

VHDL

VHDL'87/'93
en voorbeelden

Bert Molenkamp

The appendices C, D, E, and F are reprinted from IEEE Std 1076-1993 IEEE Standard VHDL Language Reference Manual, Copyright © 1994 by The Institute of Electrical and Electronics Engineers, Inc. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner. Information is reprinted with the permission of the IEEE.

The quick reference cards in this book, appendices I and J, are reprinted with the permission of QUALIS Design Corporation, Beaverton, USA.

VHDL,
VHDL'87/'93 en voorbeelden

© 1997 ir. E. Molenkamp

All rights reserved.

No part of this book may be reproduced in any form or by any means without permission in writing from the author. The appendices C, D, E, and F are property of the IEEE. The appendices I and J are property of QUALIS Design Corporation.

ISBN 90-802634-3-5

Universiteit Twente
Informatica
Postbus 217
7500 AE Enschede
Nederland

Inhoud

1. INLEIDING.	1
1.1 Eigenschappen van VHDL.	4
2. INLEIDING IN DE BESCHRIJVINGSTAAL VHDL.	5
2.1 Introductie.	5
2.2 Communicerende processen.	5
2.3 Simulatiemodel.	9
2.4 Procesbeschrijving.	12
2.4.1 Expliciete processen.	13
2.4.2 Impliciete processen.	21
2.5 Structuurbeschrijving met VHDL.	23
2.5.1 Ingangen, uitgangen en bidirectionele poorten.	25
2.5.2 Regelmatige structuren.	26
2.5.3 Koppeling van actuele met formele parameters.	27
2.6 Data types.	28
2.6.1 Scalar types.	28
2.6.2 Composite types.	30
2.6.3 Subtypes.	33
2.7 Operatoren.	34
2.8 Attributen.	35
2.8.1 Attributen voor arrays.	35
2.8.2 Enkele attributen voor signalen.	36
2.9 Overloading van functies en procedures.	38
2.10 Package en package body.	40
2.10.1 Component declaraties in een package.	43
2.10.2 Deferred constant declaration.	43
2.11 Libraries.	43
2.12 Samenvatting.	45
3. VERDIEPING IN ONDERWERPEN VAN VHDL.	47
3.1 Analyse, 'Elaboration' en Simulatie.	47
3.1.1 Analyse volgorde	47
3.1.2 Elaboratie en simulatie.	49
3.2 Configuratie-declaratie.	49
3.3 Modelleren van vertraging.	53

3.4	Procedures en functies.	55
3.4.1	Procedures en het gebruik van signalen.	56
3.4.2	Procedure met wait statements.	57
3.4.3	Een functie mag geen wait statements bevatten.	58
3.4.4	Concurrent procedure call.	58
3.5	Signaal met meerdere sources.	59
3.5.1	Resolved signal.	60
3.5.2	Modelleren van een bidirectionele bus.	61
3.5.3	Guarded signal.	62
3.5.4	Modelleren van een bus tussen processor en geheugen.	65
3.5.5	Nogmaals de resolutiefunctie.	69
3.6	Het block statement.	71
3.6.1	De relatie tussen een block en instantiatie van een component.	72
3.6.2	Block met een boolean expressie (de guard).	74
3.7	Passieve processen in de entity-declaratie.	77
3.8	Access types.	78
3.9	Files.	80
3.9.1	Geformateerde file i/o.	80
3.9.2	Tekst file i/o.	81
3.10	Koppeling van niet compatibele types in component instantiaties	82
3.11	Mode linkage	87
3.12	Attributen gedefinieerd door de gebruiker	88
3.13	Samenvatting.	89
4.	VHDL 1076-1993	91
4.1	Onderhoud aan de standaard.	91
4.2	Aanpassingen in de syntax van de taal.	91
4.3	Globale variabelen.	92
4.3.1	Generatie van random getallen.	94
4.4	Pure en impure functions.	95
4.5	Group.	96
4.6	Bit string literals.	96
4.7	Component instantiatie.	96
4.8	Expressies in de association list in de port map.	97
4.9	Configuration.	97
4.10	Nieuwe operatoren.	98
4.10.1	Xnor operator.	98
4.10.2	Schuif operatoren.	98

4.11	Labels.	99
4.12	Report statement.	99
4.13	Uitbreiding van het delay mechanisme.	99
4.14	Postponed process.	100
4.15	Concurrent conditional signal assignment statement.	100
4.16	Unaffected.	101
4.17	Generate statement.	101
4.18	Karakterset.	102
4.19	Extended identifiers.	102
4.20	Alias.	102
4.21	File.	103
4.22	Attributen.	104
4.22.1	Attributen verwijderd.	104
4.22.2	Attributen toegevoegd.	104
5.	VAN EEN VHDL BESCHRIJVING NAAR EEN REALISATIE	107
5.1	Wat is niet synthetiseerbaar?	108
5.1.1	Dynamische structuren, recursie en loops.	108
5.1.2	Expliciet opgegeven vertragingstijden	109
5.1.3	Asynchrone schakelingen	110
5.1.4	Wait statements	110
5.1.5	Dynamische structuren, recursie en files	111
5.1.6	Typering op "bit niveau"	111
5.1.7	Het event-driven simulatiemodel	114
5.1.8	Initiële waarden	114
5.2	Wat is synthetiseerbaar	115
5.2.1	Datatypes	115
5.2.2	Expliciete procesbeschrijving	116
5.2.3	Proces met een sensitivity list	117
5.2.4	Proces met een wait until statement	117
5.2.5	Impliciete procesbeschrijving	120
5.2.6	'Standaard' omgeving	120
5.2.7	Toestandsmachines en de synthese er van.	122
5.2.8	IEEE Synthesis Package	124
5.3	'Hoog niveau' synthese	125
5.3.1	Meerdere 'wait until' statements in een beschrijving	125
5.3.2	Samennemen van operaties in een beschrijving	125
5.3.3	Resource sharing	126
5.4	Samenvatting	126

6. EENVOUDIGE VOORBEELDEN.	129
6.1 Welke input bevat meer enen?	129
6.1.1 Inleiding.	129
6.1.2 Specificatie in VHDL.	129
6.1.3 Implementatie.	130
6.1.3.1 Bitcompare.	131
6.1.3.2 Opteller.	131
6.1.3.3 Structuurbeschrijving.	132
6.1.3.4 Synthese-resultaat van gedrag en structuur vergeleken.	132
6.1.3.5 Sequentiële oplossing.	133
6.2 Recursief filter.	135
6.3 Cache.	137
6.3.1 Inleiding.	137
6.3.2 Set-associatieve cache.	138
6.3.3 VHDL beschrijving van de set-associatieve cache.	138
6.4 Random generator.	147
6.4.1 Inleiding.	147
6.4.2 Beschrijving van een pseudo-random generator.	148
6.5 Scheduler.	149
6.5.1 Probleemstelling.	149
6.5.2 Specificatie in VHDL.	150
6.5.3 Implementatie in VHDL.	150
6.5.3.1 Het gebruik maken van priority encoders.	150
6.5.3.2 Een iteratieve structuur.	152
6.5.3.3 Het gebruik maken van een pla.	153
6.5.4 Samenvatting van de resultaten van de scheduler.	156
6.6 Simulatiemodel van een flipflop.	157
6.6.1 Inleiding.	157
6.6.2 Vergelijking van de simulatiesnelheid.	157
7. HET ONTWERPEN VAN EEN EENVOUDIGE PROCESSOR.	159
7.1 Inleiding.	159
7.2 Specificatie van de processor.	159
7.3 Hoe moet het gewenste gedrag geïmplementeerd worden?	162
7.3.1 De testomgeving.	163
7.4 De opdeling in een data-pad en een besturing.	165
7.4.1	166
7.4.1 Het data-pad.	166
7.4.2 De besturing.	168
7.4.2.1 De specificatie van de besturing.	169
7.4.2.2 Een toestandsmachine beschrijving voor de besturing.	171
7.4.2.3 Microprogrammering toegepast voor de besturing.	173
7.5 De realisatie.	176

7.6 De gebruikte packages.	178
7.6.1 Package control_names.	178
8. SPECIFICATIE VAN EEN PROCESSOR.	179
8.1 Informele beschrijving van de processor.	179
8.2 Het geheugen.	183
8.3 De formele beschrijving van de processor.	184
8.3.1 Package <i>utilities</i> .	184
8.3.2 Package <i>processor_types</i> .	185
8.3.3 Beschrijving van het gedrag van de processor.	186
8.4 Interactie tussen de processor en het geheugen.	190
A. DOCUMENTEN OVER VHDL.	193
A.1 Selectie van de uitgaven van de IEEE.	193
A.2 Boeken.	193
A.3 Conferenties.	196
A.4 Nieuwsbrieven.	196
A.5 Documenten van ESA.	196
B. VHDL AKTIVITEITEN.	198
C. GERESERVEERDE WOORDEN.	200
D. SYNTAX VHDL STD. 1076-1993.	201
E. PACKAGE STANDARD.	218
F. PACKAGE TEXTIO.	225
G. IEEE PACKAGE MATH_REAL.	229
G.1 Predefined Constants.	229
G.2 Predefined Functions.	229
G.3 Overloaded Operators.	230
G.4 Predefined Procedure.	230
H. IEEE PACKAGE MATH_COMPLEX.	231
H.1 Predefined Types.	231

H.2	Predefined Constants.	231
H.3	Predefined Functions.	231
H.4	Overloaded Operators.	231
I.	VHDL QUICK REFERENCE CARD	233
J.	VHDL PACKAGES QUICK REFERENCE CARD	237
K.	INDEX.	241

Voorwoord

Dit boek behandelt de IEEE/ANSI standaard VHDL. Het is bedoeld voor ontwerpers van digitale systemen met programmeervaardigheden. Zowel de 'oude' VHDL standaard Std. 1076-1987 als de huidige standaard Std. 1076-1993 worden beschreven. Dit vooral omdat veel CAE omgevingen nog niet de nieuwe standaard ondersteunen. Nagenoeg alle aspecten van VHDL komen aan de orde en worden geïllustreerd met kleine voorbeelden. Ook een aantal complexere voorbeelden zijn opgenomen in dit boek. Tevens is aangegeven waar actuele informatie met betrekking tot VHDL te vinden is. Hoewel getracht is de vele aspecten van VHDL te behandelen vervangt dit boek niet de IEEE standaard. De IEEE standaard is als naslagwerk onmisbaar.

In dit boek is ook materiaal van anderen gebruikt. De IEEE heeft toestemming verleend voor de overname van delen van de VHDL standaard Std. 1076-1993. Naast de omvangrijke syntax beschrijving van VHDL blijkt er ook de behoefte te bestaan aan een 'quick reference card' van VHDL. QUALIS Design Corporation heeft toestemming gegeven om de door hen ontwikkelde 'quick reference cards' op te nemen in dit boek. De 'quick reference cards' zijn onvolledig en vervangen dus niet de syntax beschrijving van de VHDL standaard.

Dankzij de zinvolle opmerkingen van diverse mensen is de kwaliteit van dit boek zowel tekstueel als inhoudelijk verbeterd. Olaf Greve, student Informatica, heeft veel en zinvolle opmerkingen over zowel de tekst als de inhoud gegeven. John Mooijekind, docent aan de Hogeschool Rijswijk, Corrie Huijs en Jaap Hofstede, collega's, ben ik erkentelijk voor hun kritische en opbouwende opmerkingen.

Oktober 1997, Bert Molenkamp

Inhoudsopgave

1. INLEIDING.	1
1.1 Eigenschappen van VHDL.	4
2. INLEIDING IN DE BESCHRIJVINGSTAAL VHDL.	5
2.1 Introductie.	5
2.2 Communicerende processen.	5
2.3 Simulatiemodel.	9
2.4 Procesbeschrijving.	12
2.4.1 Expliciete processen.	13
2.4.2 Impliciete processen.	21
2.5 Structuurbeschrijving met VHDL.	23
2.5.1 Ingangen, uitgangen en bidirectionele poorten.	25
2.5.2 Regelmatige structuren.	26
2.5.3 Koppeling van actuele met formele parameters.	27
2.6 Data types.	28
2.6.1 Scalar types.	28
2.6.2 Composite types.	30
2.6.3 Subtypes.	33
2.7 Operatoren.	34
2.8 Attributen.	35
2.8.1 Attributen voor arrays.	35
2.8.2 Enkele attributen voor signalen.	36
2.9 Overloading van functies en procedures.	38
2.10 Package en package body.	40
2.10.1 Component declaraties in een package.	43
2.10.2 Deferred constant declaration.	43
2.11 Libraries.	43
2.12 Samenvatting.	45
3. VERDIEPING IN ONDERWERPEN VAN VHDL.	47
3.1 Analyse, ‘Elaboration’ en Simulatie.	47
3.1.1 Analyse volgorde.	47
3.1.2 Elaboratie en simulatie.	49
3.2 Configuratie-declaratie.	49
3.3 Modelleren van vertraging.	53
3.4 Procedures en functies.	55
3.4.1 Procedures en het gebruik van signalen.	56

3.4.2 Procedure met wait statements.	57
3.4.3 Een functie mag geen wait statements bevatten.	58
3.4.4 Concurrent procedure call.	58
3.5 Signaal met meerdere sources.	59
3.5.1 Resolved signal.	60
3.5.2 Modelleren van een bidirectionele bus.	61
3.5.3 Guarded signal.	62
3.5.4 Modelleren van een bus tussen processor en geheugen.	65
3.5.5 Nogmaals de resolutiefunctie.	69
3.6 Het block statement.	71
3.6.1 De relatie tussen een block en instantiatie van een component.	72
3.6.2 Block met een boolean expressie (de guard).	74
3.7 Passieve processen in de entity-declaratie.	77
3.8 Access types.	78
3.9 Files.	80
3.9.1 Geformateerde file i/o.	80
3.9.2 Tekst file i/o.	81
3.10 Koppeling van niet compatibele types in component instantiaties.	82
3.11 Mode linkage.	86
3.12 Attributen gedefinieerd door de gebruiker.	87
3.13 Samenvatting.	88
4. VHDL 1076-1993.	91
4.1 Onderhoud aan de standaard.	91
4.2 Aanpassingen in de syntax van de taal.	91
4.3 Globale variabelen.	92
4.3.1 Generatie van random getallen.	94
4.4 Pure en impure functions.	95
4.5 Group.	96
4.6 Bit string literals.	96
4.7 Component instantiatie.	96
4.8 Expressies in de association list in de port map.	97
4.9 Configuration.	97
4.10 Nieuwe operatoren.	98
4.10.1 Xnor operator.	98
4.10.2 Schuif operatoren.	98
4.11 Labels.	99
4.12 Report statement.	99
4.13 Uitbreiding van het delay mechanisme.	99
4.14 Postponed process.	100

4.15 Concurrent conditional signal assignment statement.	100
4.16 Unaffected.	101
4.17 Generate statement.	101
4.18 Karakterset.	102
4.19 Extended identifiers.	102
4.20 Alias.	102
4.21 File.	103
4.22 Attributen.	104
4.22.1 Attributen verwijderd.	104
4.22.2 Attributen toegevoegd.	104
 5. VAN EEN VHDL BESCHRIJVING NAAR EEN REALISATIE.	 107
5.1 Wat is niet synthetiseerbaar?	108
5.1.1 Dynamische structuren, recursie en loops.	108
5.1.2 Expliciet opgegeven vertragingstijden.	109
5.1.3 Asynchrone schakelingen.	110
5.1.4 Wait statements.	110
5.1.5 Dynamische structuren, recursie en files.	111
5.1.6 Typering op "bit niveau".	111
5.1.7 Het event-driven simulatiemodel.	114
5.1.8 Initiële waarden.	114
5.2 Wat is synthetiseerbaar.	115
5.2.1 Datatypes.	115
5.2.2 Expliciete procesbeschrijving.	116
5.2.3 Proces met een sensitivity list.	116
5.2.4 Proces met een wait until statement.	117
5.2.5 Impliciete procesbeschrijving.	120
5.2.6 'Standaard' omgeving.	120
5.2.7 Toestandsmachines en de synthese er van.	122
5.2.8 IEEE Synthesis Package.	124
5.3 'Hoog niveau' synthese.	125
5.3.1 Meerdere 'wait until' statements in een beschrijving.	125
5.3.2 Samennemen van operaties in een beschrijving.	125
5.3.3 Resource sharing.	126
5.4 Samenvatting.	126
 6. EENVOUDIGE VOORBEELDEN.	 129
6.1 Welke input bevat meer enen?	129
6.1.1 Inleiding.	129
6.1.2 Specificatie in VHDL.	129

