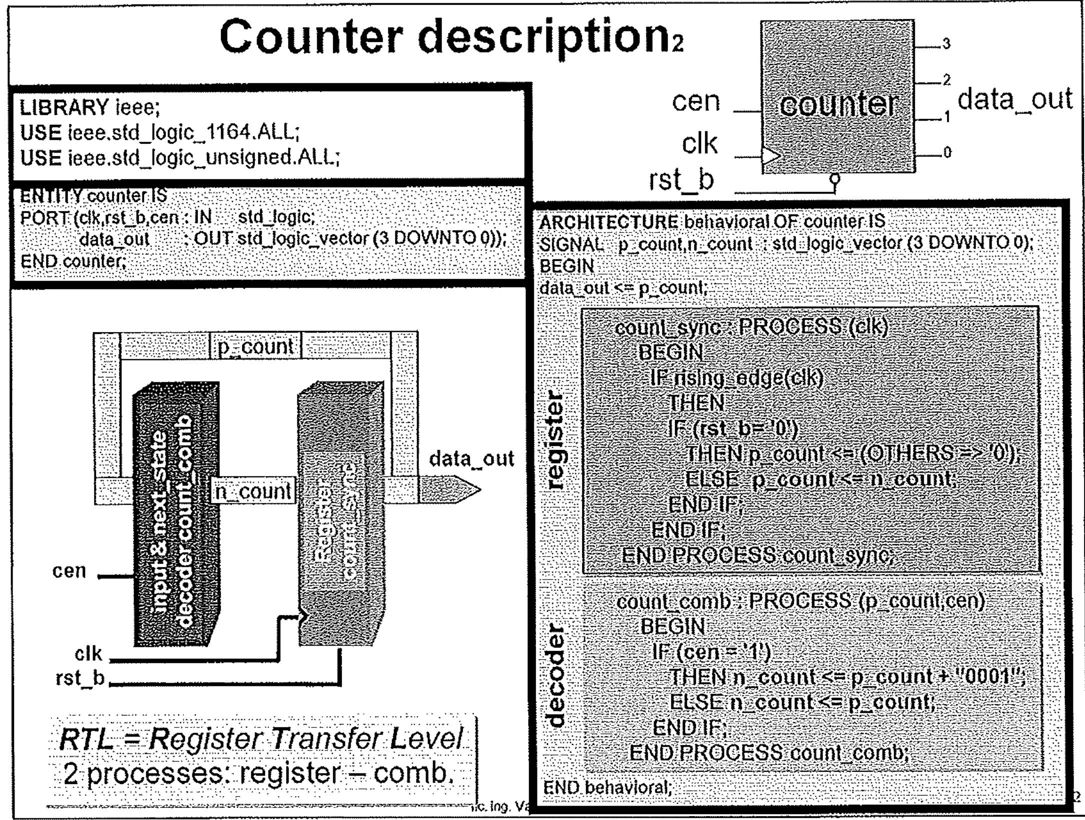
```
VARIABLE i : integer :=3;
BEGIN
 IF rising_edge(clk)
      THEN
          IF (rst = '0')
             THEN    shiftreg <= (OTHERS=> '0');
             ELSE    WHILE  i > 0  LOOP
                          shiftreg(i) <= shiftreg(i-1);
                          i := i -1;
                     END LOOP;
                     shiftreg(0) <= ser_in;.............
```



A popular method for describing a shiftregister is listed above. The four DFFs are represented by the signal shiftreg of type std_logic_vector of size four. Par_out cannot be used to represent the DFFs since it is an output and the FF outputs must be used internally. The right shift is achieved by applying the concatenation operator & to the shiftreg input ser_in and to the left three bits of shiftreg. This moves the contents of shiftreg one bit to the right and loads the value on ser_in into the leftmost bit. Concurrent to the process that performs the shift operation are two statements, one which assigns the value in shiftreg to the output par_out and the other which defines the shiftreg out signal ser_out as the contents of the rightmost bit of shiftreg.



This VHDL code describes the counter at the behavioral level. The four DFFs are represented by the signal count of type std_logic_vector and of size four. Data_out cannot be used to represent the DFFs since it is an output and the DFF outputs must be used internally. Counting up is achieved by adding 1 in the form of "0001" to count. Since addition is not a normal operation on type std_logic_vector, it is necessary to use an additional package from the ieee library, std_logic_unsigned.all, which defines unsigned number operations on type std_logic. Besides performing the reset command and the counting up operation, the process checks the count enable signal cen. If cen is '1' then the counter will increment with one, on the other hand when cen is not equal to '1' the counter stays in his memory state. Concurrent to the process that performs the counting up is a statement which assigns the value in count to the output data_out.



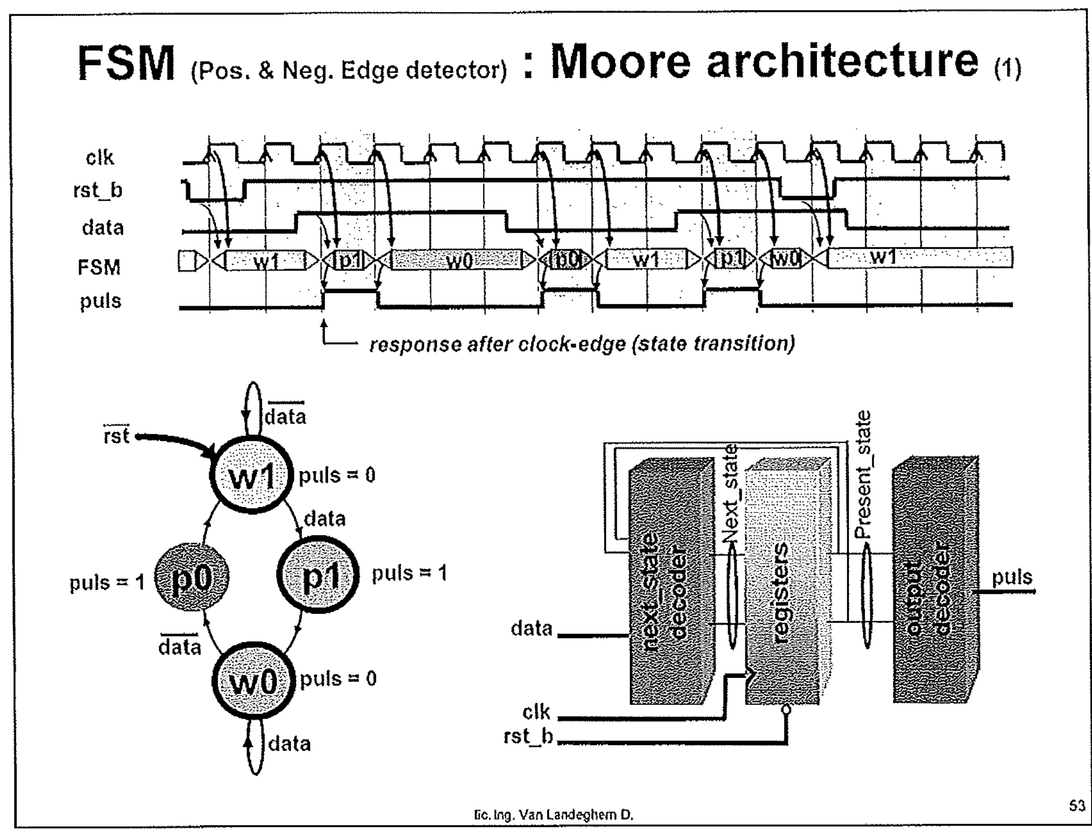
This VHDL code describes the counter at the behavioral level. The global counter function here is decomposed in the input & next_state decoder and the registers. The input & next_state decoder contains all the combinational logic of the counter, while the register block only contain a set of DFFs (in this case four). This is corresponding to hardware reality.

Since DFFs only can function as a memory, it looks logical that the output of these FFs contain the present state (p_count) of the counter, and the input contain the next state (n_count) of the counter. The present state becomes equal to the next state value on the rising edge of a clock. So, the count_sync process describes the data transfer on a positive edge of the clock for every DFF of counter p_count. This process is completely equivalent with the one of a single DFF as shown before.

The process count_comb describes the input & next_state decoder. Because this is a completely combinational circuit part the sensitivity list must contain all inputs, cen and all the bits of p_count. This means the output n_count of process count_comb depends on the single input cen and the value of the feedback signal p_count.