6.1.3 Implementatie.	130
6.1.3.1 Bitcompare.	131
6.1.3.2 Opteller.	131
6.1.3.3 Structuurbeschrijving.	132
6.1.3.4 Synthese-resultaat van gedrag en structuur vergeleken.	132
6.1.3.5 Sequentiële oplossing.	133
6.2 Recursief filter.	135
6.3 Cache.	137
6.3.1 Inleiding.	137
6.3.2 Set-associatieve cache.	138
6.3.3 VHDL beschrijving van de set-associatieve cache.	138
6.4 Random generator.	147
6.4.1 Inleiding.	147
6.4.2 Beschrijving van een pseudo-random generator.	148
6.5 Scheduler.	149
6.5.1 Probleemstelling.	149
6.5.2 Specificatie in VHDL.	150
6.5.3 Implementatie in VHDL.	150
6.5.3.1 Het gebruik maken van priority encoders.	150
6.5.3.2 Een 1-dimensionaal array met een loop.	152
6.5.3.3 Het gebruik maken van een pla.	153
6.5.4 Samenvatting van de resultaten van de scheduler.	156
6.6 Simulatiemodel van een flipflop.	157
6.6.1 Inleiding.	157
6.6.2 Vergelijking van de simulatiesnelheid.	157
 7. HET ONTWERPEN VAN EEN EENVOUDIGE PROCESSOR.	 159
7.1 Inleiding.	159
7.2 Specificatie van de processor.	159
7.3 Hoe moet het gewenste gedrag geïmplementeerd worden?	162
7.3.1 De testomgeving.	163
7.4 De opdeling in een data-pad en een besturing.	165
7.4.1 Het data-pad.	166
7.4.2 De besturing.	168
7.4.2.1 De specificatie van de besturing.	169
7.4.2.2 Een toestandsmachine beschrijving voor de besturing.	171
7.4.2.3 Microprogrammering toegepast voor de besturing.	173
7.5 De realisatie.	176
7.6 De gebruikte packages.	178
7.6.1 Package control_names.	178

8. SPECIFICATIE VAN EEN PROCESSOR.	179
8.1 Informele beschrijving van de processor.	179
8.2 Het geheugen.	183
8.3 De formele beschrijving van de processor.	184
8.3.1 Package <i>utilities</i> .	184
8.3.2 Package <i>processor_types</i> .	185
8.3.3 Beschrijving van het gedrag van de processor.	186
8.4 Interactie tussen de processor en het geheugen.	190
 A. DOCUMENTEN OVER VHDL.	 193
A.1 Uitgaven van de IEEE.	193
A.2 Boeken.	193
A.3 Conferenties.	196
A.4 Nieuwsbrieven.	196
A.5 Documenten van ESA.	196
B. VHDL AKTIVITEITEN.	198
 C. GERESERVEERDE WOORDEN.	 200
 D. SYNTAX VHDL STD. 1076-1993.	 201
 E. PACKAGE STANDARD.	 218
 F. PACKAGE TEXTIO.	 225
 G. IEEE PACKAGE MATH_REAL (DRAFT).	 229
G.1 Predefined Constants.	229
G.2 Predefined Functions.	229
G.3 Overloaded Operators.	230
G.4 Predefined Procedure.	230
 H. IEEE PACKAGE MATH_COMPLEX (DRAFT).	 231
H.1 Predefined Types.	231
H.2 Predefined Constants.	231
H.3 Predefined Functions.	231
H.4 Overloaded Operators.	231

I.VHDL QUICK REFERENCE CARD.	233
I.1 Library Units.	233
I.2 Declarations.	233
I.2.1 Type Declarations.	233
I.2.2 Other Declarations.	233
I.3 Expressions.	234
I.3.1 Operators,Increasing Precedence.	234
I.4 Sequential Statements.	234
I.5 Parallel Statements.	234
I.6 Predefined Attributes.	235
I.7 Predefined Types.	235
I.8 Predefined Functions.	235
I.9 Lexical Elements.	236
J.1164 PACKAGES QUICK REFERENCE CARD.	237
J.1 IEEE 's STD_LOGIC_1164.	237
J.1.1 Logic Values.	237
J.1.2 Predefined Types.	237
J.1.3 Overloaded Operators.	237
J.1.4 Conversion Functions.	237
J.2 IEEE 's NUMERIC_STD.	237
J.2.1 Predefined Types.	237
J.2.2 Overloaded Operators.	237
J.2.3 Predefined Functions.	238
J.2.4 Conversion Functions.	238
J.3 IEEE 's NUMERIC_BIT.	238
J.3.1 Predefined Types.	238
J.3.2 Overloaded Operators.	238
J.3.3 Predefined Functions.	238
J.3.4 Conversion Functions.	238
J.4 Synopsys ' STD__LOGIC_ARITH.	238
J.4.1 Predefined Types.	238
J.4.2 Overloaded Operators.	238
J.4.3 Predefined Functions.	238
J.4.4 Conversion Functions.	239
J.5 Synopsys 'STD_OGIC_MISC.	239
J.5.1 Predefined Functions.	239
J.6 Synopsys ' STD__LOGIC_UNSIGNED.	239
J.6.1 Overloaded Operators.	239
J.6.2 Conversion Functions.	239
J.7 Synopsys ' STD__LOGIC_SIGNED.	239

J.7.1 Overloaded Operators.	239
J.7.2 Conversion Functions.	239
J.8 Synopsys ' STD__LOGIC_TEXTIO.	239
J.8.1 Overloaded Operators.	239
J.9 Cadence 's STD_LOGIC_ARITH.	239
J.9.1 Overloaded Operators.	239
J.9.2 Predefined Functions.	239
J.9.3 Conversion Functions.	240

1. Inleiding.

Na de ratificatie door de IEEE van de hardware beschrijvingstaal VHDL Std 1076-1987 heeft het gebruik ervan een enorme vlucht genomen in de industrie. IEEE standaarden moeten minimaal om de vijf jaar opnieuw geratificeerd worden. Inmiddels is de standaard dan ook vervangen door de Std 1076-1993. Beide worden ook vaak aangeduid met VHDL'87 en VHDL'93, de laatste soms ook wel met VHDL'92. Dit is het oorspronkelijk geplande jaar van de nieuwe standaard. Voor alle duidelijkheid er is slechts één VHDL standaard namelijk de meest recente VHDL 1076-1993.

Helaas is de ondersteuning van de VHDL'93 door de na vier jaar na ratificatie nog minimaal. Aangezien verwacht mag worden dat de VHDL'87 voorlopig nog wel de standaard zal blijven die elke tool ondersteund begint dit boek met de beschrijving van de 'oude' VHDL standaard, gevolgd door een beschrijving van de huidige standaard. De voorbereidingen voor de Std. 1076-1998 (VHDL'98) is in een vergevorderd stadium. VHDL'98 zal voornamelijk de fouten corrigeren die voorkomen in VHDL'93. Daarnaast wordt nu al gewerkt aan "VHDL 200X", de opvolger van de VHDL'98 standaard. Deze standaard zal naar alle waarschijnlijk wel aanzienlijke wijzigingen bevatten.

Dit boek probeert een zeer groot deel van de standaard te behandelen waarbij gebruik gemaakt is van veel voorbeelden. Een subset van de VHDL beschrijvingen is ook synthetiseerbaar. Synthese aspecten maken echter geen deel uit van de VHDL standaard. Helaas beperken nog veel gebruikers van VHDL zich tot een subset van VHDL welke synthetiseerbaar is door haar/zijn synthese-tool en gaan daarbij voorbij aan de kracht van VHDL als specificatietaal van het nog te ontwerpen systeem.

VHDL is een levende taal. Zoals al is vermeld wordt de standaard tenminste elke vijf jaar opnieuw geratificeerd. Maar ook naast deze formele verplichting zijn er tal van activiteiten. Internationaal opereert "VHDL international" kortweg aangeduid met "VI". Via anonymous login op ftp adres "vhdl.org" en het www op adres "http://vhdl.org/" is een veelheid aan informatie verkrijgbaar over o.a. de actuele ontwikkelingen, zoals beta versies van in ontwikkeling zijn packages. Daarnaast is er een actieve newsgroup "comp.lang.vhdl" waar gebruikers terecht kunnen met vragen en opmerkingen.

Binnen de IEEE zijn een aantal werkgroepen, welke gecoördineerd worden door de DASC (Design Automations Standards Committee), actief met nauw verwante onderwerpen, o.a.:

PAR ¹	Group Title
1029.1	Waveform and Vector Exchange Specification Working Group
1076	VHDL Analysis and Standardization Working Group
*2 1076a	VHDL Shared Variable Working Group
1076.1	VHDL Analog Extensions Working Group
1076.2	VHDL Math Package Working Group
1076.3	VHDL Synthesis Package Working Group
1076.4	VHDL Timing Methodology Working Group
* 1076.5	VHDL Utility Library Working Group
* 1076.6	Standard for VHDL Register Transfer Level Synthesis
1165	VHDL/EDIF Interoperability Working Group

¹ PAR = Project Authorization Request

² De met een asterix aangegeven PAR's zijn nog geen IEEE standaard

De Std. 1029.1 wordt vaak aangeduid met WAVES en beschrijft op welke wijze in VHDL op een gestandaardiseerde wijze testvectoren beschreven kunnen worden. In de zomer van 1997 is de herziene versie ter beoordeling voorgelegd aan de stemgerechtigde leden. De leden hebben in ruime meerderheid ingestemd met de herziene standaard zodat verwacht mag worden dat aan het eind van 1997 of begin 1998 dit de nieuwe standaard zal worden.

De “shared variable working group” (PAR 1076a) moet met een oplossing komen voor het gebruik van de *shared variable* welke onderdeel is van VHDL’93. Over de problematiek is meer te vinden in dit boek.

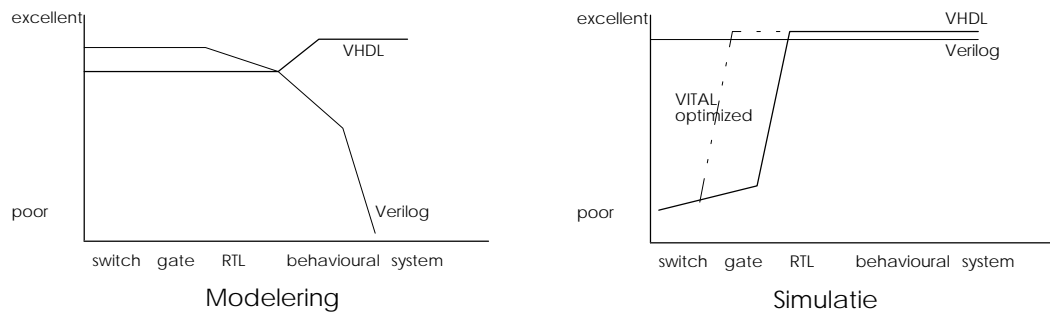
De “VHDL Analog Extensions Working Group” (PAR 1076.1) komt met een ‘superset’ van de VHDL standaard, met als doel dat ook analoge systemen, eventueel in combinatie met digitale systemen, beschreven en gesimuleerd kunnen worden. Hieraan wordt in dit boek geen aandacht besteed. De draft versie is in de zomer van 1997 ter beoordeling voorgelegd aan de balloting groep. Een meerderheid heeft inmiddels, met de nodige correcties, ingestemd met het concept. Verwacht wordt dat in het voorjaar van 1998 het een IEEE standaard zal zijn.

De “VHDL Math Package Working Group” (PAR 1076.2) is in 1996 geratificeerd (std. 1076.2-1996) en bevat een tweetal packages *math_real* en *math_complex*. Deze package biedt een uitkomst voor diegene die met reële of complexe getallen werkt.

De “VHDL Synthesis Package Working Group” (PAR 1076.3), geratificeerd in het voorjaar van 1997, maakt een einde aan de wildgroei op het gebied van synthese van VHDL. Elk synthese systeem gebruikt zijn eigen interpretatie en typering voor bit_vectoren en de operaties die hierop gebaseerd zijn, waardoor er nauwelijks sprake is van portabiliteit tussen de verschillende synthese systemen. In de afgelopen jaren werd wel steeds vaker de package *std_logic_arith* van Synopsys gebruikt als standaard. Een tweetal packages zijn het resultaat van de IEEE standaard, één is gebaseerd op het type bit (package *numeric_bit*) en de andere gaat uit van de IEEE package *std_logic_1164* (package *numeric_std*). Deze standaard legt dus niet vast welke subset van VHDL ondersteund moet worden door een synthese-tool. Het is frustrerend om tot de ontdekking te komen dat een beschrijving om niet essentiële zaken gewijzigd moet worden om door een ander synthese-tool geaccepteerd te worden. De bedoeling van de PAR 1076.6 is te komen tot afspraken over een subset (syntax en semantiek) van VHDL en Verilog voor synthese.

De “Verilog Analysis and Standardization Working Group” (PAR 1364) kan in zekere zin gezien worden als een concurrent van VHDL. De basis van Verilog is in de wintermaanden van 1983-1984 gelegd door Phil Moorby en op de EDA markt gebracht in 1985 door Cadence Design Systems. In 1990 zijn de rechten public domain geworden en sindsdien heeft de onafhankelijke *Open Verilog International* (OVI) groep er bekendheid aan gegeven. Nu Verilog ook een IEEE standaard is geworden (de standaard is in december 1995 geratificeerd) zullen beide talen naast elkaar gebruikt worden. Naast elkaar omdat de beide talen niet helemaal hetzelfde doel hebben. Verilog is bedoeld vanaf RTL (Register Transfer Level) *tot en met* de realisatie (denk bijvoorbeeld aan de beschrijving van een transistor) terwijl VHDL bedoeld is vanaf specificatie *tot* aan de realisatie. Veel CAE omgevingen ondersteunen al een co-simulatie van VHDL en Verilog beschrijvingen, immers het ligt voor de hand dat op een hoog niveau in VHDL wordt ontworpen terwijl de realisatie in Verilog is beschreven. Op dit moment gebruiken de meeste gebruikers

beide talen op register transfer niveau en welke taal is dan het beste? Geconstateerd moet worden dat in Amerika veel gebruik gemaakt wordt van Verilog en in Europa voornamelijk van VHDL. Vanaf 1997 zien de Verilog en VHDL gebruikers elkaar in de gezamenlijk georganiseerde IVC/VIUF (International Verilog Conference/VHDL International Users' Forum) conferentie elk voorjaar gehouden aan de oostkust van Amerika. Janick Bergeron heeft een vergelijking gemaakt tussen VHDL en Verilog met betrekking tot de simulatiesnelheid en het modeleren.



figuur 1.1 Vergelijking tussen Verilog en VHDL.³

In de VHDL standaard is niet aangegeven op welke wijze ASIC bibliotheken met al hun timingsaspecten beschreven moeten zijn. In mei 1992 werd tijdens de VHDL International User's Forum in Scottsdale besloten dit probleem op te pakken. Uitgangspunt hierbij was het gebruik maken van het SDF (standard delay format), welke ook bij Verilog wordt gebruikt, en het gebruik maken van de al ontwikkelde Std_timing package van de VHDL Technology Group en de technieken van "Ryan & Ryan". Verder moest voor een groot draagvlak worden gezorgd bij de industrie.

Dit initiatief kreeg dan ook de naam VITAL: VHDL Initiative Towards ASIC Libraries. In maart 1994 kwam de "VITAL 2.2b Model Development Specification" uit. Na dit prototype van de nieuwe standaard is gewerkt aan VITAL 3.0 welke is voorgedragen als IEEE standaard. De IEEE ratificeerde deze standaard in december 1995. Inmiddels leveren veel ASIC vendors ook de VITAL bibliotheken. In VITAL wordt onder andere een tweetal packages beschreven, *VITAL_Timing* en *VITAL_Primitives*, waarin respectievelijk een groot aantal timing procedures zijn beschreven en primitieve operaties op bit niveau zijn beschreven. Met daaraan voor de CAE tools de uitdaging de package body's zo te implementeren dat de simulaties snel zijn. Verder is er een document beschreven waaraan de gebruiker zich moet houden. Een consequentie van het gebruik van VITAL is ook aangegeven in figuur 1.1. Nu al wordt gewerkt aan een nieuwe VITAL standaard voor 1998 (IEEE Std. 1076.4-1998). Belangrijkste aanpassing in deze nog te ratificeren standaard is dat ook het modelering van geheugens is beschreven.

De "Standard VHDL Language Utility Library" (PAR 1076.5) wil een aantal packages definiëren om het ontwikkelen van modellen te vereenvoudigen en ook de portabiliteit te verhogen.

Een PAR hoeft niet altijd een IEEE standaard tot gevolg te hebben. Zo is er een PAR 1163 geweest met als titel "Interface for IEEE Std. 1076-1987 to Computer Aided

³ Janick Bergeron, "Verilog for experienced VHDL User's", Tutorial at the Spring VIUF'95 conference 1995, San Diego.

Design as Manufacturing (CAD/CAM) Tools”. Inmiddels is deze PAR ingetrokken door de IEEE.

1.1 Eigenschappen van VHDL.

VHDL heeft een groot aantal aantrekkelijke eigenschappen:

1. Hiërarchie wordt ondersteund.
2. De gebruiker is vrij om haar/zijn eigen ontwerpmethode te gebruiken: top-down, bottom-up, etc.
3. Technologie onafhankelijk. Dit betekent concreet dat een ontwerp van enkele jaren geleden zonder problemen hergebruikt kan worden in een andere technologie.
4. Ondersteuning van verschillende beschrijvingsstijlen, zoals een finite state machine, een algoritme, booleaanse vergelijkingen etc.
5. “Second-sourcing” eenvoudiger. Aangezien de taal CAD/CAE Tool onafhankelijk is kan zonder veel problemen worden omgeschakeld naar een andere omgeving. Belangrijk is het dat ook de testomgeving in VHDL is beschreven. Hiervoor kan bijvoorbeeld gebruik worden gemaakt van WAVES.
6. Verhoging van het abstractieniveau waardoor een betere beheersing van de complexiteit mogelijk is. Verder kunnen details aan de synthese-tools worden overgelaten.
7. Het vroegtijdig opsporen van fouten doordat de specificatie al te simuleren is en elke ontwerpstap tegen de simulatie is te testen.
8. Large Scale Integration (LSI) modelering is eenvoudig door het gebruik van ‘componenten’, ‘functies’, ‘procedures’ en ‘packages’.
9. VHDL legt geen beperking op aan de grootte van het ontwerp.
10. ‘Test benches’ kunnen in dezelfde taal worden beschreven en zijn dus overdraagbaar.
11. Dankzij de ‘generics’ en ‘attributes’ is backannotation eenvoudig. Hiervan wordt ook gebruik gemaakt in VITAL.
12. VHDL is een sterk getypeerde taal. Nieuwe datatypes zijn door de gebruiker zelf te definiëren

2. Inleiding in de beschrijvingstaal VHDL.

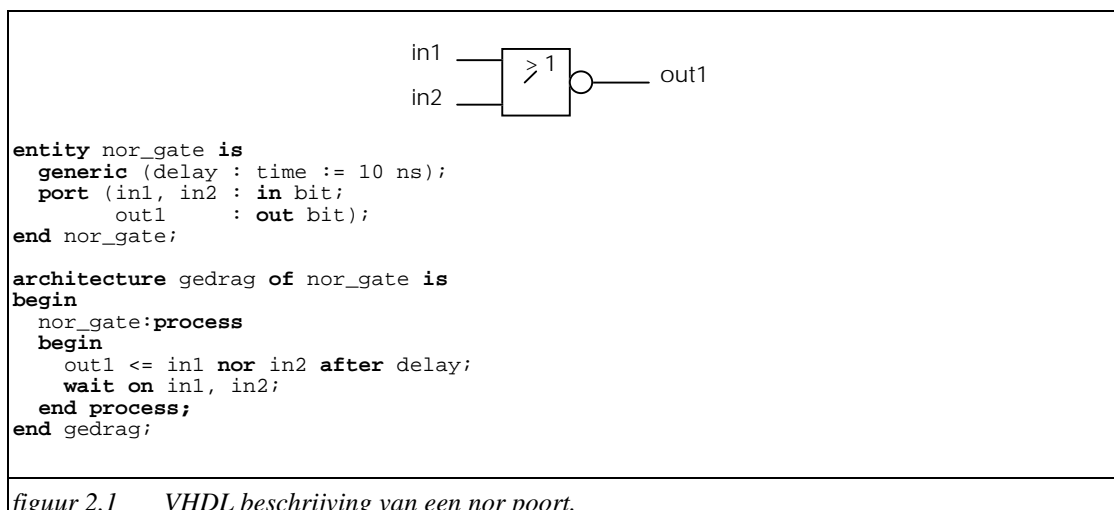
2.1 Introductie.

Dit hoofdstuk geeft een inleiding in de taal. Hoewel VHDL geen imperatieve taal is lijkt de syntax van de nog te behandelen processen veel op die van sterk getypeerde imperatieve talen, zoals MODULA-2. Verondersteld is dat de lezer dergelijke talen kan lezen, er wordt dan ook niet uitvoerig ingegaan op de syntax van de taal. De gegeven voorbeelden in combinatie met de syntax-beschrijving in appendix B moeten voldoende zijn om zelf beschrijvingen te lezen en vervolgens ook te schrijven. In de voorbeelden zijn de gereserveerde woorden van de taal dik gedrukt weergegeven en in de ASCII files worden hoofdletters gebruikt. VHDL maakt geen onderscheid tussen kleine en hoofdletters. Verder is in dit boek niet gestreefd naar het vertalen van veel gebruikte Engelse benamingen.

Als eerste zal de kern van de taal worden besproken: communicerende processen. Hierbij wordt ook aangegeven hoe deze processen gesimuleerd worden. Vervolgens wordt een inleiding gegeven in de taalconstructies.

2.2 Communicerende processen.

Hardware kan worden beschouwd als een verzameling van parallelle processen die met elkaar communiceren. Elk proces berekent continu op basis van zijn ingangssignalen de daarbij behorende uitgangssignalen. In een *nor* poort vindt als het ware continu een berekening plaats, ook al zijn de ingangssignalen stabiel. Voor het simuleren van een *nor* poort is het echter voldoende om alleen de veranderingen door te rekenen (figuur 2.2). Dus wanneer een ingangssignaal verandert moet een nieuw uitgangssignaal worden berekend. Dit simulatiemodel, dat ook in VHDL wordt gebruikt, noemt men *event-driven simulation*.



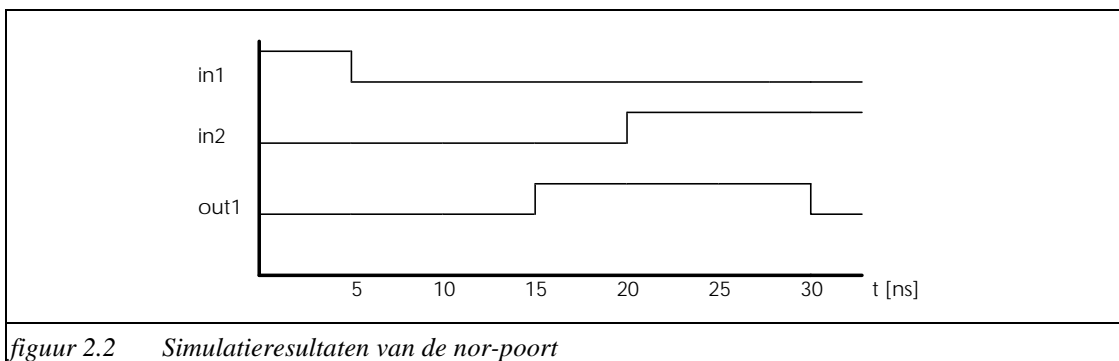
figuur 2.1 VHDL beschrijving van een nor poort.

Figuur 2.1 geeft een VHDL beschrijving van een nor poort. Deze beschrijving bestaat uit twee delen: de ENTITY en de ARCHITECTURE.

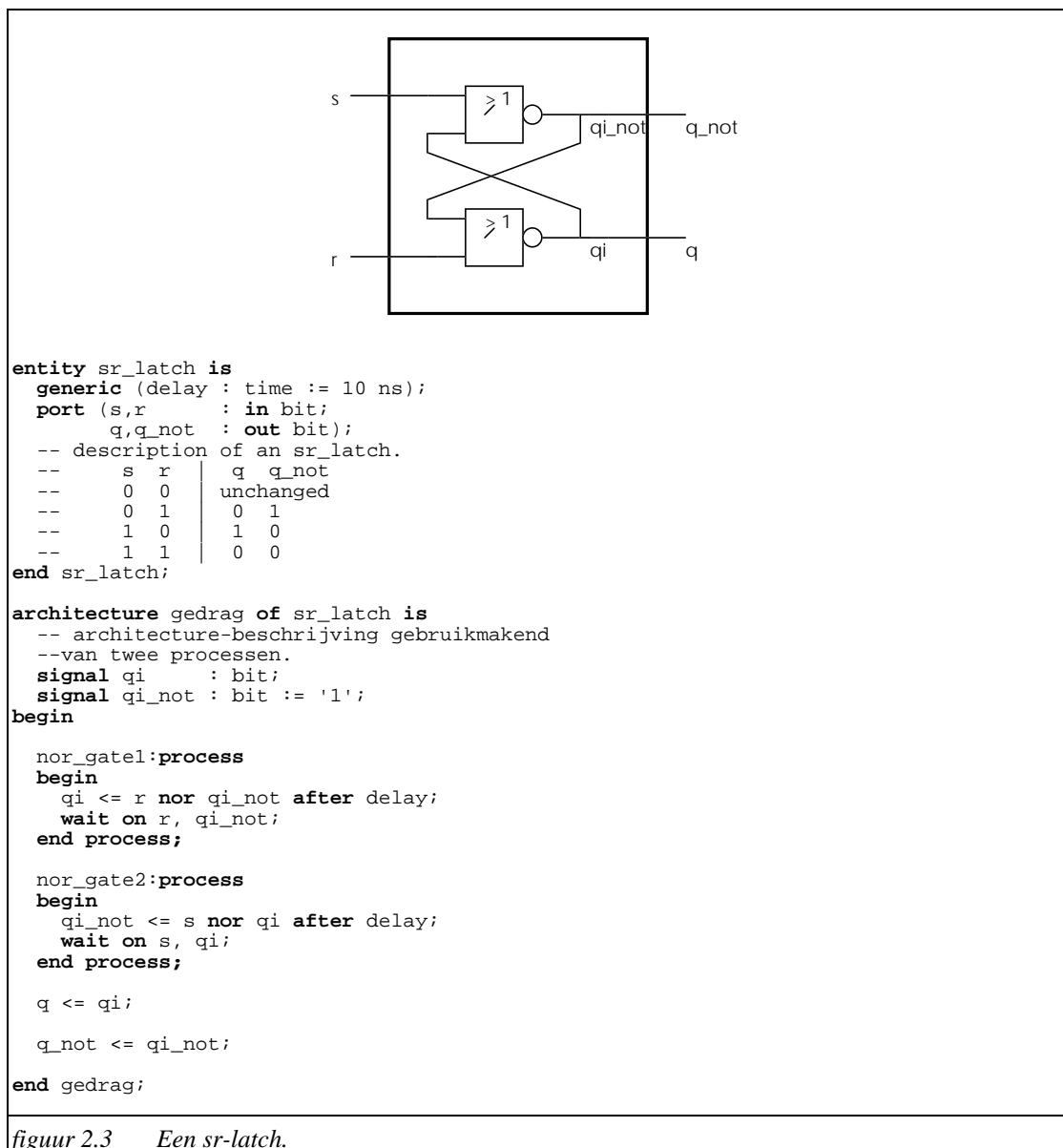
De entity geeft de naam van het component en wat de in- en uitgangssignalen zijn. Bij deze signalen moet ook het type worden opgegeven. In dit voorbeeld zijn beide ingangssignalen en het uitgangssignaal van het type bit. Dit type is een standaard type van VHDL en is gedefinieerd als een enumeratie: *type bit is ('0','1');*. In de entity-beschrijving kan ook nog statische informatie worden opgenomen middels een *generic*

statement, in dit voorbeeld geeft het aan dat de poort default een vertraging heeft van 10 ns. Beschouw de generic voorlopig als een constante. De generic en de port statements mogen achterwege blijven.

De architecture bepaalt het gedrag tussen de in- en uitgangssignalen in dit voorbeeld door middel van een procesbeschrijving. Een procesbeschrijving is een sequentiële beschrijving te vergelijken met die van de imperatieve talen. Voor de *nor* poort kan het gedrag door middel van een eenvoudige procesbeschrijving worden bepaald. Deze geeft aan dat het uitgangssignaal OUT1 de logische nor is van IN1 en IN2 en vervolgens wordt er gewacht op een verandering van IN1 en/of IN2. Na een verandering van een ingangssignaal vervolgt de executie van het proces zich weer na de wait statement en aan het einde gekomen van de procesbeschrijving vervolgt de executie zich aan het begin.



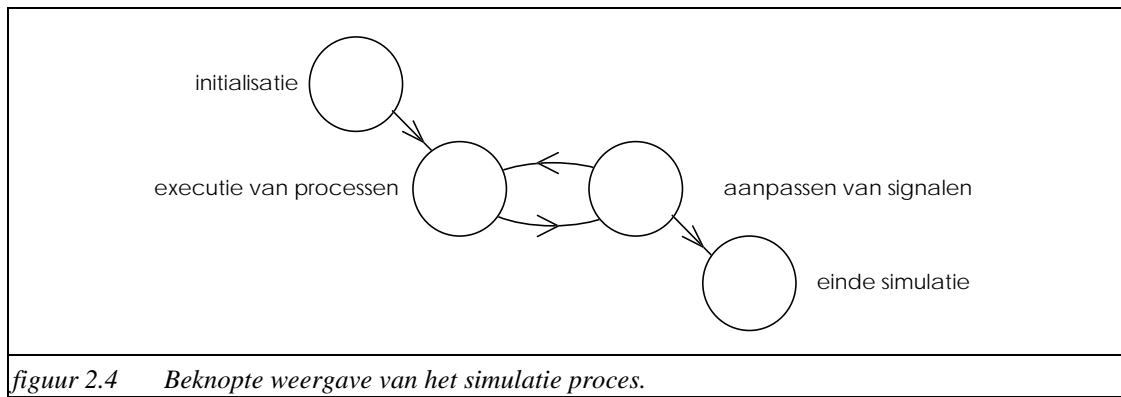
figuur 2.2 Simulatieresultaten van de nor-poort



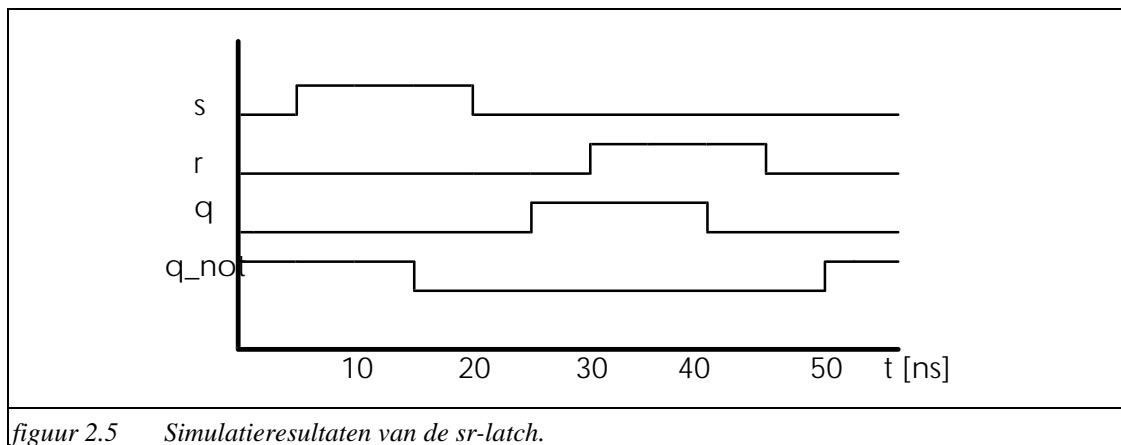
figuur 2.3 Een sr-latch.