Notice, that the two processes in the architecture above are concurrent. The process count_sync will update the value of p_count every rising edge of the clock. The process count_comb will update the value of n_count on each event of cen or p_count. Therefore, p_count is the output of the 4 DFFs and is assigned to data_out. This takes place outside the two processes and brings the number of concurrent statements inside the architecture to 3.

This coding style gives from an educational point of view a better understanding of hardware reality and a good preparation designing state machines with VHDL.



Take for an example the problem of a positive and negative edge detector described in the timing diagram. A controller must be used to generate a synchronous puls (pulse width=Tclock) each time data transitions. The signal data is asynchronous with reference to the clock. Good design methodology tells us that the first step is to construct a state diagram. Analyzing the problem (timing diagram), 4 different situations can be seen:

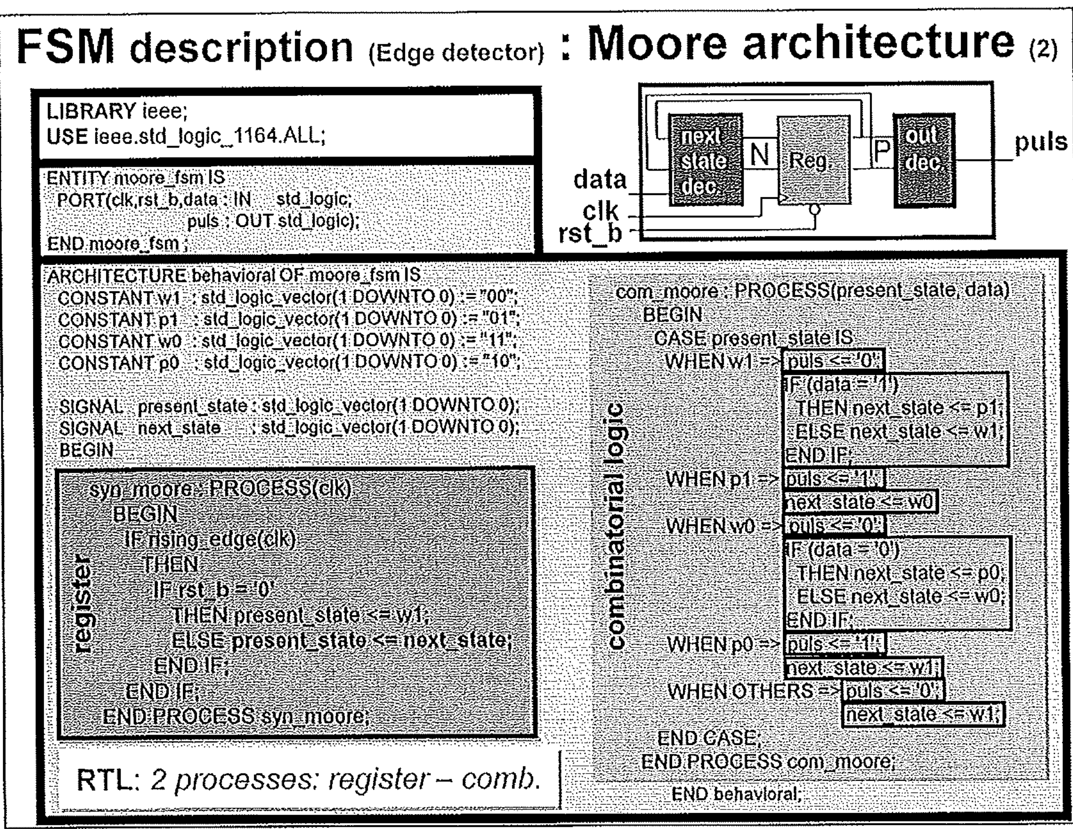
the first, data is '0' and so is puls; the second, data is '1' and so is puls;

the third, data is '1' and puls is '0'; the fourth, data is '0' and puls is '1';

This means a Finite State Diagram is needed with 4 states (w1, p1, w0 and p0). To finish the FSM the state transitions must be drawn. These transitions depend on the input signal data and the present state. So, when the FSM is in the initial state w1 (after a reset for instance), data must be '1' to become a transition to state p1. If data is not '1' , there will be no transition and the FSM keeps state w1. On the other hand the FSM transitions from state p1 to w0 after one Tclock because there is no condition specified. Once the FSM is in state w0 data must be '0' to become a transition to state p0, if data is not '0' the state w0 doesn't change. Finally the FSM transitions from state p0 to state w1 after one Tclock because of the absence of a condition.

Notice, that besides an asserted condition, an FSM requires a positive edge on the clock to transition from one state to an other.

To generate the puls signal, the corresponding FSM states must be decoded. In this case it is very simple: when the FSM state is equal to p1 or p0 the output signal puls must be '1', otherwise it must be '0'. A sequential circuit whose outputs depend on the state alone, is called a Moore machine.



Until now the basic building blocks and language constructs in VHDL were covered. To design FSMs these concepts will be applied. FSMs are commonly used control blocks. The focus lies on a synthesizable design style for state machines.

The FSM diagram can be translated into a behavioral VHDL description without having to perform the state transition table or determine the next-state equations based on the FF types available. The example of the positive and negative edge detector is used. Writing a behavioral FSM description in VHDL is simply a matter of translating a FSM diagram to case-when and if-then-else statements. Each state can be translated to a case in a case-when construct. The state transitions can then be specified in if-then-else statements.

Declarartion

A constant of type std_logic_vector (1 DOWNTO 0) is defined for each state name (w1, p1, w0, p0). Two signals (present_state and next_state) of the same type are defined. They represent the output and input of the memory of the FSM.

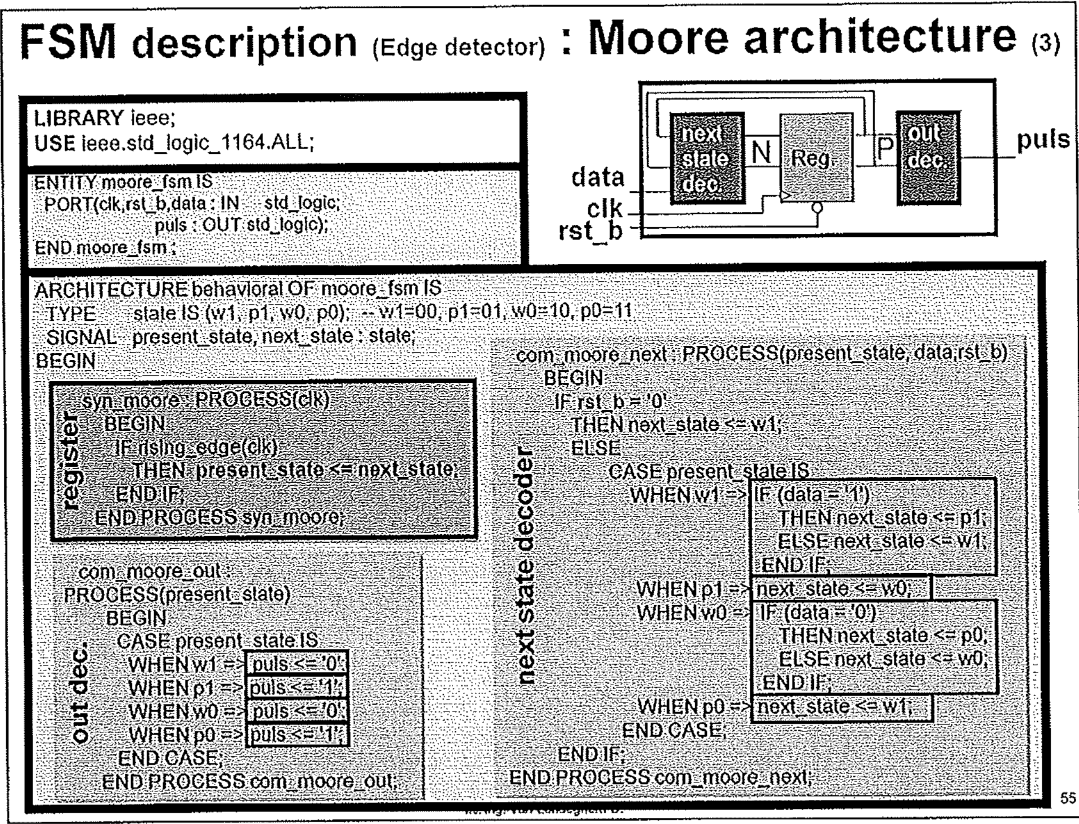
FSM description: TWO PROCESS METHOD

Combinatorial process

In the process comb_moore the value of next_state is determined by a function of the present_state and the input data. Thus, this combinatorial process is sensitive to these signals. Within this process the FSM outputs and transitions are defined per state in a case-when statement. The first case (when condition) is for the w1 state. The output puls and the transitions from state w1. There are two transistons for w1: (1) transition to p1 if data is asserted or (2) remain in w1 state. Coding of the remaining states requires the same procedure: Create a when statement for each state (when state_name =>), specify that state's outputs and define the state transitions with if-then-else statements.