Figuur 2.3 geeft een VHDL beschrijving van een sr-latch. Tevens is in de entity-beschrijving commentaar opgenomen, door gebruik te maken van de '--'. Documenteren is een belangrijk onderdeel van het ontwerpproces. *Wat een entity behoort te doen staat altijd als commentaar in de entity en hoe dat wordt verkregen als commentaar in de architecture.*

Elke nor poort is beschreven met een procesbeschrijving. De volgorde waarin de beide processen zijn beschreven is niet belangrijk. Communicatie tussen beide processen kan alleen plaats vinden door middel van signalen. Hoe lokale signalen worden gedeclareerd is ook in het voorbeeld aangegeven, deze declaraties bevinden zich tussen de gereserveerde woorden ARCHITECTURE en BEGIN. De signalen QI en QI_NOT zijn in deze beschrijving lokaal. Verder is aangegeven dat het signaal QI_NOT tijdens het initialiseren de waarde '1' krijgt. *Indien een signaal niet expliciet geïnitieerd wordt krijgt het de meest linkse waarde van het bereik van het signaal.* Signaal QI wordt niet expliciet geïnitieerd, daarom krijgt deze tijdens het initialiseren de waarde '0' (type bit is ('0','1');). Ook de ingangssignalen hebben een initiële waarde. De ingangen S en R zijn beide '0' aangezien ze van het type bit zijn.



Na het starten van de simulatie vindt het simuleren in twee fasen plaats (figuur 2.4). Op één simulatie tijdstip (in het vervolg afgekort tot tijdstip) worden alle processen geëxecuteerd waarvan aan de wait conditie is voldaan, nadat alle processen zijn geëxecuteerd op dit tijdstip worden de signalen vervangen door de nieuw berekende waarden. De combinatie van executie en update fase wordt een simulatie cyclus genoemd. Dit betekent dus dat aan een signaal nooit direct een waarde wordt toegekend. In de volgende paragraaf wordt het simulatiemodel verder uitgewerkt. In figuur 2.5 zijn de resultaten van een simulatie gegeven.



In VHDL zijn taalconstructies aanwezig waarmee op eenvoudige wijze complexe systemen beschreven kunnen worden. De VHDL compiler vertaalt al deze beschrijvingen tot communicerende processen. Eerst zal uitvoeriger ingegaan worden op het simuleren.

Uit de beschrijving van de sr-latch volgt dat een signaal niet alleen een actuele waarde heeft, maar ook toekomstige waarden met de daarbij behorende tijdstippen⁴. De signal assignment statement $qi \leq r \text{ nor } qi_not \text{ after delay};$ geeft aan dat de waarde van signaal qi pas na een tijdsduur $delay$ verandert. Figuur 2.6 geeft een voorbeeld van een alle geplande toekenningen aan signaal DEMO, dit wordt een *waveform* genoemd voor signaal DEMO, en elk paar van waarde en tijdstip wordt een transaction genoemd.

⁴ Niet alle toekomstige waarden worden ook werkelijk aan het signaal toegekend (zie hoofdstuk 3, "het modelleren van vertraging")

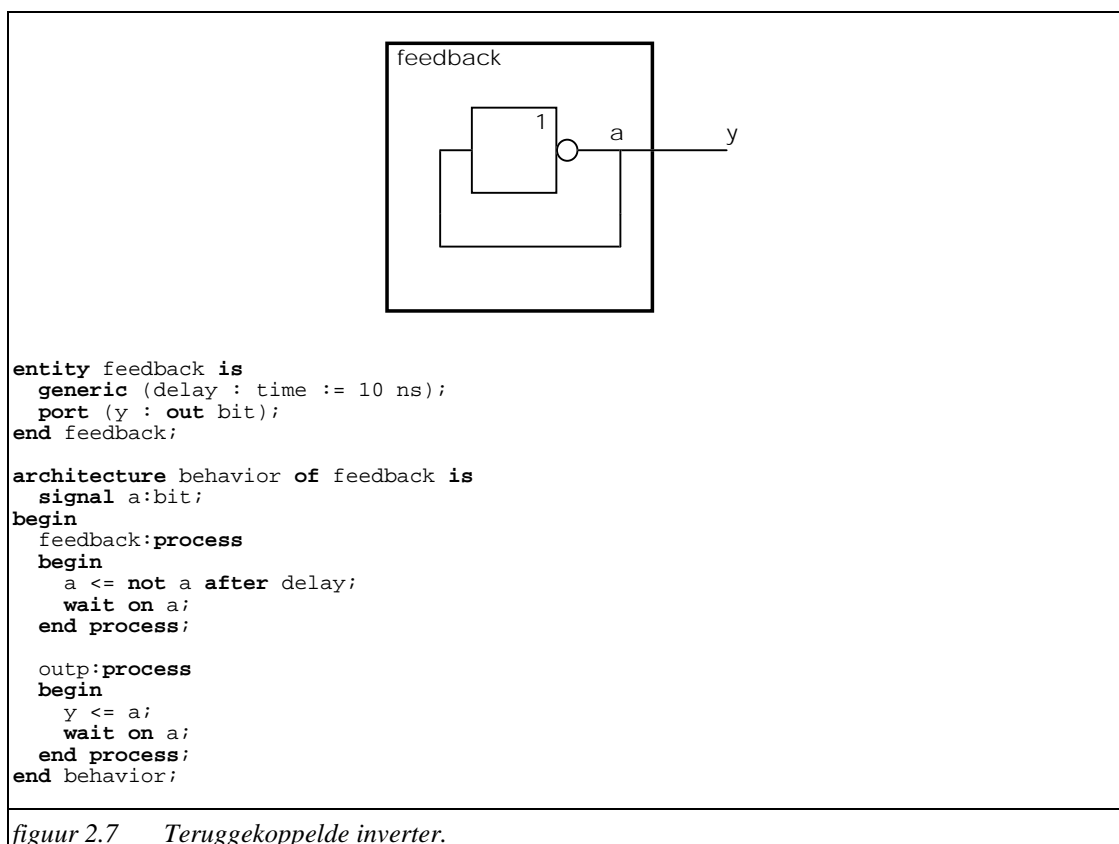
'0'	'1'	'0'		'0'	'0'
	110 ns	130 ns		300 ns	500 ns

signaal demo '0', '1' @ 110 ns, '0' @ 130 ns,..., '0' @ 300 ns, '0' @ 500 ns

figuur 2.6 Voorbeeld van een signaal met zijn toekomstige waarden en tijdstippen.

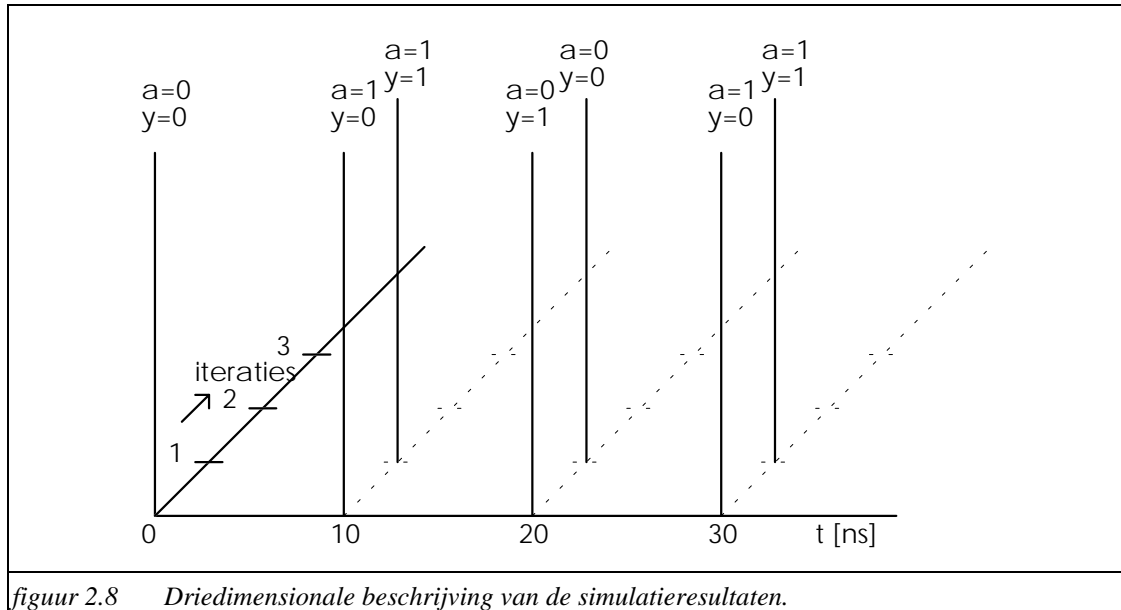
2.3 Simulatiemodel.

In deze paragraaf zal enige aandacht aan het simulatie proces van VHDL worden besteed omdat het soms voor het begrijpen van taalconstructies handig is. Ook wordt dan duidelijk waarom de simulator oneindig lang op hetzelfde tijdstip bezig kan zijn, waardoor de gebruiker de indruk kan hebben dat zijn computer 'hangt'.



In figuur 2.7 is een beschrijving gegeven van een teruggekoppelde inverter. De terugkoppeling is in het proces feedback beschreven en proces OUTP beschrijft het volgen van het lokale signaal A door het uitgangssignaal Y.

De vertragingstijd van de inverter is 10 ns waardoor het lokale signaal iedere 10 ns van waarde verwisselt en hiermee dus ook het uitgangssignaal.



figuur 2.8 Driedimensionale beschrijving van de simulatieresultaten.

Figuur 2.4 geeft globaal het simulatie proces weer. Wat in dit vereenvoudigd model nog niet duidelijk tot uiting komt is dat er verschil gemaakt moet worden tussen simulatie berekeningen op hetzelfde tijdstip, welke iteraties of delta-delays worden genoemd, en simulatie tussen de verschillende simulatie tijdstippen. Aan de hand van het driedimensionale vlak in figuur 2.8 wordt dit toegelicht.

De waarde van een signaal X hangt af van het tijdstip en de iteratie (ook wel 'delta' genoemd) binnen een tijdstip, de waarde van X wordt dus bepaald door $X(\text{TIJDS TIP}, \text{ITERATIE})$.

Uitgaande van het voorbeeld beschreven in figuur 2.7 geldt:

1: Initialisatie.

Ten gevolge van de initialisatie geldt voor de signalen A en Y resp. $A(0 \text{ ns}, 0) = '0'$ en $Y(0 \text{ ns}, 0) = '0'$.

2: Initialisatie van alle processen

Beide processen worden achtereenvolgens geëxecuteerd waarbij de volgorde niet van belang is. In proces OUTP is beschreven dat het signaal Y de waarde van het signaal A volgt, de toekenning vindt niet direct plaats maar pas tijdens de volgende iteratie op hetzelfde tijdstip. Dus $Y(0 \text{ ns}, 1) = '0'$. Proces FEEDBACK geeft aan dat het signaal A pas na 10 ns de nieuwe waarde krijgt. De waarde van het signaal A blijft onveranderd gedurende dit tijdstip. Dit betekent een stabiele situatie op dit tijdstip. Nu wordt de simulatie vervolgd op het tijdstip van de eerstvolgende verandering.

Voor de waveforms geldt:

signaal A : '0', '1' @ 10 ns

signaal Y : '0'

3: Iteratie(s) op tijdstip 10 ns.

Na 10 ns verandert het signaal A van '0' naar '1'. Er geldt dan: $A(10 \text{ ns}, 0) = '1'$ en $Y(10 \text{ ns}, 0) = '0'$; Ook nu weer verandert het signaal A niet gedurende dit tijdstip maar wordt voor het signaal A op tijdstip 20 ns de waarde '0' gepland. Proces OUTP zal er voor zorgen dat het signaal Y een delta-delay later, $Y(10 \text{ ns}, 1)$, de waarde '1' krijgt. Nu vindt

na iteratie 1 geen verandering meer plaats, zodat de simulatie wordt vervolgd op het tijdstip van de eerstvolgende verandering.

Voor de waveforms geldt:

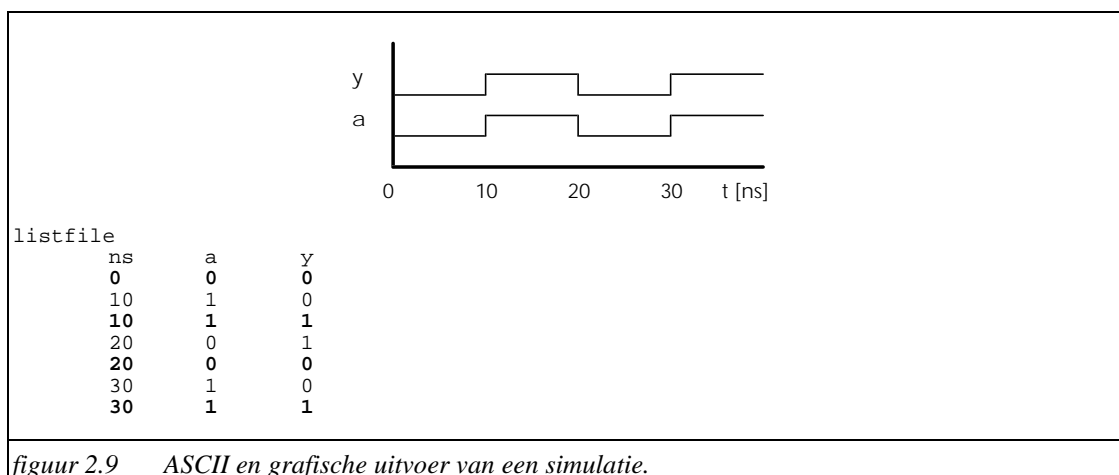
signaal A: '1', '0' @ 20 ns

signaal Y: '1'

4: De daarop volgende stappen.

Ga naar het eerstvolgende moment (dat kan zijn een iteratie, of indien er geen iteraties meer zijn een ander tijdstip) waarop een signaal veranderd⁵. Executeer alleen die processen die 'gevoelig' zijn voor dit signaal. Na executie van deze processen vindt wederom een update plaats van signalen.

Simulatoren geven de simulatie resultaten meestal grafisch weer, alleen de waarde van de signalen van de laatste iteratie wordt dan weergegeven. Soms kan ook voor een niet grafische representatie worden gekozen waarbij dan ook de signaalwaarden van de verschillende iteraties zichtbaar worden (figuur 2.9).



figuur 2.9 ASCII en grafische uitvoer van een simulatie.

De simulatie omgeving bepaalt tot welk tijdstip wordt gesimuleerd. Er kunnen zich echter situaties voordoen waardoor dit tijdstip nooit wordt bereikt. Als er tijdens de simulatie een tijdstip is waar de iteraties zich niet stabiliseren ontstaat er een oneindige herhaling van iteraties. Door de vertragingstijd van de inverter terug te brengen tot de niet realistische waarde van 0 ns ontstaat een dergelijke situatie. Denkt u trouwens dat de simulatieresultaten getoond in figuur 2.9 wel realistisch zijn voor de teruggekoppelde inverter?

Het verloop van de simulatie is dan als volgt:

1: Initialisatie

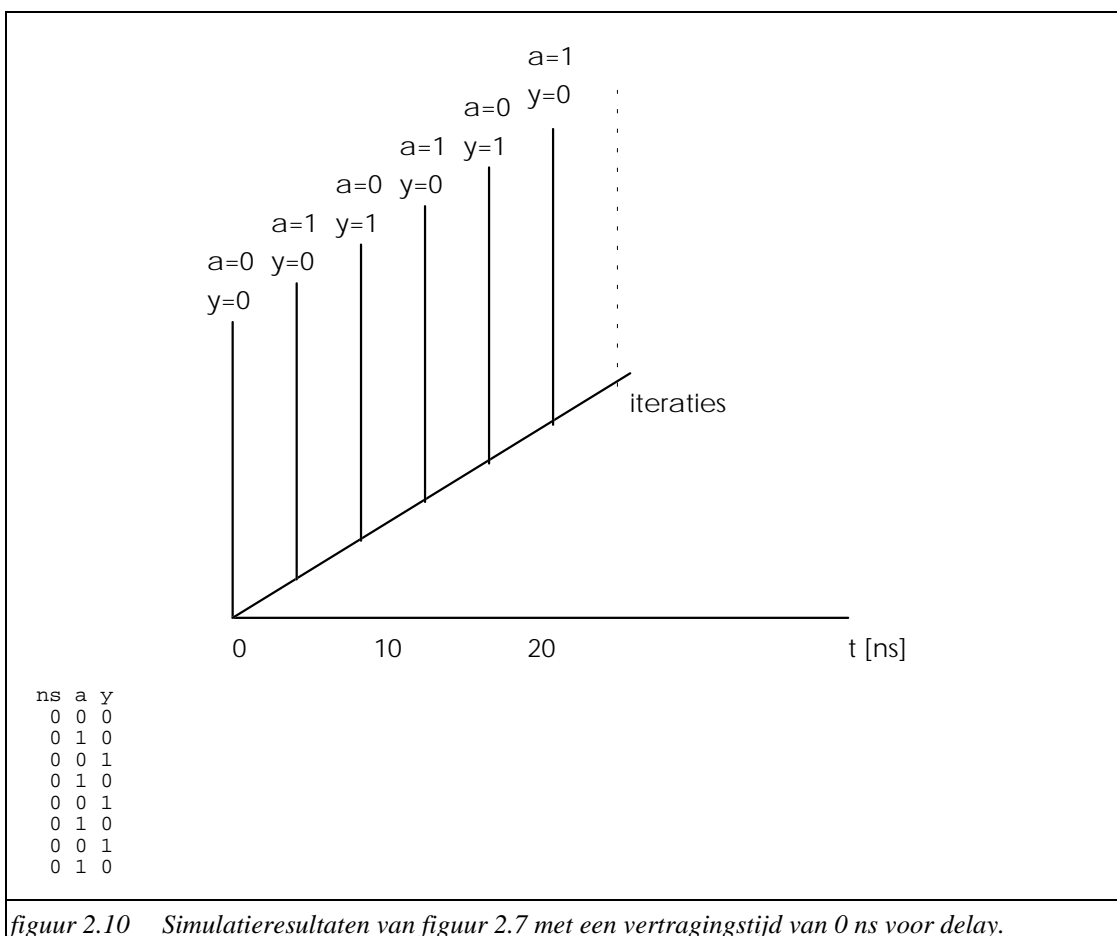
Ten gevolge van de initialisatie geldt voor de signalen A en Y resp. $A(0 \text{ ns}, 0) = '0'$ en $Y(0 \text{ ns}, 0) = '0'$.

2: Executie van alle processen.

⁵ Later zal blijken dat de executie van een proces ook voor een opgegeven tijdsduur gestopt kan worden. Na het verstrijken van deze tijdsduur moet uiteraard ook dit proces weer worden geëxecuteerd.

Beide processen worden (achtereenvolgens) geëxecuteerd waarbij de volgorde niet van belang is. In proces OUTP is beschreven dat het signaal Y de waarde van het signaal A volgt, echter deze toekenning vindt niet direct plaats maar pas tijdens de volgende iteratie op hetzelfde tijdstip. Dus $Y(0\text{ ns},1)='0'$. Proces FEEDBACK geeft aan dat het signaal A na 0 ns verandert, dus op hetzelfde tijdstip! Ook nu vindt deze toekenning niet direct plaats maar tijdens de volgende iteratie, m.a.w. $A(0\text{ ns},1)='1'$. Er is nu nog geen stabiele situatie verkregen, dus wordt nog een iteratie op hetzelfde tijdstip uitgevoerd waarbij als uitgangspunt $A(0\text{ ns},1)='1'$ en $Y(0\text{ ns},1)='0'$ wordt gebruikt. etc.

Figuur 2.10 geeft een aantal iteratie stappen weer. Wanneer beëindigt een VHDL simulator in een dergelijke situatie het simulatie proces? In principe nooit, echter bij de meeste simulatoren kan een bovengrens aan het aantal iteraties worden opgegeven. Het aantal iteraties hangt sterk af van de VHDL beschrijving.



figuur 2.10 Simulatieresultaten van figuur 2.7 met een vertragingstijd van 0 ns voor delay.

Realiseert u zich dat er tijdens het simuleren geen verschil gemaakt wordt tussen: $y \leq x \text{ after } 0\text{ ns}$; en $y \leq x$; . In beide gevallen geldt dat y niet direct een nieuwe waarde krijgt maar pas een iteratie later. Dit wordt ook wel *delta-delay* genoemd.

2.4 Procesbeschrijving.

Elke VHDL beschrijving wordt intern vertaald naar communicerende processen. In figuur 2.3 zijn de componenten NOR_GATE1 en NOR_GATE2 expliciet beschreven. Deze twee processen worden parallel uitgevoerd. Expliciete procesbeschrijvingen zijn gekoppeld aan de gereserveerde woorden PROCESS en END PROCESS. De *concurrent*

signal assignment statements $q \leq q_i$ en $q_{\text{not}} \leq q_{i_{\text{not}}}$ uit figuur 2.11 zijn ook processen maar deze zijn impliciet beschreven. Om aan te geven dat de beide concurrent signal assignment statements in feite processen zijn die impliciet beschreven zijn wordt de term *concurrent* gebruikt. Het concurrent signal assignment statement $q \leq q_i$ is equivalent met de expliciete procesbeschrijving:

```
process
begin
   $q \leq q_i$ ;
  wait on  $q_i$ ;
end process;
```

Het statement $q \leq q_i$ in de voorgaande procesbeschrijving is geen *concurrent signal assignment statement*, maar een *signal assignment statement*. Processen kunnen dus niet genest voorkomen, er is slechts één niveau van processen.

In het algemeen geldt dat een concurrent signal assignment statement omgezet kan worden naar een expliciete procesbeschrijving waarbij alle ingangssignalen als lijst aan de wait on statement worden toegevoegd, dit statement bevindt zich aan het eind van de procesbeschrijving.

Nogmaals kijkend naar figuur 2.3 kan het proces NOR_GATE1 vervangen worden door de concurrent signal assignment statement $q_i \leq r \text{ nor } q_{i_{\text{not}}} \text{ after delay}$;, evenzo is dit mogelijk voor het proces NOR_GATE2. De leesbaarheid en compactheid van de architecture-beschrijving wordt hierdoor aanzienlijk verbeterd (figuur 2.11). Het gebruik van een label voor een concurrent signal assignment statement is toegestaan.

```
architecture gedrag1 of sr_latch is
  signal  $q_i$  : bit;
  signal  $q_{i_{\text{not}}}$  : bit := '1';
begin
  nor_gate1:  $q_i \leq r \text{ nor } q_{i_{\text{not}}} \text{ after delay}$ ;
  nor_gate2:  $q_{i_{\text{not}}} \leq s \text{ nor } q_i \text{ after delay}$ ;
   $q \leq q_i$ ;
   $q_{\text{not}} \leq q_{i_{\text{not}}}$ ;
end gedrag1;
```

figuur 2.11 Impliciete procesbeschrijvingen, equivalent met figuur 2.3.

2.4.1 Expliciete processen.

Het gedrag dat gemodelleerd moet worden is meestal aanzienlijk gecompliceerder dan dat van een nor poort. Vaak kan een expliciete procesbeschrijving dan duidelijker zijn dan vele afzonderlijke processen. De structuur van een expliciete procesbeschrijving is (niet volledig):

```
process
  [ declaraties ]
begin
  [ sequentiële statements ]
end process;
```

De declaraties komen globaal overeen met de mogelijkheden uit de imperatieve talen (types, constanten, variabelen, functies, procedures, etc.). De statements worden in volgorde geëxecuteerd, aan het eind gekomen vervolgt de executie zich aan het begin van de sequentiële statements.


```

process
  variable i:integer:=0;
begin
  i:=i+1;
  y<=i;
end process

```

figuur 2.12 Proces met een oneindige executie.

Figuur 2.12 geeft een voorbeeld van een procesbeschrijving. Lokaal is er een variabele *I* gedeclareerd van het type *integer* met initiële waarde 0. Vervolgens wordt in de procesbeschrijving de variabele *I* opgehoogd en krijgt het signaal *Y* delta-delay later de waarde van de variabele *I*. De verhoging van de variabele *I* gebeurt direct. Merk het verschil in de assignment operatoren op. De *:=* wordt gebruikt voor toekenning aan constanten, variabelen en het aangeven van initiële waarden, terwijl de *<=* wordt gebruikt voor toekenningen aan signalen.

Zoals al is aangegeven worden alle statements in een proces in de gegeven volgorde uitgevoerd. Bij het bespreken van het simulatiemodel is al aangegeven dat alleen een wait statement de executie van een proces kan onderbreken. Als een wait statement ontbreekt, zoals in figuur 2.12, ontstaat er een oneindige lus tijdens het simuleren. De VHDL compilers geven meestal een waarschuwing indien in een proces geen wait statement voorkomt.

2.4.1.1 Wait statements.

Om de executie van een proces te onderbreken is er een wait statement. De algemene vorm van dit statement is:

```
wait [on <sensitivity list>][until <boolean expression>][for <time expression>]
```

In één procesbeschrijving mogen meerdere wait statements voorkomen.

2.4.1.1.1 Wait on <sensitivity list>.

Bij het *wait on* statement wordt de executie vervolgd indien een of meer signalen in de gegeven lijst met *signalen* van waarde verandert. Deze lijst wordt de *sensitivity list* genoemd. Voorbeeld: *wait on a,b;*

2.4.1.1.2 Wait until <boolean expression>.

Het *wait until* statement geeft vaak aanleiding tot misverstanden. Ten onrechte wordt vaak verondersteld dat de executie zich vervolgt zolang de boolean expressie waar is, terwijl de conditie juist waar moet worden! Het wait until statement kan ook eenvoudig vertaald worden in een loop constructie.

```
wait until clk='1';
```

is equivalent met:

```
wait_until_equivalent : loop
  wait on clk;
  exit when clk='1';
end loop wait_until_equivalent;
```

Er van uitgaande dat een latch is gedefinieerd als zijnde een niveau gevoelig geheugen en een flipflop een flankgevoelig geheugen geeft figuur 2.13 een beschrijving van een flipflop omdat de wait until conditie overeenkomt met het flank gevoelig zijn van het signaal op de klokingang.

```

entity flipflop is
  port (d,clk : in bit;
        q      : out bit);
end flipflop;

architecture gedrag of flipflop is
begin
  label:process -- label is optioneel
  begin
    wait until clk='1';
    q <= d;
  end process;
end gedrag;

ns d clk q
0 0 0 0
0 1 0 0
100 1 1 0
100 1 1 1
200 0 1 1
300 0 0 1
400 0 1 1
400 0 1 0

```

figuur 2.13 Beschrijving van een flipflop.

2.4.1.1.3 Wait on <sensitivity list> until <boolean expression>.

Een proces met een combinatie van de vorige twee wait statements vervolgt pas zijn executie indien er een event optreedt in een van de signalen van de sensitivity list, tevens moet de boolean expressie waar zijn.

wait on a until b='1';

is equivalent met:

```

wait_until_equivalent : loop
  wait on a;
  exit when b='1';
end loop wait_until_equivalent;

```

Merk op dat het statement

wait until b='1';

equivalent is met:

wait on b until b='1';

2.4.1.1.4 Wait for <time expression>.

Het *wait for* statement onderbreekt de executie gedurende de aangegeven tijdsperiode van de expressie. Voorbeeld: *wait for 10 ns;*

2.4.1.1.5 Wait on <sensitivity list> until <boolean expression> for <time expression>.

Een procesbeschrijving met een dergelijk wait statement vervolgt haar executie pas weer nadat:

- er een signaal in de sensitivity list van waarde verandert en de boolean expressie waar is, of
- de opgegeven tijdsduur is verstreken.

2.4.1.1.6 Wait.

De executie van een proces wordt nooit meer vervolgd. Dit lijkt misschien niet zinvol, toch kan dit goed worden gebruikt om bijvoorbeeld tijdens de simulatie eenmalig iets

uit te voeren zoals het initialiseren van een geheugen waarbij de data in een file is opgeslagen.

2.4.1.1.7 Impliciete wait.

Achter het gereserveerde woord PROCESS kan een lijst van signalen worden opgegeven, de sensitivity list. Een dergelijke beschrijving komt overeen met een procesbeschrijving met als laatste statement een wait on statement met daarin de sensitivity list.

```
process (a,b)
begin
  y <= a and b;
end process;
```

Verwar process (a,b) niet met een functieaanroep met de parameters a en b. Bovenstaande beschrijving is gelijk aan:

```
process
begin
  y <= a and b;
  wait on a,b;
end process;
```

Als een dergelijk impliciet wait statement is gebruikt mogen er geen wait statements voorkomen in de procesbeschrijving.

2.4.1.2 Besturingsstructuren in een proces.

Evenals bij imperatieve talen kan ook de volgorde van executie van de statements in een proces worden beïnvloed. In deze paragraaf zullen de verschillende besturingsstructuren kort worden toegelicht, te weten de conditionele en iteratieve structuren.

2.4.1.2.1 Conditionele besturingsstructuren.

Syntax beschrijvingen behorend bij de conditionele besturingsstructuren zijn, o.a. de bekende 'if then' structuren:

- **if** conditie **then** statement **end if**;
- **if** conditie **then** statement **else** statement **end if**;
- **if** conditie **then** statement **elsif** conditie **then** statement **end if**;

Figuur 2.14 geeft bijvoorbeeld een beschrijving van een latch gebruik makend van deze vorm.

```
process
begin
  if clk='1' then q<=d; end if;
  wait on clk,d;
end process;
```

figuur 2.14 Beschrijving van een latch.

```

entity multiplexer is
  port (sel1,sel0      : in bit;
        in3,in2,in1,in0 : in bit;
        outp          : out bit);
  -- multiplexer met selectie ingangen sel1,sel0
  -- en ingangslijnen in3 t/m in0
end multiplexer;

architecture gedrag1 of multiplexer is
begin
  process(sel1,sel0,in3,in2,in1,in0)
  begin
    if sel1='0' and sel0='0' then outp <= in0;
    elsif sel1='0' and sel0='1' then outp <= in1;
    elsif sel1='1' and sel0='0' then outp <= in2;
    else outp <= in3;
    end if;
  end process;
end gedrag1;

architecture gedrag2 of multiplexer is
begin
  process(sel1,sel0,in3,in2,in1,in0)
  subtype bv2 is bit_vector(0 to 1);
  begin
    case bv2'(sel1&sel0) is -- 6
      when "00" => outp <= in0;
      when "01" => outp <= in1;
      when "10" => outp <= in2;
      when others => outp <= in3;
    end case;
  end process;
end gedrag2;

```

figuur 2.15 Gedragsbeschrijving van een multiplexer.