Synchronous process (memory)

This process syn_moore indicates when the next_state becomes the present_state. This happens synchronously on the rising edge of the clock.



Consider again the example of the positive and negative edge detector.

FSM description: THREE PROCESS METHOD

This FSM description has a full equivalence with the hardware reality. The process com_moore in the previous FSM description is split into two processes com_moore_next and com_moore_out. The process com_moore_next describes only the next_state logic, while the process com_moore_out only describes the output decoder logic.

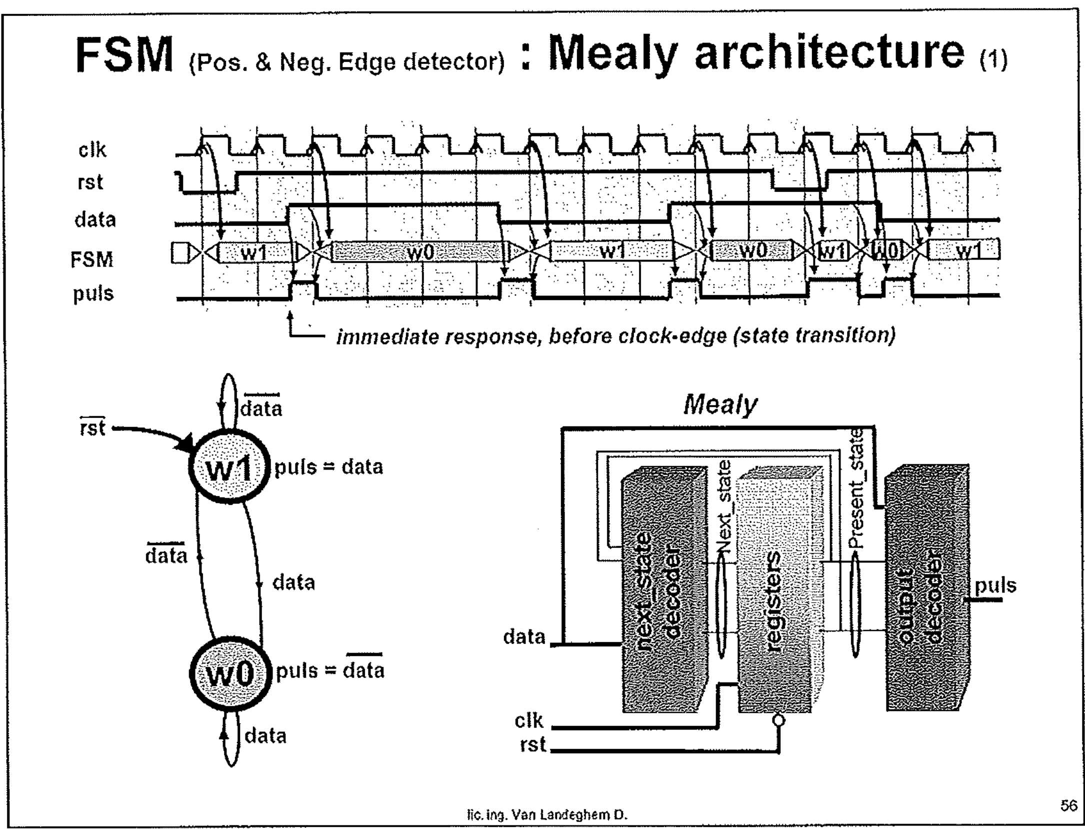
Enumerated tupe – state assignment

As an alternative, declare an enumeration type state, consisting of the state names (w1, p1, w0, p0) and two signals present_state and next_state of that type state.

Notice, that the case-when statement in both processes com_moore_next and com_moore_out has no WHEN OTHERS part because the type of present_state is enumerated. There are in this case no other possible values for present_state than w1, p1, w0, p0. The consequence of defining the enumeration type state is in this case a different state assignment. The elements in an enumeration type are numbered from the left to the right, and so are the element values. This means that the value of the last two states (w0=10 and p0=11) are different from the values they had in the previous example (w0=11 and p0=10). It may be quite obvious that declaring TYPE state IS (w1, p1, p0, w0) should result in the same state assignment as in the previous architecture. The state assignment determines the efficiency of the synthesized hardware (an smaler input and next_state logic).

Reset

Concerning the rst signal, there is a difference between this architecture and the previous one, that has nothing to do with the three-process FSM description style. Since the reset operation is synchronously with the rising edge of the clock, the reset can also be described in the input and next-state decoder process com_moore_next instead of process syn_moore.



In Moore machines the outputs are strictly functions of the current state.

Mealy machines may have outputs that are function of the present_state and the present input signal(s).

Lets make a Mealy version of the positive and negative edge detector. Analyzing the problem (timing diagram), there are 2 different situations:

(1) when data transitions from '0' to '1' puls changes directly to '1' and returns to '0' based on a rising edge of the clock.

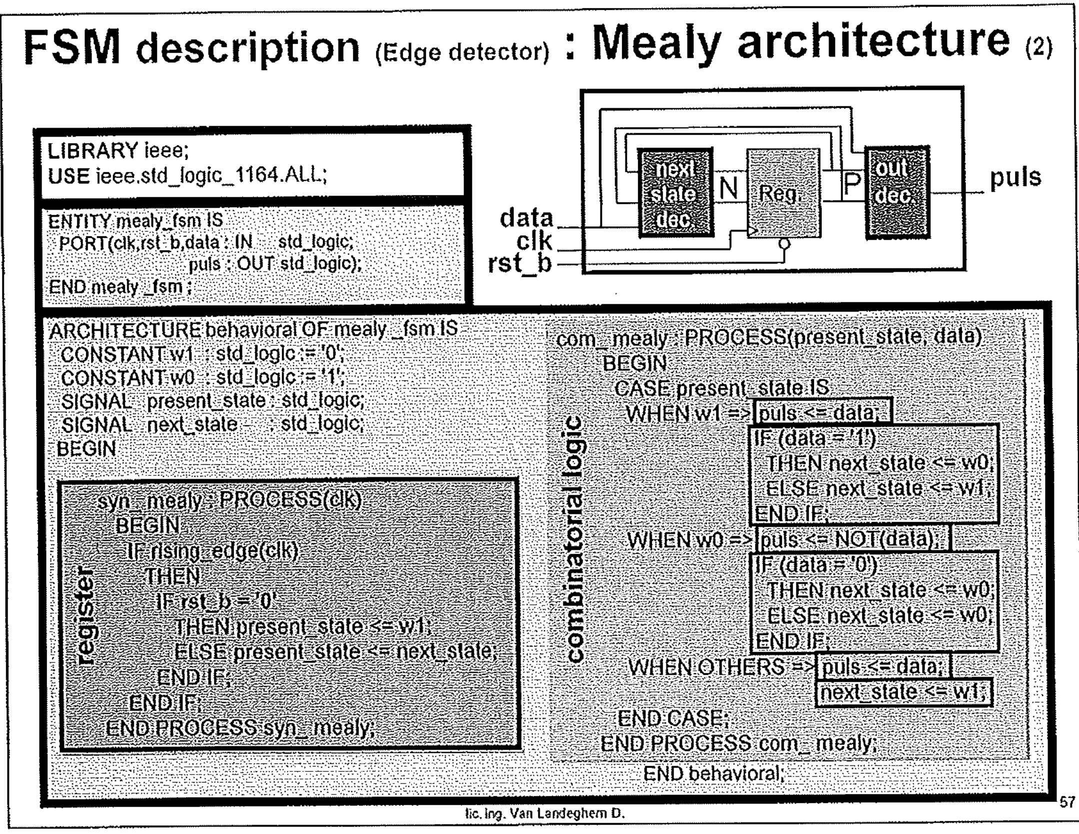
(2) when data transitions from '1' to '0' puls changes directly to '1' and returns to '0' based on a rising edge of the clock.

Now an FSM is needed with only 2 states (w1 and w0). The state transitions depend on the input signal data and the present state. When the FSM is in the initial state w1 (after a reset for instance), data must be '1' to transition to state w0. If data is not '1', there will be no transition and the FSM remains in w1. Once the FSM is in state w0, data must be '0' to transition to state w1, if data is not '0' the state w0 doesn't change. The principle of the state transitions are similar to the Moore version of the design.

Notice, that besides an asserted condition, an FSM requires a positive edge on the clock to transition from one state to an other.

The difference is seen in the output decoder of the FSM. The output puls depends on the present_state (w1 or w0) and on the value of the input data.

This results in writing puls = data in state w1 and in writing puls = not data in state w0. The pulse width of the signal puls can vary from about 0 to 1Tclock. This is because the start of puls depends on the transition of the input data to an active value and the end depends on the rising edge of the clock, since there is a state transition required to get an other output equation for puls.

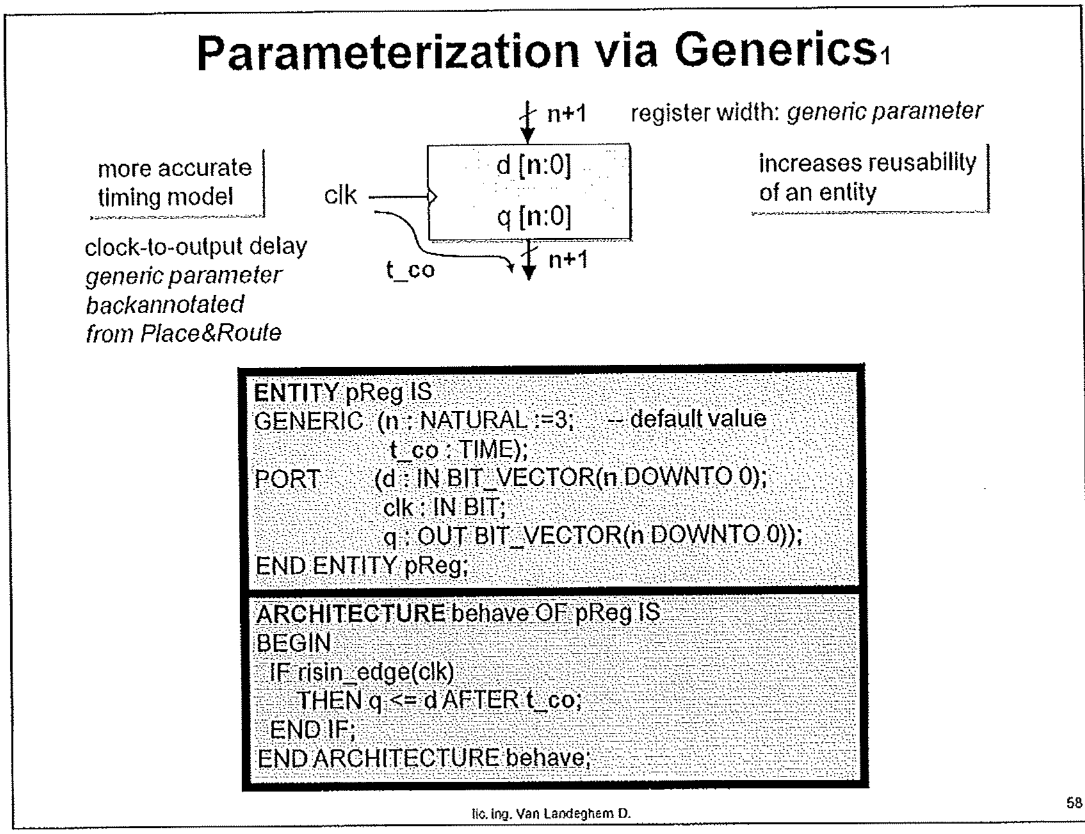


The additional work involved in describing Mealy machines versus Moore machines is minimal. In a Mealy machine an output is a function of both a state and an input. The output puls in process com_mealy depends on data and on the present_state, by writing « puls <= data; » when the present_state is w1. On the other hand « puls <= NOT(data); » when the present_state is w0.

The consequence of this is that output puls can change each time data will change. So the pulse width of the output puls can vary from about 0 to 1Tclock.

The output puls is not synchronous anymore with the clock signal when input data is an asynchronous signal. This could cause some problems like spike generation in the design.

If on the other hand data is a synchronous input, the puls-width of the output puls will be the same as the one in a Moore architecture, 1Tclock. The number of states required in the Mealy solution is in this case 1/2 of the number of states in the Moore solution. This means the number of FFs in a Mealy design can be reduced to 1FF without loosing the functionality of the Moore design. The only difference that maintains is the timing of the output puls. Puls is asserted 1Tclock earlier in the Mealy design versus the Moore design of the positive and negative edge detector.



VHDL includes a mechanism, generic constants, that allows components and entities to be parameterized with formal constants. Generic parameters are used to create parameterizable units in a design. Generic parameters may for example be used to define bus widths and delay parameters.

Definition of generics

GENERICs are defined like PORTs in the ENTITY definition.

The generic constant mechanism is widely used to specify design parameters and timing parameters, among other things.

Design parameters

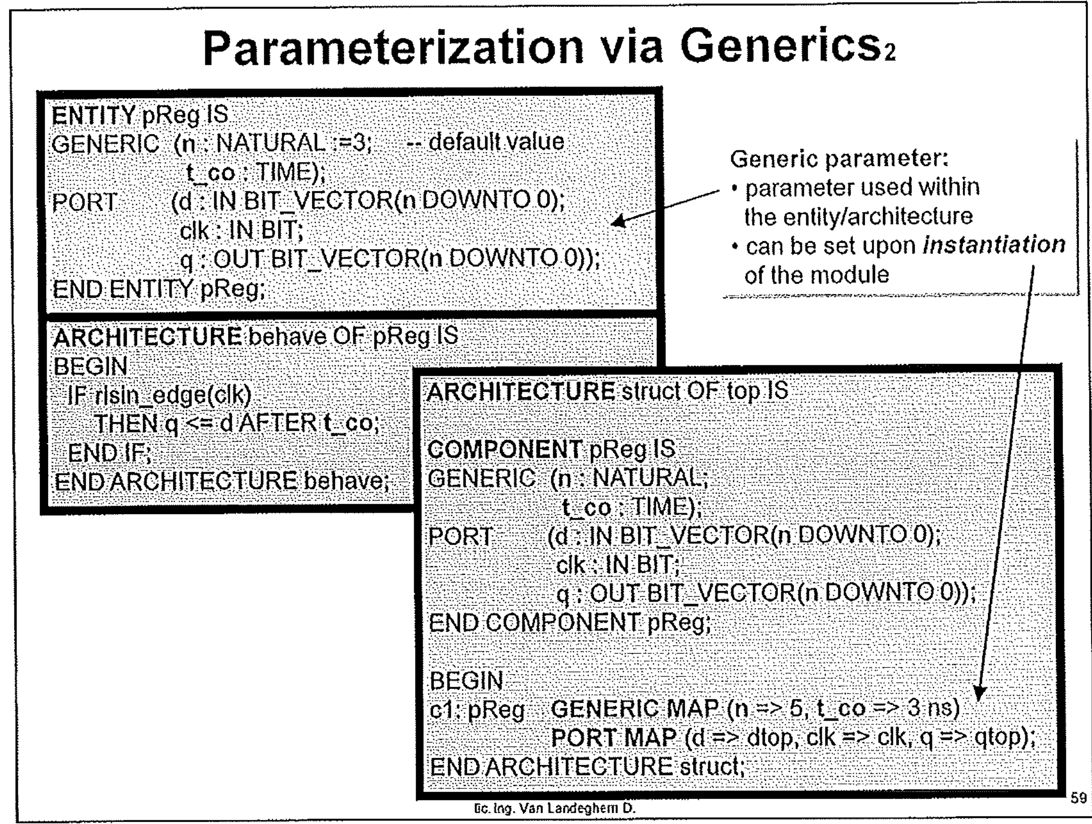
Typical parameters: array port bounds, bus widths, modulo values, ...

Reuse of a design unit can be improved by making it applicable in a wider set of contexts, by making it more generic. This done by using parameterization.

Timing parameters

Typical parameters: delays, set-up and hold times

Delays depend on the used technology, the design and the actual place&route. Timing parameters are therefore useful when generating backannotated VHDL files from synthesis and Place&Route tools. It makes the VHDL description more accurate for timing simulations.



Setting values of generics

Actual generic constants are specified when components are instantiated and when entities are bound. First when the unit shall be used the parameters must get values. Generics may be given a default value in the generic clause, which will be used if the generic is not explicitly assigned a value during instantiation.