In figuur 2.15 is een architecture-beschrijving gegeven van een multiplexer waarbij gebruik gemaakt is van de 'if then' structuur. Tevens is ook een alternatieve beschrijving gegeven m.b.v. een *case statement*. In deze beschrijving worden door middel van SEL1&SEL0 de twee ingangsbits geconcateneerd tot een vector, indien deze vector gelijk is aan "00" wordt OUTP gelijk aan IN0 etc.. Tevens is in de beschrijving ook het statement WHEN OTHERS opgenomen hiermee wordt bereikt dat niet alle mogelijkheden afzonderlijk beschreven hoeven te worden.

Voor het case statement geldt:

1. Alle keuzemogelijkheden moeten voorkomen.
2. Keuzemogelijkheden mogen elkaar niet overlappen.

⁶ De lezer is mogelijk verrast door de constructie " bv2'(sel1&sel0) " Dit is een qualified expression welke nodig is volgens de standaard. Het volgende is dus NIET toegestaan:

case sel1&sel0 is ...

Hetzelfde geldt ook voor het concurrent selected signal assignment statement. Ook hier mag de expressie waarop wordt geselecteerd geen concatenatie van signalen zijn.

qualified expression.

Er zijn situaties denkbaar waarin een type niet eenduidig gedefinieerd is. Bij de concatenatie van twee bits ontstaat een bit_vector. Echter de range van deze bit_vector is hiermee niet bepaald. Door een qualified expression te gebruiken wordt ook dit vastgelegd.

Stel twee procedures:

```
procedure q(inp:character);
```

```
procedure q(inp:bit);
```

Als deze procedure wordt aangeroepen met q('1') dan is het niet duidelijk of er sprake is van het bit '1' of het character '1'! M.a.w. welke procedure q moet worden gekozen? Dit probleem kan ook door een qualified expression worden opgelost:

```
q(character('1'));
```

3. Verder moet het selectie criterium *lokaal statisch* zijn, d.w.z. dat tijdens het analyseren van de design unit waarin het case statement wordt gebruikt het bereik al vast moet liggen. Het mag dus bijvoorbeeld niet zijn afgeleid van een ‘generic’! Om het eerste te bereiken wordt vaak **when others** gebruikt. Bij het case statement moeten de keuzes statisch bepaald zijn, het volgende is toegestaan:

```

case int is
    when 1 | 3 | 5          => .... -- 1 of 3 of 5
    when 10 to 15          => .... -- 10 t/m 15
    when 2**4 to 2**5      => .... -- 16 t/m 32
    when others            => null;-- overige waarden
end case;

```

Het NULL-statement geeft aan dat er geen actie uitgevoerd hoeft te worden.

2.4.1.2.2 Iteratieve besturingsstructuren.

Er zijn verschillende vormen aanwezig om iteratie te bereiken o.a.: de *for*, de *while* en de *loop* met een *exit* conditie of een *next* conditie. In figuur 2.16 zijn de drie vormen gebruikt om de faculteit te berekenen van een getal. De loop variabele *i* in architecture *FOR_LUS* wordt niet gedeclareerd.

```

entity faculteit is
    port ( n      : in integer;
           fac    : out integer);
end faculteit;

architecture for_lus of faculteit is
begin
    process(n)
        variable prd : integer;
    begin
        prd := 1;
        for i in 1 to n loop
            prd := prd * i;
        end loop;
        fac <= prd;
    end process;
end for_lus;

architecture while_lus of faculteit is
begin
    process(n)
        variable prd,f : integer;
    begin
        prd:=n; f:=1;
        while prd>0 loop
            f:=f*prd;
            prd:=prd-1;
        end loop;
        fac <= f;
    end process;
end while_lus;

architecture exit_lus of faculteit is
begin
    process(n)
        variable prd,f : integer;
    begin
        prd:=n; f:=1;
        loop
            exit when prd<=0;
            f:=f*prd;
            prd:=prd-1;
        end loop;
        fac <= f;
    end process;
end exit_lus;

```

figuur 2.16 Drie architectures voor faculteit.

Een loop mag ook voorzien worden van een *looplabel* verder mogen loops ook genest voorkomen:

```
buiten:loop
..
binnen:loop
..
exit buiten when conditie;
..
end loop binnen;
end loop buiten;
verder: ..
```

Enkele opmerkingen:

- *Exit* statement
exit <looplabel> <when conditie>
 In het exit statement *mag* ook een looplabel worden opgenomen. In dat geval wordt, wanneer de conditie waar is, de executie vervolgd na *end loop* behorende bij deze looplabel. In het voorbeeld hierboven wordt de executie dan vervolgd vanaf label *VERDER*. Indien geen gebruik wordt gemaakt van het looplabel vervolgt de executie zich na de end loop behorende bij het huidige loop statement.
 De when conditie is optioneel. Het ontbreken van een conditie houdt in dat executie altijd verder gaat na de end loop. Dus equivalent met *exit <looplabel> when true*.
- *Next* statement
 In plaats van het exit statement kan ook een next statement worden opgenomen.
next <looplabel> <when conditie>
 Indien de conditie waar is wordt de executie vervolgd bij het begin van de loop met het aangegeven looplabel. M.a.w. de sequentiële statements na de next statements worden niet uitgevoerd. Ook hier is het looplabel optioneel. Indien deze niet voorkomt wordt de executie vervolgd aan het begin van het huidige loop statement.
- De *for* en de *while* loop mogen ook voorzien worden van een label. Ook deze labels kunnen gebruikt worden bij het exit statement en het next statement.
 In een for en een while loop kan ook een exit statement en een next statement worden opgenomen.
 In de hieronder gegeven beschrijving wordt van een input vector (INP is een bit_vector) bepaald of deze alleen uit enen bestaat.

```
for i in inp'range loop
  all_one<=false
  exit when inp(i)/='1';
  all_one<=true
end loop;
```

2.4.1.2.3 Het assert statement.

In een procesbeschrijving kan ook een assert statement worden opgenomen.

assert condition **report** message **severity** level;

Bijvoorbeeld:

assert a>b report "b is groter of gelijk a" severity warning;

Als niet aan de voorwaarde $a > b$ is voldaan wordt de melding *b is groter of gelijk a* afgedrukt en afhankelijk van de waarde van de severity (note, warning, error, failure) vervolgt de simulatie zich, of wordt het simuleren afgebroken. In dit boek is ook een voorbeeld opgenomen met een toepassing waarbij op setup en hold tijden van geheugen elementen getest kan worden.

Een faculteit van een getal is alleen gedefinieerd voor de natuurlijke getallen, echter in figuur 2.16 is de input van het type integer. Door middel van het assert statement *assert n>=0 report "invoer is kleiner dan 0" severity error;* wordt aangegeven dat er een negatief getal is aangeboden.

Het type integer is eigenlijk een te 'ruim' type voor de beschrijving van faculteit. Beter is het om alleen de gehele getallen groter of gelijk aan 0 te accepteren. Standaard heeft VHDL daarvoor het type, of beter het *subtype*, 'natural'.

2.4.1.3 Functies en procedures.

Ook VHDL laat het gebruik van functies en procedures toe. Deze kunnen in het declaratie gedeelte binnen een proces worden gedeclareerd. Een verschil tussen functies en procedures is dat een functie één resultaat oplevert terwijl een procedure een willekeurig aantal resultaten kan opleveren. Bij een functie mogen de parameters dus alleen van de mode 'in' zijn. Figuur 2.17 geeft een voorbeeld van het bepalen van het maximum van twee integer getallen. Het uitgangssignaal van entity MAXIMUM is MAX. Merk op dat een functie direct aan een signaal een toekenning kan doen, terwijl dit bij de procedure niet lukt. Daarom is er een hulp variabele TMP_MAX gedeclareerd. In hoofdstuk 3 zal worden aangegeven dat ook procedures, onder bepaalde omstandigheden, aan signalen waarden kunnen toekennen. Vaak zullen functies en procedures een zodanige opzet hebben dat deze algemeen bruikbaar zijn. Het is dan ook mogelijk deze onder te brengen in een aparte module, die *package* wordt genoemd. Op pagina 40 wordt hieraan meer aandacht besteed.

Op pagina 55 wordt uitvoerig ingegaan op de verschillen tussen functies en procedures.

```

entity maximum is
  port ( in1, in2      : in integer;
         max           : out integer);
  -- max is maximum van in1 en in2
end maximum;

architecture max_function of maximum is
begin
  process(in1,in2)
    function find_maximum (input1,input2: in integer)
      return integer is
        -- functieresultaat is maximum van input1 en input2
      begin
        if input1>input2
          then return input1;
          else return input2;
        end if;
      end find_maximum;
    begin
      max <= find_maximum(in1,in2);
    end process;
  end max_function;

  architecture max_procedure of maximum is
  begin
    process(in1,in2)
      procedure find_maximum (input1,input2: in integer;
                             outp : out integer) is
        -- levert het maximum van input1 en input2
      begin
        if input1>input2
          then outp := input1;
          else outp := input2;
        end if;
      end find_maximum;
      variable tmp_max : integer;
    begin
      find_maximum(in1,in2,tmp_max);
      max <= tmp_max;
    end process;
  end max_procedure;

```

figuur 2.17 Functies en procedures.

2.4.2 Impliciete processen.

In een architecture-beschrijving kunnen meerdere processen beschreven zijn. Een proces kan impliciet zijn beschreven door middel van (niet volledig):

- concurrent signal assignment statements
- concurrent assert statements
- concurrent procedure call (zie hoofdstuk 3)

2.4.2.1 Concurrent signal assignment statement.

Binnen de categorie concurrent signal assignment statements is wederom een opdeling te maken in:

- concurrent conditional signal assignment statement
- concurrent selected signal assignment statement

In figuur 2.18 worden de beide vormen gebruikt om de architecture van de multiplexer te beschrijven waarvan de entity gegeven is in figuur 2.15. De architecture-beschrijving CONDITIONAL_SIGNAL wordt intern vertaald naar een expliciete procesbeschrijving met 'if then elsif' zoals gegeven is in figuur 2.15. Zo wordt ook de architecture SELECTED_SIGNAL intern omgezet naar een expliciete procesbeschrijving met een case statement. Een bijzondere vorm van het conditional signal assignment statement is het statement zonder conditie, zoals al eerder gebruikt is, bijvoorbeeld y <= x; .

In de architecture SELECTED_SIGNAL is ook weer aangegeven dat 'when others' gebruikt kan worden om aan te geven wat er moet gebeuren met de niet genoemde selecties. In dit voorbeeld had het statement 'when others' eenvoudig vervangen kunnen worden door 'when "11" '. Merk ook het gebruik van de ',' en de ';' op in het selected signal assignment statement.

```
architecture conditional_signal of multiplexer is
begin
    outp <= in0 when sel1='0' and sel0='0' else
              in1 when sel1='0' and sel0='1' else
              in2 when sel1='1' and sel0='0' else
              in3;
end conditional_signal;

architecture selected_signal of multiplexer is
    subtype bv2 is bit_vector(1 downto 0);
begin
    with bv2'(sel1&sel0) select -- 7
        outp <= in0 when "00",
              in1 when "01",
              in2 when "10",
              in3 when others;
end selected_signal;
```

figuur 2.18 Conditional en selected signal assignment statement.

2.4.2.2 Concurrent assert statement.

```
architecture for_lus of faculteit is
begin
    process(n)
        variable prd : integer;
    begin
        prd := 1;
        for i in 1 to n loop
            prd := prd * i;
        end loop;
        fac <= prd;
    end process;

    assert n>=0 report "invoer is kleiner dan 0" severity error;
end for_lus;
```

figuur 2.19 Faculteit met concurrent assert statement.

Een assert statement wordt gebruikt om aan te geven of aan bepaalde condities is voldaan. Zo moet bij een register de data ingang gedurende de setup en hold tijd constant blijven en voor de entity faculteit moet de invoer altijd groter of gelijk aan nul zijn (figuur 2.16). Figuur 2.19 geeft aan hoe een concurrent assert statement wordt gebruikt om aan te geven dat niet aan deze voorwaarde is voldaan.

De concurrent assert uit figuur 2.19 is equivalent met de volgende expliciete procesbeschrijving:

```
process
begin
    assert n>=0 report "invoer is kleiner dan 0" severity error;
    wait on n;
end process;
```

⁷ zie voetnoot op pagina 17.

2.5 Struktuurbeschrijving met VHDL.

Een complex digitaal systeem beschrijven met and's, or's en inverters is niet gebruikelijk. In een dergelijk systeem kan meestal weer onderscheid gemaakt worden tussen de verschillende functionele blokken, zoals alu's (arithmetic logic units), geheugens etc. Niet alleen tijdens het beschrijven maar ook tijdens het ontwerpen van een digitaal systeem is het belangrijk dat geleidelijk aan meer detail kan worden toegevoegd.

Door middel van een struktuurbeschrijving kan hiërarchie in het ontwerp worden aangebracht. In figuur 2.3 is een architecture gegeven van de entity SR_LATCH. In deze architecture zijn de beide nor poorten als proces beschreven. Veronderstel dat er een entity NOR2 is, zoals gegeven in figuur 2.20. In de entity is omschreven wat de functie van deze entity is. In het algemeen zijn er een groot aantal van dergelijke entiteiten, zeg maar componenten, ter beschikking. Met dergelijke entiteiten is men in staat om weer grotere systemen te beschrijven, welke dan op hun beurt ook weer entiteiten zijn. In figuur 2.21 is dit geïllustreerd aan de hand van de sr-latch. De entity-beschrijving van deze latch is gegeven in figuur 2.3 en een 'bijna' struktuurbeschrijving is gegeven in figuur 2.21. *Een struktuurbeschrijving in VHDL is een opsomming van componenten en hoe deze met elkaar verbonden zijn.*

```
entity nor2 is
  generic (delay      : time := 5 ns);
  port (in1,in2       : in bit;
        outp          : out bit);
  -- nor poort met twee ingangen in1 en in2 en uitgang outp.
  -- en een generic vertraging van 5 ns.
end nor2;
```

figuur 2.20 Nor gate.

```
architecture bijna_struktuur of sr_latch is
  signal qi      : bit;
  signal qi_not  : bit := '1';
  component nor_gate_2
    generic (delay : time);
    port (in1,in2  : in bit;
          outp     : out bit);
  end component;
  for nor_g1:nor_gate_2 use entity work.nor2(gedrag);
  for nor_g2:nor_gate_2 use entity work.nor2(struktuur);
begin
  nor_g1:nor_gate_2
    generic map (delay)
    port map (r,qi_not,qi);

  nor_g2:nor_gate_2
    generic map (open)
    port map (s,qi,qi_not);

  q <= qi;
  q_not <= qi_not;
end bijna_struktuur;
```

figuur 2.21 Architecture voor de sr-latch.

Opmerkingen bij figuur 2.21

- Merk op dat er naast signalen ook componenten gedeclareerd kunnen worden. Er is een component NOR_GATE_2 gedeclareerd met dezelfde generic en in- en uitgangssignalen als de entity NOR2. In hoofdstuk 3 zal worden aangegeven dat dit niet zo hoeft te zijn.
- In de architecture-beschrijving zijn een tweetal instantiaties van component NOR_GATE_2 aanwezig gelabeld met NOR_G1 en NOR_G2. Labels voor

componentinstantiaties zijn verplicht. In de entity SR_LATCH (figuur 2.3) heeft de generic DELAY de waarde 10 ns, deze overruled daarmee de default waarde van 5 ns van de nor2 poort. Als een constante van waarde wordt gewijzigd moet ook de entity opnieuw geanalyseerd worden, bij het overrulen van een generic waarde is dit niet nodig. Hieruit wordt duidelijk dat een generic meer inhoudt dan alleen maar een constante.