Generic parameters are "connected" to values using a GENERIC MAP that functions just as a PORT MAP does for signals.

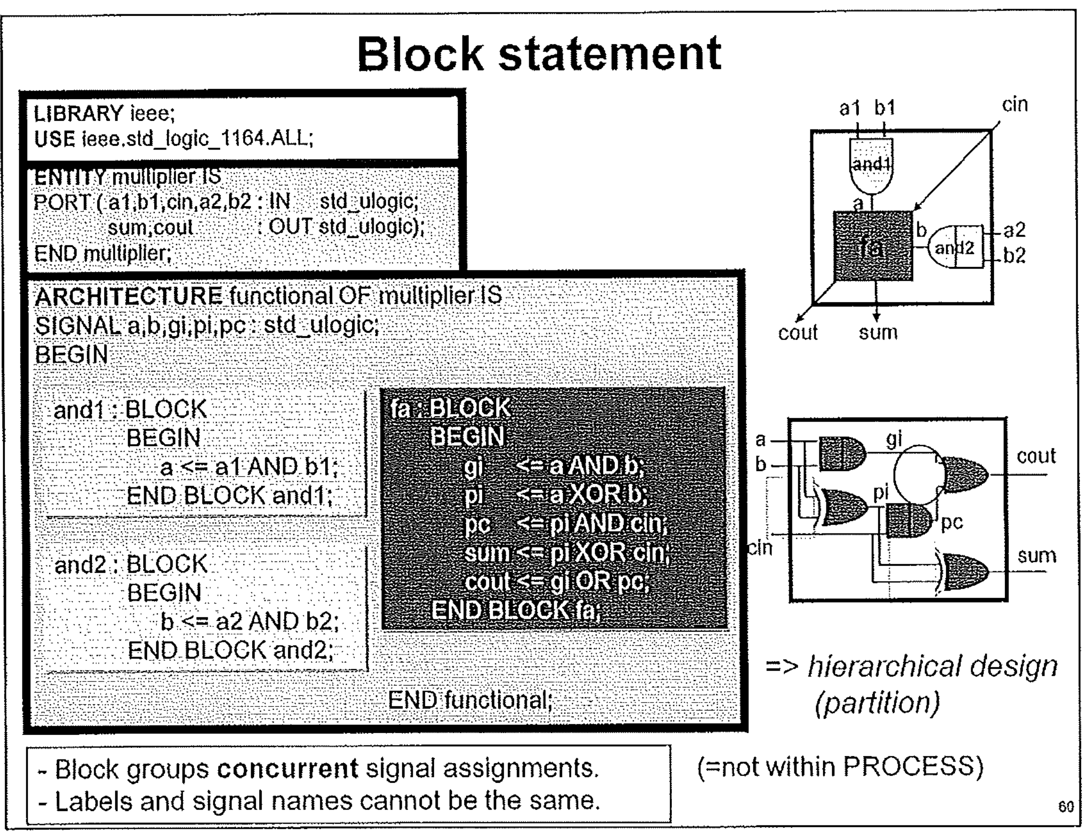
The same component can be instantiated several times with different generic parameters.

Generics are a general mechanism used to pass information to an instantiated entity.

· The data passed to an instance is static data. After the model has been linked to the simulator, the data does not change during simulation.

· The information contained in generics passed into a component instance or a block, can be used to alter simulation results, but results can not modify generics.

· Generics can not be assigned information as part of a simulation run.



Blocks are a *partitioning mechanism* within VHDL that allow the designer to logically group areas of a model description. The statement area in an architecture can be broken up into a number of separate logical areas by using blocks.

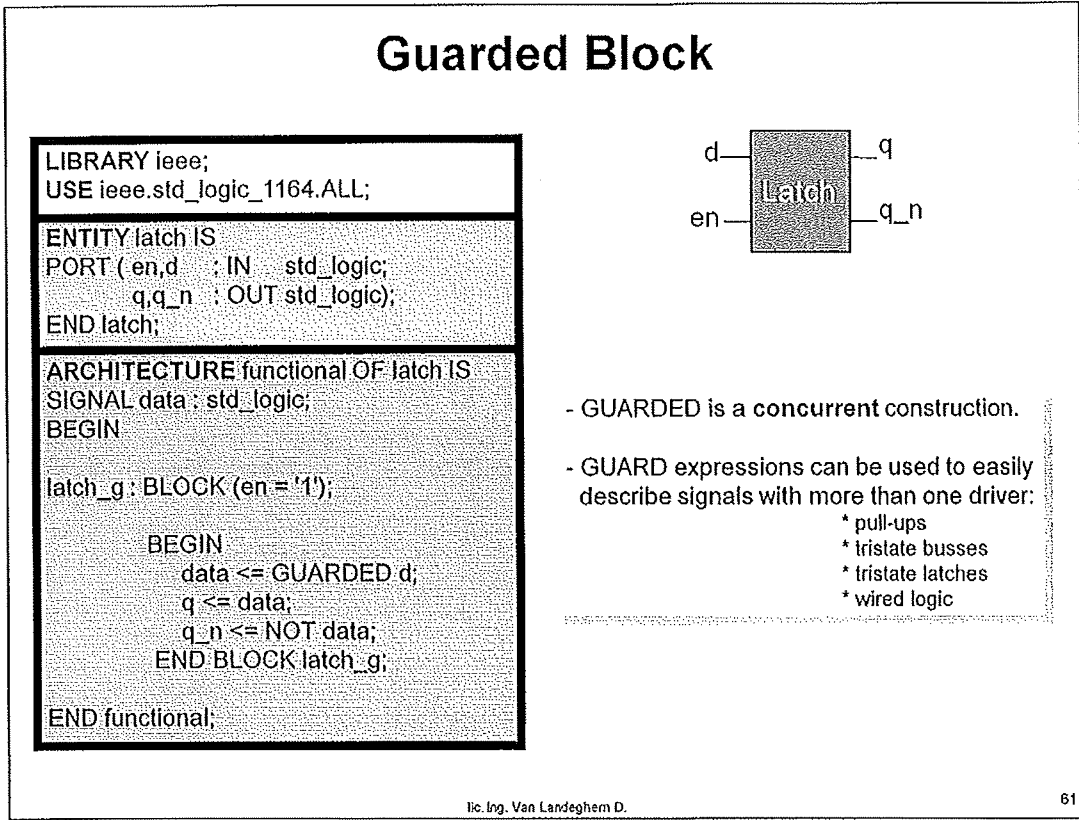
• A BLOCK has two purposes – to introduce a structure in the design and to be used in combination with *guarded signals*.

• A BLOCK must be named by a label.

• It is possible to declare a BLOCK inside another BLOCK. That creates a structure in the design. Block statements can be nested to create a design approach that can be referred to as an hierarchical design. A block statement groups concurrent statements, including processes. Inside a process block statements are excluded because processes group sequential statements.

A circuit may be specified by an entity showing its inputs and outputs and a description defining exactly how it operates. In terms of implementation, however, a circuit is composed of smaller building blocks that are interconnected. A complex digital system may contain millions of interconnected gates. Thus, no complex system can be designed simply by interconnecting gates one at a time. Also at the behavior level, a description can become to complex. In order to deal with such a complexity, a "divide and conquer" approach is used. The circuit is broken up into pieces, called blocks. The blocks are interconnected to form the circuit. The functions of the blocks and the interfaces between them are carefully defined, so that the circuit formed by interconnecting the blocks corresponds to the circuit specification. If a block is still too large, it can be broken up into smaller blocks. This process can be repeated as necessary.

The "divide and conquer" approach is illustrated in the simple example of the multiplier bit-module above. The multiplier can be decomposed into 3 blocks, 2 identical blocks each of which is one bit multiply (AND) function and one block to describe the full adder function. In practice, larger blocks, with a high level functionality, are used.



Guarded blocks

A guarded block contains a guard expression that can enable and disable drivers inside the block.

· The guard expression is a Boolean expression when *true*, drivers contained in the block are enabled, and when *false*, the drivers are disabled.

· Signal assignment can be guarded by using the keyword *guarded*

· Guarded block statements allow the capability of turning off drivers within a block.

Example (latch)

The guard expression in the example is (en='1'). When en='1', this return true an false when en='0'.

When the guard expression is true, 'all' of the drivers of guarded signal assignment statements are enabled or turned ON. When the guard expression is false all of the drivers of guarded signal assignment statements are disabled or turned OFF. The guarded signal does not change value.

. The signal data will change value when en = '1' and d changes value.

. The signal data will not change value when en = '0'.

Concatenation

```
a,b : IN    bit_vector (1 DOWNTO 0);
c   : OUT bit_vector (3 DOWNTO 0);
   c <= a  &  b;      -- c(3) <= a(1);    or   c(3 DOWNTO 2) <= a;
   ↑    ↑     ↑       -- c(2) <= a(0);
   |    |     |       -- c(1) <= b(1);    or   c(1 DOWNTO 0) <= b;
 4bits 2bits 2bits    -- c(0) <= b(0);
```

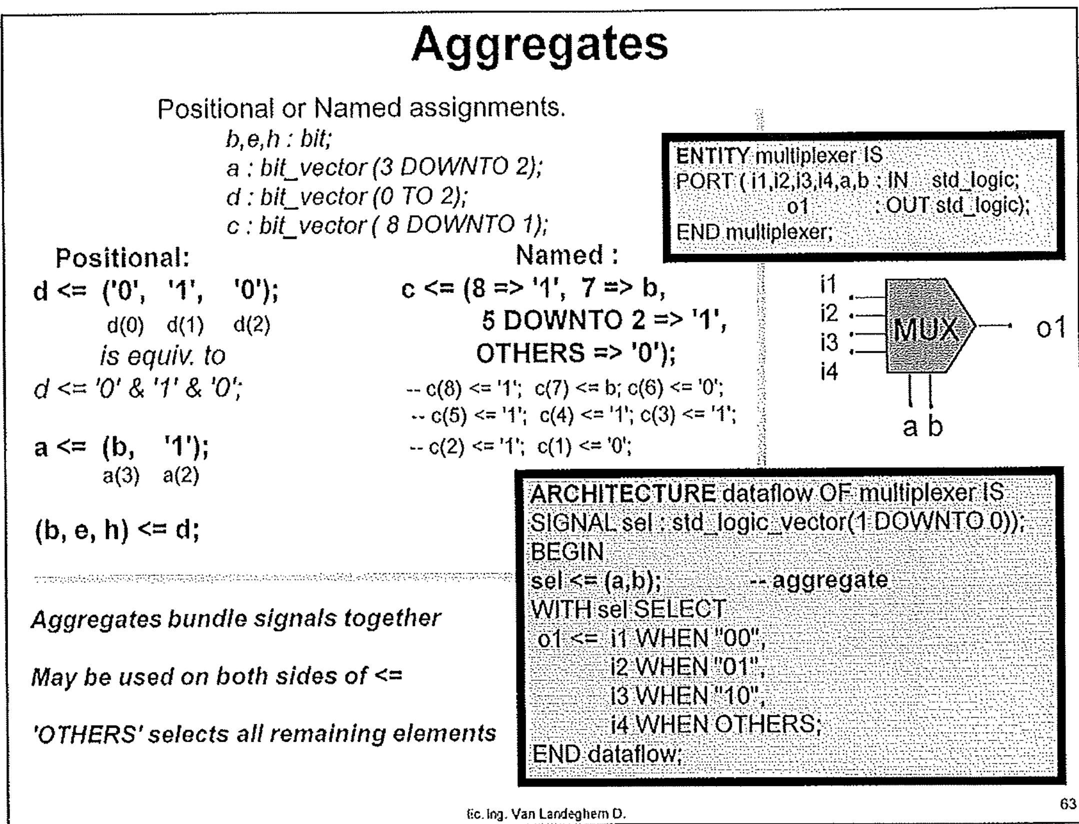
concatenation operator & only allowed at the right side of <=

```
ARCHITECTURE behav OF cnt IS
 SIGNAL  counter_state : std_ulogic_vector (7 DOWNTO 0);
BEGIN
 cntr : PROCESS (clk)
     BEGIN
     IF rising_edge(clk)
       THEN
         IF rst = '0'
           THEN counter_state <= (OTHERS => '0');
           ELSE counter_state <= counter_state + ("0000000" & '1');
         END IF;
     END IF;
 END PROCESS cntr;
END behav;
```

lic.ing. Van Landeghem D. 62

The & represents an operation called concatenation. A concatenation operator combined two signals into a single signal having its number of bits equal to the sum of the number of bits in the original signals. In the example a & b represents the signal vector a(1) a(0) b(1) b(0) with 2 + 2 = 4 signals. Note that a appears on the left in the concatenation expression and appears on the left in the signal listing of c (position!).

An application of the concatenation operator is showed in the architecture of the circuit counter.



An aggregate is a list of elements, enclosed in parentheses and separated by commas. The listed elements (for example a<=(b, '1'): signal names, signal values) are assigned to a signal with the same type and vector dimension, so the number of bits will be equal to the sum of bits in the list. The elements in the list can be assigned on a positional base or on a named base. On a positional base it is important to list the elements conform bit significance. On a named base the elements are assigned using index identification first, followed by the element (signal name or signal value).

Slices of Arrays

```
ARCHITECTURE illustration OF slices IS
  SIGNAL a_bus :           std_logic_vector (7 DOWNTO 0);
  SIGNAL b_bus, c_bus :    std_logic_vector (3 DOWNTO 0);
  SIGNAL d :               std_logic;

BEGIN
...
 b_bus <= a_bus (7 DOWNTO 4);

 a_bus (1 DOWNTO 0) <= '0' & d;

 d <= b_bus(0);

 a_bus (5 DOWNTO 0) <= b_bus;    -- WRONG!!

 a_bus (6 DOWNTO 4) <= b_bus (3 DOWNTO 1);
...
END illustration;
```

slices select elements of arrays

lic. ing. Van Landeghem D. 64

The inverse operation of concatenation and aggregation is the selection of slices of arrays, only a part of an array is to be used. The range of the desired array slice is specified in brackets and must be compatible with the range in the declaration of the signal.

Overloading

- Overloading: **using the *same name* for different functions**.

This can be done for:
* types
* subprograms (functions & procedures)
* operators as "+", "=", ...

- The compiler chooses the correct type, subprogram or operator from the types of the operands.

- Examples overloading

TYPES:

```
TYPE edge_state IS (up, down);
TYPE pos_state  IS (up, equal, down);
VARIABLE edge_detect : edge_state;
VARIABLE pos_detect  : pos_state;
         edge_detect := up;
         pos_detect  := up;
```

FUNCTIONS:

```
SIGNAL a,b,c  : bit_vector (7 DOWNTO 0);
         a <= to_bitvector([a real]);
         b <= to_bitvector([an integer]);
         c <= to_bitvector([a std_ulogic_vector]);
```

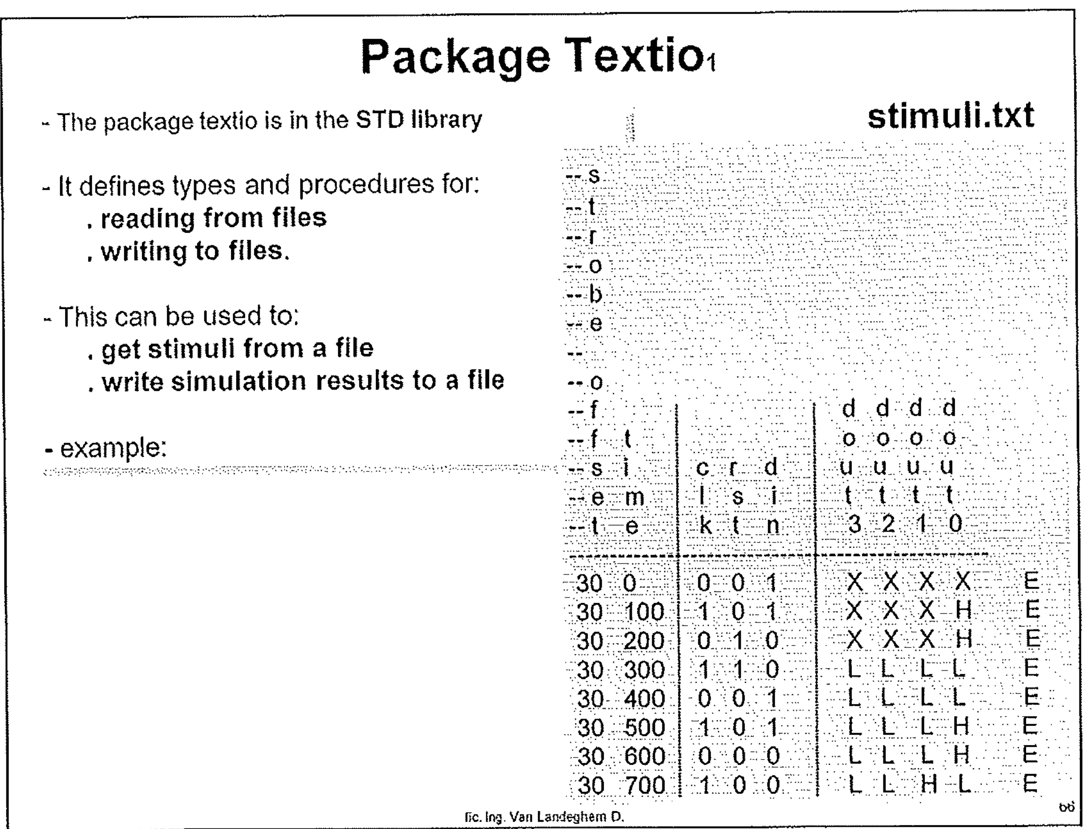
OPERATORS:

```
FUNCTION ">" (l,r : std_ulogic) RETURN std_ulogic;
FUNCTION ">" (l,r : std_ulogic) RETURN boolean;
         result <= a > b;
         IF ( a > b) THEN ...
```