Bij de instantiatie van nor_g2 wordt de default waarde gebruikt omdat het gereserveerde woord **open** is gebruikt in de generic map.

- Een configuratie-specificatie wordt gebruikt om aan te geven op welke entity met een daarbij behorende architecture een component gemapped moet worden. Voor NOR_G1 is deze configuratie-specificatie:

for nor_g1:nor_gate_2 use entity work.nor2(gedrag);

In woorden staat hier: kies voor component NOR_GATE_2 met label NOR_G1 de entity NOR2 met architecture GEDRAG. In het algemeen kunnen er een groot aantal entiteiten aanwezig zijn die bijvoorbeeld in een bibliotheek zijn ondergebracht. Daarom moet in de configuratie-specificatie nog worden aangegeven waar een entity zich bevindt, *work* geeft aan dat dit in de 'eigen' directory is. Dit wordt nog uitvoeriger behandeld.

- Voor NOR_G1 en NOR_G2 worden beide de entity NOR2 gebruikt maar verschillende architecturen. De component NOR_G1 gebruikt een gedragsbeschrijving terwijl NOR_G2 een structuurbeschrijving gebruikt. Bij één entity kunnen meerdere architecturen behoren, door middel van de configuratie-specificatie wordt aangegeven welke architectuur wordt gebruikt.
- De instantiatie van NOR_G1 en NOR_G2 is één statement, d.w.z. dat er één ';' aan het einde van de port map staat (en pas op: dus niet achter de generic map).
- Vaak zal voor een component steeds dezelfde architectuur worden gebruikt. Het opsommen van al deze componenten kan achterwege worden gelaten door gebruik te maken van *for all*, bijvoorbeeld

for all:nor_gate_2 use entity work.nor2(gedrag);

Nu wordt voor alle componenten NOR_GATE_2 dus entity NOR2 en architecture GEDRAG gebruikt. Uiteraard is het mogelijk dat van de honderd identieke componenten er bijvoorbeeld voor één component een andere architecture gebruikt gaat worden, moeten er dan honderd configuratie-specificatie worden gebruikt? Gelukkig niet:

for nor_g1:nor_gate_2 use entity work.nor2(gedrag);

for others:nor_gate_2 use entity work.nor2(struktuur);

- In dit voorbeeld is er bewust voor gekozen om de naam van de declaratie van de component af te laten wijken van de entity die er op wordt gemapped. Veelal zal dit niet gebeuren. Indien de naam van de component-declaratie overeenkomt met de naam van de entity mag een configuratie-specificatie achterwege blijven. In een dergelijke situatie wordt de laatst geanalyseerde architecture voor de entity genomen, de default configuratie. In hoofdstuk 3 zal meer aandacht besteed worden aan het analyse proces.
- De architecture heet BIJNA_STRUKTUUR. Een echte structuurbeschrijving is een opsomming van componenten en hun verbindingen. De componenten NOR_G1 en NOR_G2 voldoen aan deze voorwaarde, echter er zijn ook een tweetal concurrent signal assignment statements. Strikt genomen is het daarom geen structuurbeschrijving! Een echte structuurbeschrijving kan bijvoorbeeld worden verkregen door een component en entity OUTPUT_BUFFER te creëren die de

concurrent signal assignment statements vervangt. Een fraaiere oplossing is het gebruik van de mode *buffer* in plaats van de modes *out* of *inout*, maar dit geeft weer andere problemen (zoals in de volgende paragraaf wordt behandeld).

2.5.1 Ingangen, uitgangen en bidirectionele poorten.

In de tot nu toe gebruikte entity-beschrijvingen werd door middel van *in* en *out* aangegeven of een signaal respectievelijk in- of uitgang is. Voor het beschrijven van een bidirectionele poort is er ook nog de mode *inout*. Er gelden een aantal restricties met betrekking tot het gebruik van deze modes *in*, *out* en *inout* welke geïllustreerd zijn in figuur 2.22. De *default mode* is *IN*.

```
entity modes is
  port(input  : in bit;
        output : out bit;
        io    : inout bit);
end modes;

architecture fout_gebruik of modes is
begin
  input <= io;
  -- fout: een signaal van mode 'in' mag alleen worden gelezen
  io    <= output;
  -- fout: een signaal van mode 'out' mag alleen een waarde
  -- toegekend krijgen, ervan lezen is niet toegestaan
  io    <= input;    -- toegestaan
  output <= input or io; -- toegestaan
end fout_gebruik;
```

figuur 2.22 Gebruik van de modes 'in', 'out' en 'inout'.

Het is nu duidelijk waarom in de architecture-beschrijving van de sr-latch in figuur 2.21 de lokale signalen *qi* en *qi_not* noodzakelijk zijn, het signaal van de mode *OUT* mag alleen een waarde toegekend krijgen. Om de lokale signalen te vermijden zouden de beide uitgangssignalen mode *inout* kunnen krijgen, maar is dit wenselijk? Immers indien men niet geïnteresseerd is in de architecture dan lijkt het alsof de uitgangen van de latch ook als ingang gebruikt kunnen worden! Eigenlijk is er nog de behoefte aan een mode die het wel toelaat dat een mode 'out' intern gelezen kan worden, maar die slechts één source mag hebben, en dit is mode *buffer*.

```
entity sr_latch is
  generic (delay : time := 10 ns);
  port ( s,r      : in bit;
        q,q_not  : buffer bit);
  ..
```

In een architecture van de entity *sr_latch* mag zowel gelezen worden van als geschreven worden naar *q* en *q_not*, terwijl andere entiteiten niet aan de beide 'buffer' poorten een toekenning mogen doen. Merk op dat nu ook de locale declaraties *qi* en *qi_not* overbodig zijn geworden. Echter, op een mode *buffer* mogen geen andere modes worden verbonden dan mode *buffer*. Dit betekent dus dat bovenstaande entity-beschrijving niet past bij figuur 2.21. Dit laatste is vervelend, en betekend in de praktijk dat mode *buffer* minder vaak wordt toegepast.

Tenslotte is er nog de mode *linkage*. Deze mode kan gezien worden als een doorgeefluik van een signaal van een hoger niveau naar een lager niveau, waarbij de entity zelf dit object niet mag lezen en niet mag schrijven. Op pagina 86 wordt uitvoeriger ingegaan op deze mode. Geconstateerd moet worden dat ook deze mode zelden wordt gebruikt.

In de onderstaande tabel is aangegeven welke mode een actuele object moet hebben om correct verbonden te worden met formele parameter.

<i>mode formele parameter</i>	<i>toegestane mode actuele parameter</i>
in	in, inout, buffer
out	out, inout
inout	inout
buffer	buffer
linkage	in, out, inout, buffer, linkage

2.5.2 Regelmatige structuren.

Indien hardware uit identieke cellen bestaat die op een regelmatige wijze met elkaar verbonden zijn kan dat door middel van een *generate statement* op eenvoudige wijze worden beschreven.

De generate variable wordt niet gedeclareerd en kan alleen binnen de generate statement worden gelezen. Generate statements mogen genest voorkomen waarbij een geneste generate variabele afhankelijk kan zijn van de generate statement waar het zelf deel van uit maakt, bijvoorbeeld:

```
generate1:for i in 0 to 8 generate
  generate2:for j in i to 8 generate
    <concurrent statements>
  end generate generate2;
end generate generate1;
```

Figuur 2.23 geeft een voorbeeld van een n-bits output_buffers. Er is tevens een nieuw type geïntroduceerd, te weten de *bit_vector*. De bit_vector (8 downto 1) is een vector van acht bits waarbij de indexering van links naar rechts gaat, van 8 naar 1.

Ook kan er nog een conditie worden opgenomen in de generate statement, bijvoorbeeld:

```
cond : if aantal>0 generate
  .....
end generate;
```

```

entity output_buffer is
  port    ( input : in bit;
            output: out bit);
  -- een buffer.
end output_buffer;

architecture gedrag of output_buffer is
begin
  output <= input;
end gedrag;

entity n_output_buffers is
  generic (width : natural :=8);
  port    (inp  : in bit_vector(width downto 1);
            outp : out bit_vector(width downto 1));
end n_output_buffers;

architecture structuur of n_output_buffers is
  component output_buffer
    port    ( input : in bit;
            output: out bit);
  end component;
  -- geen configuratie-specificatie, dus de laatst
  -- correct geanalyseerde architecture van de entity nemen.
begin
  width_buffers: for i in width downto 1 generate
    outp_buf: output_buffer port map(inp(i), outp(i));
  end generate width_buffers;
end structuur;

```

figuur 2.23 N-bits buffer met een generate statement.

2.5.3 Koppeling van actuele met formele parameters.

Figuur 2.21 geeft een architecture voor een sr-latch waarbij een component NOR_GATE_2 gedeclareerd is met de formele poorten IN1, IN2 en OUTP. Bij de instantiatie van NOR_G1 worden deze formele poorten gekoppeld aan de actuele poorten, resp. r, Q1_NOT, Q1. *In plaats van de mapping van de actuele op de formele parameters te bereiken door deze op de overeenkomstige positie te plaatsen (“positional association”), kan deze koppeling ook expliciet worden gemaakt door de formele parameter te koppelen met de actuele parameter, m.b.v. de '=>' operator (“named association”).*

De instantiatie van nor_g1 kan er dan als volgt uit zien:

```

nor_g1:nor_gate_2
  generic map ( delay=>delay )
  port map ( in1=>r, in2=>q1_not, outp=>q1 );

```

maar ook:

```

nor_g1:nor_gate_2
  generic map ( delay=>delay )
  port map ( r, outp=>q1, in2=>q1_not );

```

Merk op dat de laatste een combinatie is van beide methoden. Bij een dergelijke combinatie moeten eerst de positionele koppelingen gegeven worden. Niet gebruikte uitgangen worden op een 'nette' manier afgesloten door middel van het gereserveerde woord OPEN, dit voorkomt het gebruik van dummy signalen.

Voorbeeld: als de *q_not* uitgang van de *sr-latch* uit figuur 2.3 niet aangesloten hoeft te zijn, en voor het gemak wordt verondersteld dat de component-declaratie gelijk is aan de entity-declaratie, dan wordt de instantiatie:

```

q_not_niet_gebruikt:sr_latch
  generic map(delay => 20 ns)
  port map(

```

```

s => set_ingang;
r => reset_ingang;
q => uitgang;
q_not => open);

```

2.6 Data types.

Voor het beschrijven van hardware wordt vaak volstaan met de logische niveaus '0' en '1'. Dit kan te beperkt zijn, zo is het handig dat met integers gewerkt kan worden op een hoger abstractie niveau. VHDL kent dan ook types die in grote lijn overeenkomen met die van andere sterk getypeerde imperative talen.

De verschillende data types zijn:

- scalar types
- composite types
- access types (in andere talen ook wel pointers genoemd)
- file types
- subtypes

Het gebruik van het access type en file type wordt resp. behandeld op de pagina's 78 en 80.

2.6.1 Scalar types.

VHDL kent de scalaire types integer, floating point, enumeration en fysieke types. Voor al deze types geldt een ordening, d.w.z. dat bijvoorbeeld de operator '<' gedefinieerd is.

2.6.1.1 Integer, floating point en enumeration type.

Het integer bereik dekt een subset van de gehele getallen. De standaard eist dat tenminste het volgende bereik wordt ondersteund:

type integer is range $-(2^{31}-1)$ to $(2^{31}-1)$;

voorbeelden:

```

constant i : integer := 20;    -- initialisatie verplicht (behalve bij
                                -- een 'deferred constant declaration')
signal j : integer := 0;       -- initialisatie is optioneel
variable k : integer := 3;     -- initialisatie is optioneel

```

Ook voor de floating point types geldt een soortgelijke beperking met tevens een minimale nauwkeurigheid, waarop hier verder niet wordt ingegaan.

type real is range $-1e38$ to $1e38$;

voorbeelden:

```

constant i : real := 20.0;
signal j : real := 0.3;        -- initialisatie is optioneel
variable k : real := 3.0;      -- initialisatie is optioneel

```

Naast deze types zijn de volgende enumeraties standaard in de VHDL omgeving aanwezig.

type bit is ('0','1');

type boolean is (false,true);

type character is (.. opsomming van de 128 ASCII karakters ..);

type severity_level is (note,warning,error,failure);

Het type `severity_level` wordt gebruikt in het `assert` statement.

Ook is het mogelijk nieuwe types te declareren. Zo is het type `bit` te beperkt om echte hardware te modelleren. Hoe moet van een tri-state de hoogohmige impedantie worden gemodelleerd? De gebruiker kan expliciet het volgende type declareren:

type drie_waardige_logica **is** ('0','1','Z');

De objecten - constanten, variabelen en signalen - kunnen ook van dit type zijn. De logische functies `and`, `or` etc. zijn echter alleen gedefinieerd voor het type `bit`, `bit_vector` en `boolean`. Aangezien VHDL een sterk getypeerde taal is is het volgende dus niet correct:

signal a,b,c : drie_waardige_logica;

..

`c <= a and b;` -- fout, `a` en `b` niet van het type `bit`, waarvoor de `and` operator is gedefinieerd.

Op pagina 38 is aangegeven dat het mogelijk is om een functie, zoals de `and`, voor meerdere types geschikt te maken.

2.6.1.2 Fysieke types.

Ongemerkt is er al gebruik gemaakt van een fysiek type, namelijk het type *time*. In voorgaande voorbeelden is er gewoon vanuit gegaan dat de fysieke eenheid *ns* (nanoseconde) bekend is. Dit is mogelijk omdat het type is gedefinieerd in de standaard omgeving als (waarbij ook nu weer het gegeven bereik een minimum vereiste is):

type time **is range** -(2**31-1) **to** (2**31-1) --⁸

units

```
fs;           -- primary unit
ps  = 1000 fs; -- secondary units
ns  = 1000 ps;
us  = 1000 ns;
ms  = 1000 us;
sec = 1000 ms;
min = 60 sec;
hr  = 60 min;
```

end units;

Dankzij deze standaard declaratie kan dus in een beschrijving '10 *ns*' worden gebruikt. De waarde van `fs` is bekend binnen VHDL, 10 *ns* wordt vertaald naar: 10*1000*1000 *fs*. 'fs' wordt de primary unit genoemd en de overige heten secondary units. Nieuwe fysieke types kunnen ook gedeclareerd worden, bijvoorbeeld voor de elektrische stroom, weerstand en spanning.

⁸ Uitgaande van het maximum van $2^{31}-1$ (=2147483647) kan het grootste tijdstip dus maar circa 2.1 ms zijn! Echter de language reference manual geeft aan dat een simulator de kleinste stap grootte op mag geven dus bijvoorbeeld 1 ns in plaats van 1 fs. Verder is het opvallend dat de tijd ook negatief kan zijn, dit is noodzakelijk als met verschillen van tijdstippen wordt gewerkt.

Voorbeeld:

type weerstand **is range** 0 to 10e9

units

```
ohm;                -- ohm
kohm    = 1000 ohm;  -- kilo ohm
mohm    = 1000 kohm; -- mega ohm
```

end units;

type stroom **is range** -10e9 to 10e9

units

```
ua;                -- micro ampere
ma  = 1000 ua;     -- milli ampere
a   = 1000 ma;     -- ampere
```

end units;

type spanning **is range** -10e9 to 10e9

units

```
uv;                -- micro volt
mv  = 1000 uv;     -- milli volt
v   = 1000 mv;     -- volt
kv  = 1000 v;      -- kilo volt
```

end units;

Op deze wijze is het bijvoorbeeld mogelijk de 'wet van ohm' te schrijven:

```
variable i : stroom;
variable v : spanning;
variable r : weerstand;
..
v := i * r;
v := 5 ma * 10 kohm;
```

Uiteraard moet ook nu weer de operator '*' gedefinieerd zijn voor deze nieuwe types wat in de standaard omgeving niet het geval is, dit wordt behandeld op pagina 38.