lic. ing. Van Landeghem D. 65

Operator overloading

An operator symbol or subprogram name can be used more than once as long as calls are differentiable.



Test stimuli can be coded inside each test bench. The input test vectors are applied to the entity under test and after simulation the output vectors are observed in a waveform.

With this approach each change of the test stimuli implies a modification of the test bench description. The test bench has to be recompiled each time.

The use of input and output files can be used to get around this problem. With this approach it is possible to record the output vectors and compare them with those of the reference model. The same test bench can be used to run multiple tests simply by changing the input file (input test vectors).

Package Textio (read)2

FILE tvi_file: *TEXT* OPEN *READ_MODE* IS "tvi.txt";
→ opens file with type text (charaters) to read

READLINE(file_name, linebuffer_name)
→ reads an entire line in a file an places it in a line buffer

READ(linebuffer_name, value, good)
. reads from the line buffer a string (field) to a signal/variable (value)
. good is true when the type of the string matches
that of the corresponding signal/variable
. READ operation skips white space characters: space, tab.

file: **fred "cat" 1001010 12 -4.7 a 10 ns**

VHDL:

```
readline(file_id,L); -- read line buffer
read (L,s);   -- L is a string, s is a string
read (L,s);   -- L is a string, s is a string
read (L,bv);  -- bv is a std_logic_vector
read (L,i);   -- i is an integer
read (L,r);   -- r is a real
read (L,c);   -- c is a character
read (L,t);   -- t is a time
```

lic. ing. Van Landeghem D. 67

File handling

• A file may contain all types in VHDL except for files, access types (pointers) and multidimensional arrays.

• The VHDL standard does not define how data shall be stored in a file. It is therefore preferable to use text files (type TEXT) since they are easy to read and since there is a number of predefined procedures to handle them. The procedures are defined in the package TEXTIO. By using this standardized package it is possible to move the files between different simulation environments.

• With the declaration of the file (FILE: file_name) the type is defined. The file can be opened (OPEN) to be read (READ_MODE) or to be written (WRITE_MODE).

File reading (TEXTIO)

• When reading a text file, a whole line must first be read. That is done using READLINE. After that, each element in the line is read using a number of READ operations. Each object assigned by a value from the file must be of the same type as the value. It is therefore important to know the order of the elements in the file.

• Every READ operation skips white space characters: space, tab.

• The predefined file INPUT reads a value from the keyboard or from an input file during simulation. The handling of INPUT is tool dependant.

Package Textio (write)3

FILE tvo_file: *TEXT* OPEN *WRITE_MODE* IS "tvo.txt";
→ opens file with type text (charaters) to write

WRITELINE(file_name, linebuffer_name)
→ writes an entire line buffer to a file

WRITE(linebuffer_name, value, justified, field)
. writes a string (signal/variable - value) to the line buffer
. width (#characters) of the field and justification (left/right) in this field

VHDL:

```
write (L,i,left,5);        -- L is a line, i = 45
write (L,' ');             -- insert space
write (L,i,right,5);
write (L,' ');
write (L,STRING'("fred"));
write (L,' ');
write (L,STRING'("101"));
write (L,' ');
write (L,r,left,3);        -- r = 3.14159
write (L,' ');
write (L,t,left,0,ms);     -- t = 23 micro secs
writeline(file_id,L);      -- write line L
```

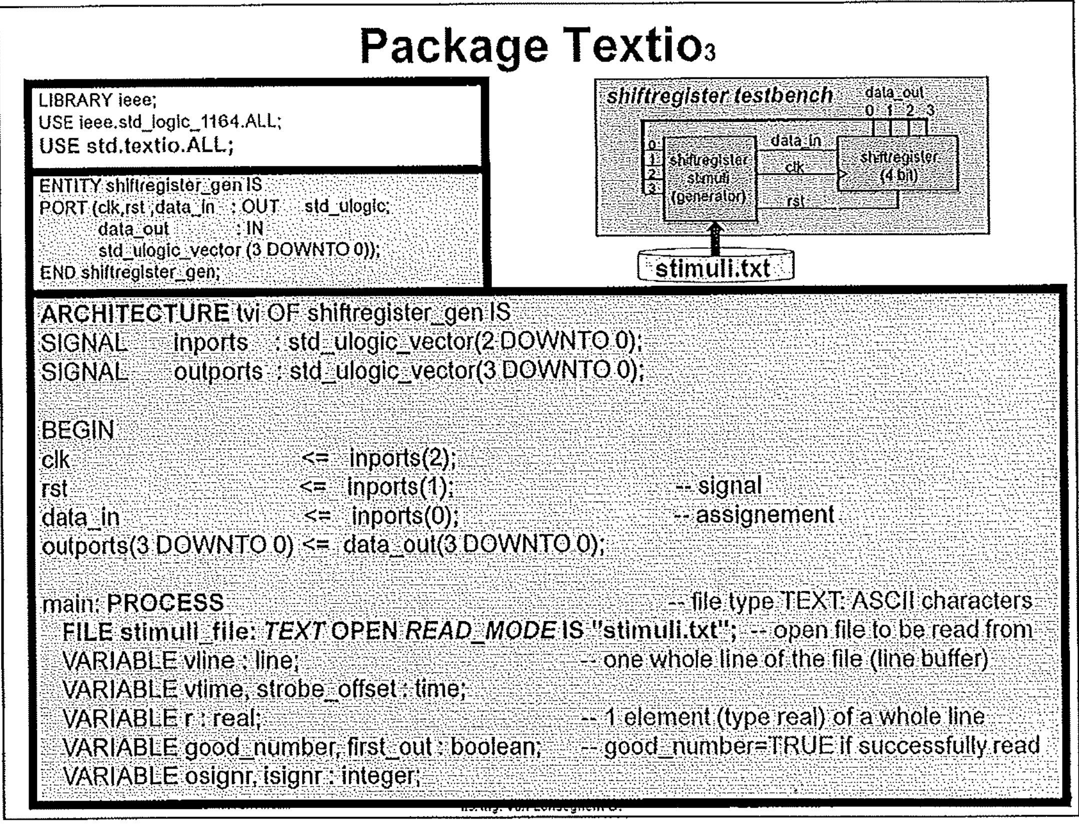
file: **45bbb bbb45 fred 101 3.1 0.023 ms** (b = blanc)

68

File writing (TEXTIO)

• When writing text to a text file, all elements are first written to a line using WRITE and finally the whole line is written to the file using WRITELINE.

• The predefined file OUTPUT writes to the screen or to an output file during simulation. No information is given about the current simulation time as it is when using REPORT. The handling of OUTPUT is tool dependant.



A very useful example of the application of file I/O is the construction of testbenches.

Example:

The testbench reads test inputs from a file, applies them to the model under test, and records model outputs for analysis.

Testbench for the shiftregister

- USE clause std.textio.ALL makes the package TEXTIO visible.
- Signals are declared to match the ports of the DUT (device under test).
- Test Vector Inputs (tvi) are available in text file stimuli.txt. These vectors will be read into the test bench and applied to the shiftregister.
- Expected Test Vector Outputs (tvo) are also available in text file stimuli.txt. The actual value will be compared with the expected one. When they differ, a REPORT will be written.

Package Textio4

```
BEGIN
LL: WHILE NOT endfile(stimuli_file) LOOP          -- if not end of file stimuli_file
      readline(stimuli_file , vline);              -- read a whole line to vline

      read(vline , r , good_number);               -- read 1 field of line buffer (vline) to r
      NEXT WHEN NOT good_number;                   -- skip line if field read is not real
      strobe_offset := r * 1ns;                    -- convert to time
      read(vline , r , good_number);
      NEXT WHEN NOT good_number ;                  -- skip line if field read is not real
      vtime := r * 1ns;                            -- convert to time
      IF (NOW < vtime)                             -- NOW: returns simulation time
         THEN WAIT FOR vtime - NOW;
      END IF;
      first_output := true;
      isignr := 2;                                 -- number of input signals
      osignr := 3;                                 -- number of output signals

  LC: FOR i IN vline'RANGE LOOP                    -- analyze line buffer vline
            CASE vline(i) IS
               WHEN '0' => inports(isignr) <= '0';           -- input drive value '0'
                           isignr := isignr - 1;
               WHEN '1' => inports(isignr) <= '1';           -- input drive value '1'
                           isignr := isignr - 1;
```

lic. ing. Van Landeghem D.

LOOP

• WHILE loops loop as long as a BOOLEAN expression is true.

• FOR loops are in general synthesizable (in this testbench this is not the intention), but not WHILE loops.

• FOR loops loop according to a loop variable that shall be an integer or an enumerated type. The loop variable shall not be declared.

• • EXIT jumps out of the loop and NEXT goes directly to the next iteration, not executing any code between NEXT and END LOOP.

• It is useful to name loops since it then is possible to specify what loop to exit or iterate using EXIT or NEXT.

• It is not possible to affect the length of the steps in a FOR loop.

NOW

NOW = predefined function that returns simulation time

Package Textio5

```
WHEN 'H' => IF first_output
               THEN WAIT FOR strobe_offset;
                  first_output := false;
            END IF;
            ASSERT outports(osignr) = '1'            -- measure '1' ?
              REPORT "expected H and found L"
              SEVERITY warning;
            osignr := osignr - 1;
WHEN 'L' => IF first_output
               THEN WAIT FOR strobe_offset;
                  first_output := false;
            END IF;
            ASSERT outports(osignr) = '0'            -- measure '0' ?
              REPORT "expected L and found H"
              SEVERITY warning;
            osignr := osignr - 1;
WHEN 'X' => null;
WHEN '-' => EXIT LC;                                 -- stop loop next char
WHEN 'E' => EXIT LC;                                 -- go to next line
WHEN others => ASSERT false
                 REPORT "illegal character at pos. :" & vline(i)
                 SEVERITY error;
               EXIT LC;
      END CASE;
    END LOOP LC;                                     -- to next char in line buffer
END LOOP LL;                                         -- to next line in file
```

- EXIT jumps out of the loop and NEXT goes directly to the next iteration, not executing any code between NEXT and END LOOP.
- It is useful to name loops (in this case LL and LC) since it then is possible to specify what loop to exit or iterate using EXIT or NEXT.

Package Textio6

```
-- end of file reached

  WAIT FOR strobe_offset;
    ASSERT false
      REPORT "end of simulation"
      SEVERITY note;

   WAIT;                                    -- infinite PROCESS

END PROCESS main;
END tvi;
```

lic. ing. Van Landeghem D. 72

WAIT statement

• The WAIT statement has three conditional parts that may be combined:

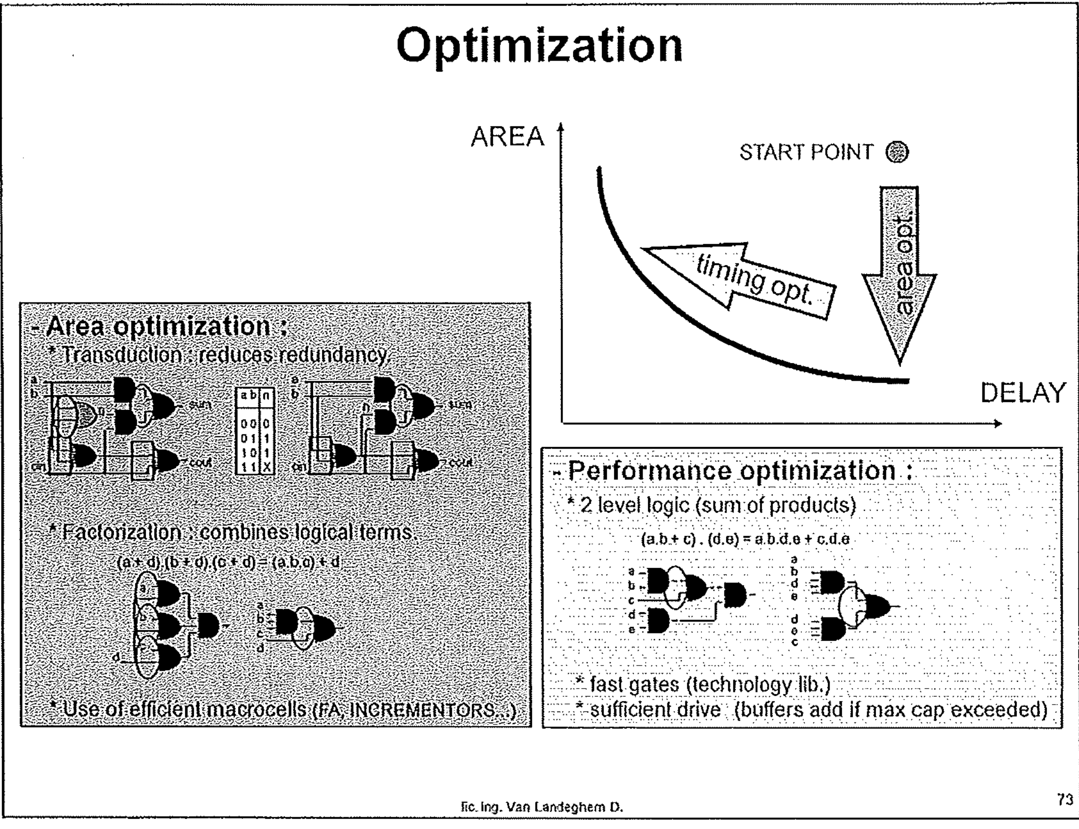
- ON: detects value changes
- UNTIL: verifies a logical expression
- FOR: limits in time (timeout).

• A WAIT statement may exist without a condition and will then never be passed.

• A variable does not have an *event* and does therefore not work in a WAIT ON statement. For the same reason expressions without signals do not work in a WAIT UNTIL statement. Such WAIT statements are suspended forever.

• Most synthesis tools accept just one WAIT statement for each process. The number is unlimited for simulation.

• At simulation start every process executes until it reaches its first WAIT statement.



The optimization process depends on three things:

. the form of the Boolean expressions

. the type of technolgy available (CPLD architecture or FPGA architecture)

. automatic or user applied synthesis directives (sometimes called constraints).

Optimizing for CPLDs usually involves reducing the logic to a minimal sum of products, which is then further optimized for a minimal literal count. This reduces the product-term utilization and the number of logic block inputs required for any given expression. These equations are then passed to the fitter for further device-specific optimization.

Optimizing for FPGAs typically requires that the logic is expressed in forms other than a sum of products. Instead, equations may be factored based on device-specific resources and directive-driven optimization goals. The factors can be evaluated for efficiency of implementation.

Summary

- VHDL = strongly typed
- BUILT-IN types: BIT, BOOLEAN, INTEGER
- USER-DEFINED types: std_ulogic ... (packages)

  TYPE std_ulogic IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
- ENTITY can have several underlying ARCHITECUREs
- OPERATORS: logicals : not and or nand nor xor xnor

  relationals : = /= < <= > >=

  arithmetic : ** * / mod rem + - &
- CONCURRENT statements (WHEN, WITH-SELECT):

  executed parallel, concurrent signal assignment is place in-dependent
- PROCESS:
  - . *sensitivity list: - combinatorial logic: all inputs*

    - sequential logic: only clk & async reset
  - . statements (IF, CASE, FOR, ...) inside PROCESS: SEQUENTIAL
  - . PROCESS itself = CONCURRENT statement
- **VARIABLEs** ⟺ **SIGNALs**

VARIABLEs	SIGNALs
. are LOCAL to PROCESSES	. can be used between PROCESSES
. cannot be used as ports	. can be used as ports
. are assigned instantaneously (:=)	. are assigned before a WAIT statement (<=)
. have no drivers (not scheduled)	. have drivers (scheduled)