Als de positie van een component beschreven moet worden zou hiervoor bijvoorbeeld het fysieke type lengte gedefinieerd kunnen worden.

2.6.2 Composite types.

Een tweetal composite types zijn mogelijk: de *records* en de *arrays*. Beide types zijn ook weer gedefinieerd voor constanten, signalen en variabelen. Een voorbeeld van een record is bijvoorbeeld de positie van een component, dit kan het volgende zijn:

type position **is record**

x_coordinate, y_coordinate : lengte;

end record;

Enkele voorbeelden van het gebruik zijn:

```
variable a:position:=(12 mm, -10 mm);9
variable d:position:=(y_coordinate => 2 mm, x_coordinate => 3 mm);
variable b,c:position;
..
b.x_coordinate:=100 mm; -- alleen de x_coordinate krijgt een waarde
c:=a;    -- a.x en a.y worden resp. toegekend aan c.x en c.y
```

Een array type kan in VHDL *unconstrained* of *constrained* zijn. Bij een unconstrained array is de grootte, het aantal elementen, nog niet vastgelegd. VHDL heeft in de standaard omgeving twee unconstrained array types:

***type string is array** (positive **range** <>) **of** character;*
***type bit_vector is array** (natural **range** <>) **of** bit;*

Door 'range <>' wordt aangegeven dat het bereik nog niet vastligt. Positive is een subtype van integers die alleen de positieve getallen bevat en natural is een subtype van integers die alle getallen groter of gelijk aan nul bevat (zie ook pagina 33).

Voorbeelden van constrained arrays:

```
variable a : bit_vector(3 downto 0);
variable b : bit_vector(5 to 10);
variable c : bit_vector(-5 to 10); -- FOUT, index buiten bereik.
variable e : string := "dit is tekst"; -- initialisatie bepaalt lengte van vector.
```

Bit_vectoren worden vaak gebruikt om een getal met een ander grondtal dan 2 (binair) te representeren. Daarom voorziet VHDL in conversies voor binaire (B), octale (O) en hexadecimale (X) getallen naar het type bit_vector. In de toekenning mag een '_' karakter voorkomen om de leesbaarheid te vergroten, echter niet meer dan één achter elkaar.

Voorbeelden (representeren alle 100 decimaal):

```
variable honderd_binair : bit_vector (7 downto 0):= B"0110_0100";
variable honderd_octaal : bit_vector (8 downto 0):= O"144";
variable honderd_hex : bit_vector (7 downto 0):= X"64";
```

De initiële waarde van een signaal en een variabele, indien niet expliciet opgegeven, is het meest linkse element van het type. Dus variable a, zoals hiervoor is gedeclareerd, heeft als initiële waarde B"0000". Veronderstel dat alleen a(2) '1' moet zijn dan kan dit als volgt worden gedeclareerd:

```
variable a : bit_vector(3 downto 0) := B"0100";
```

De toekenning gebeurt hier door de elementen van het type op de juiste plaats te zetten ("positional association"). Ook het volgende levert hetzelfde resultaat op:

```
variable a : bit_vector(3 downto 0) := (0 | 1 | 3 => '0'; 2 => '1');
```

⁹ Indien het veld van een record (of een array) uit één element bestaat moet een toewijzing plaats vinden via "named association". Voorbeeld:

```
type position is record
    x : lengte;
end record;
variable a : position := ( x => 12 mm);
```

De volgende declaratie is fout: **variable** a : position := (12);

Dit betekent dat op de posities 0, 1 en 3 de waarde '0' moet komen en op positie 2 de waarde '1'. Nog fraaier kan dit worden beschreven als:

```
variable a : bit_vector(3 downto 0) := (2 => '1' ; others => '0');
```

Het gereserveerde woord OTHERS kent aan alle niet expliciet toegekende posities de waarde '0' toe. Dit is met name interessant indien er functies of procedures geschreven worden waarbij formele parameters nog unconstrained zijn.

Het voorgaande illustreert ook weer het gebruik van een koppeling door middel van de plaats (“positional association”) en een koppeling door middel van de naam, in dit geval de positie in het array (“named association”).

Naast deze standaard unconstrained arrays kunnen ook nieuwe unconstrained arrays worden gedeclareerd. Zo is al eerder een type drie_waardige_logica gedeclareerd als:

```
type drie_waardige_logica is ('0','1','Z');
```

Van dit type kan dan een vector worden gemaakt door:

```
type drie_waardige_vector is array (natural range <>)
                                of drie_waardige_logica;
```

De arrays mogen ook multidimensional zijn en/of elementen van een array zijn weer arrays. Enkele voorbeelden:

```
type byte is array (7 downto 0) of bit;
type memory1 is array (0 to 127) of byte;
type memory2 is array (0 to 127,7 downto 0) of bit;
signal mem1 : memory1;
signal mem2 : memory2;
..
mem1(2) <= B"0001_1100";
mem2(2) <= B"0001_1100"; -- FOUT
mem2(2,0) <= '0';
mem1(2,3 downto 0) selecteert van element 2 de elementen 3 downto 0. Dit
wordt een slice genoemd en moet altijd een aaneengesloten één-dimensionale rij
zijn.
```

2.6.2.1 Alias.

Veronderstel dat een processor wordt gemodelleerd in VHDL waarbij het instructie formaat als volgt is gedefinieerd:

```
variable instructie : bit_vector (15 downto 0);
```

met op de posities

15 t/m 8	de uit te voeren operatie
7 t/m 4	de eerste operand
3 t/m 0	de tweede operand

Het lezen van de operatie code (op_code) en de beide operanden zal dan als volgt beschreven kunnen worden:

```
op_code := instructie(15 downto 8);
op1 := instructie(7 downto 4);
op2 := instructie(3 downto 0);
```

Als de variabele instructie veranderd moet worden ziet dat er als volgt uit: instructie(7 **downto** 4):= Door middel van een *alias* kunnen dergelijke selecties worden voorkomen waardoor de leesbaarheid wordt verhoogd. Een alias legt als het ware een *venster* op het oorspronkelijke object. Voor dit voorbeeld zijn de volgende drie aliases handig:

alias op_code : bit_vector(7 **downto** 0) **is** instructie(15 **downto** 8);

alias op1 : bit_vector(3 **downto** 0) **is** instructie(7 **downto** 4);

alias op2 : bit_vector(3 **downto** 0) **is** instructie(3 **downto** 0);

Er wordt door een alias geen nieuw object gedeclareerd. Indien bijvoorbeeld aan op1 een waarde wordt toegekend wordt dit direct in instructie(7 **downto** 4) weggeschreven.

2.6.2.2 Array in omgekeerde volgorde.

Veronderstel dat variable a als volgt is gedeclareerd *variable a : bit_vector(0 to 7)* en dat de elementen in omgekeerde volgorde moeten komen te staan. Het ligt voor de hand om dit als volgt te beschrijven a := a(7 **downto** 0); Dit is niet toegestaan omdat de vector gedeclareerd is een stijgende rij en dan mogen ook alleen de slices een stijgende rij vormen. Een oplossing is bijvoorbeeld het volgende:

for i **in** 0 **to** 7 **loop** a := a (7-i); **end loop**;

2.6.3 Subtypes.

De functie 'faculteit', zoals eerder gedeclareerd is op pagina 18, heeft als input de formele parameter n van het type integer. Voor dit type zijn allerlei operatoren gedefinieerd zoals delen, vermenigvuldigen etc. Echter de faculteit is niet gedefinieerd voor een negatief getal. Een assert statement was dan ook toegevoegd om dit te signaleren. Een subtype lost dit probleem veel fraaier op. Een subtype 'erft' namelijk alle operatoren van het type waarvan het is afgeleid, maar niet alle elementen van het oorspronkelijke type hoeven voor te komen in het subtype. VHDL heeft standaard twee subtypes:

subtype natural **is** integer **range** 0 **to** integer'**high**;

subtype positive **is** integer **range** 1 **to** integer'**high**;

“ ‘high “ is een attribuut dat de bovengrens van een type als resultaat geeft. Het type integer is implementatie afhankelijk. Dankzij dit attribuut is het mogelijk dat ook de daarvan afgeleide subtypes implementatie afhankelijk zijn.

De entity faculteit moet dan als invoer een signaal van het type natural krijgen.

In hoofdstuk 3 zal nog een andere toepassing worden gegeven van een subtype.

2.7 Operatoren.

gedefinieerde operatoren:		
groep	symbool	functie
aritmetisch (binary)	+	optellen
	-	aftrekken
	*	vermenigvuldigen
	/	delen
	mod	modules ¹⁰
	rem	rest
	**	macht verheffen
aritmetisch (unary)	+	teken
	-	teken
	abs	absolute waarde
relationeel	=	gelijk
	/=	ongelijk
	<	kleiner dan
	>	groter dan
	<=	kleiner dan of gelijk
	>=	groter dan of gelijk
logische (binary)	and	and
	or	or
	nand	nand
	nor	nor
	xor	xor
logische (unary)	not	complement
concatenatie	&	

figuur 2.24 Overzicht van de operatoren.

Figuur 2.24 geeft een overzicht van de standaard operatoren. Een enkele opmerking moet hierbij nog worden gemaakt:

- Een fysiek type en een integer of floating point type kan ook worden gebruikt bij het vermenigvuldigen en delen.

¹⁰ De integer deling en de rest zijn gedefinieerd d.m.v. de volgende relatie:

$$A = (A/B)*B + (A \text{ rem } B)$$

waarin het teken van (A rem B) gelijk is aan het teken van A en de absolute waarde kleiner dan de absolute waarde van B.

De modules operatie is zodanig gedefinieerd dat (A mod B) het teken heeft van B en de absolute waarde kleiner is dan de absolute waarde van B, en bovendien moet voor een integer N de volgende relatie gelden:

$$A = B*N + (A \text{ mod } B)$$

Voorbeelden:

5	rem	3	=	2
5	mod	3	=	2
(-5)	rem	3	=	-2
(-5)	mod	3	=	1
(-5)	rem	(-3)	=	-2
(-5)	mod	(-3)	=	-2
5	rem	(-3)	=	2
5	mod	(-3)	=	-1

voorbeeld:

```
variable tijd,totaal : time;  
variable veelvoud : integer;
```

..

```
totaal := veelvoud * tijd;
```

Zie voor een voorbeeld figuur 2.27.

- De logische operatoren, behalve de xor, zijn 'short circuit'. Dit betekent dat als de linker operand voldoende is om het resultaat te bepalen, de operand rechts ervan niet meer wordt geëvalueerd. Voor een and functie wordt de operand rechts niet meer geëvalueerd indien de operand links van de and een '0' is.
- De logische operatoren zijn gedefinieerd voor de types boolean, bit en bit_vector.
 '1' **and** '1' -- geeft '1'
 "0101" **and** "0110" -- geeft "0100"
- Voor 'macht verheffen' geldt dat de rechter operand alleen van het type integer mag zijn en dus geen real. De linker operand mag wel een real zijn.

2.8 Attributen.

Met attributen kan toegang worden verkregen tot de eigenschappen van objecten. In een bijlage is een overzicht gegeven van alle standaard attributen die VHDL kent. Ook is het mogelijk nieuwe attributen te definiëren.

2.8.1 Attributen voor arrays.

De attributen worden geïllustreerd aan de hand van de volgende gegevens:

```
variable stijgend : bit_vector(4 to 7):= B"0001";  
variable dalend : bit_vector(7 downto 4):= B"0001";
```

'left

Geeft linker index van array.

stijgend**'left** is dus 4

dalend**'left** is dus 7

'right

Geeft rechter index van array

'high

Geeft hoogste index van array

stijgend**'high** is dus 7

dalend**'high** is dus 7

'low

Geeft laagste index van array

'range

Geeft alle indexen van links naar rechts.

stijgend**'range** levert op 4 **to** 7

dalend**'range** levert op 7 **downto** 4

'reverse_range

Geeft alle indexen van rechts naar links.

stijgend **'reverse_range** levert op 7 **downto** 4

dalend **'reverse_range** levert op 4 **to** 7

```
function sommatie_1 (inp: bit_vector)
  return natural is
    variable number : natural := 0;
begin
  for i in inp'left to inp'right loop
    if inp(i)='1' then number:=number+1; end if;
  end loop;
  return number;
end sommatie_1;

function sommatie_2 (inp: bit_vector)
  return natural is
    variable number : natural := 0;
begin
  for i in inp'range loop
    if inp(i)='1' then number:=number+1; end if;
  end loop;
  return number;
end sommatie_2;
```

figuur 2.25 Bepalen van het aantal enen in een bit_vector.

De functies sommatie_1 en sommatie_2 in figuur 2.25 illustreert het gebruik van een attribuut in combinatie met een unconstrained array. Aangezien de formele parameter unconstrained is kan geen vaste grens voor de "for loop" worden aangegeven. De functie sommatie_2 is algemener dan de functie sommatie_1, immers deze is geschikt voor stijgende en dalende indicering van *inp*.

2.8.2 Enkele attributen voor signalen.

Er zijn attributen die op zich weer signalen zijn, zogenaamde impliciete signalen en attributen die een functie zijn. Dit is een belangrijk verschil. In een subprogram mogen bijvoorbeeld geen attributen worden gebruikt welke op zich een signaal zijn. Dus het volgende is niet toegestaan:

```
procedure foute_procedure (signal a : in bit; signal b : out boolean, t : time) is
begin
  b <= a'stable(time);
end foute_procedure;
```

Verder zijn er attributen die gevoelig zijn voor een transactie op een signaal, ongeacht of dit signaal van waarde verandert of niet, terwijl andere attributen alleen gevoelig zijn voor een event, een verandering, van het signaal.

S'transaction Een *signaal* van het type bit dat van waarde verandert iedere keer als er een transaction op het signaal S is.

Indien de waveform van signaal SGN gelijk is aan: '1', '0' @ 10 ns, '1' @ 20 ns, '1' @ 30 ns. Dan zal, hoewel het signaal SGN zelf niet van waarde verandert, ook op tijdstip 30 ns het signaal SGN'transaction van waarde veranderen.

De initiële waarde van dit signaal is niet gedefinieerd!

S'event

Een *functie* met een boolean resultaat die aangeeft of er gedurende de simulatie cyclen een event op signaal S is opgetreden.

S'delayed(t)	Een <i>signaal</i> van hetzelfde type als S. Het signaal is gelijk aan het signaal S echter met de opgegeven tijdsduur verschoven in de tijd. Indien de parameter '(t)' ontbreekt of t is 0 is de vertraging gelijk aan delta-delay.
S'active	Een <i>functie</i> met een boolean resultaat die aangeeft of er gedurende de simulatie cyclus een transaction op signaal S is geweest.
S'last_event	Een <i>functie</i> die de verstreken tijd aangeeft tot het optreden van het laatste <i>event</i> op een signaal.
S'last_active	Een <i>functie</i> die de verstreken tijd aangeeft tot het optreden van de laatste <i>transaction</i> op een signaal.
S'last_value	Een <i>functie</i> die de vorige waarde van S teruggeeft, vlak voor de laatste verandering.
S'quiet(t)	Een <i>signaal</i> van het type boolean. Geeft aan of aan het signaal S gedurende tijdsduur t een toekenning is gedaan. Indien de parameter t ontbreekt of t is 0, dan heeft de tijdsduur betrekking op de laatste simulatie cyclus.
S'stable(t)	Een <i>signaal</i> van het type boolean. Geeft aan of aan het signaal S gedurende tijdsduur t een event is opgetreden. Indien de parameter t ontbreekt of t is 0, dan heeft de tijdsduur betrekking op de laatste simulatie cyclus.

Er is een belangrijk verschil tussen het attribuut quiet en het attribuut stable. De eerste is ook gevoelig voor een toekenning aan een signaal indien het dezelfde waarde betreft.

```

y <=  '1',
      '1' after 20 ns,
      '0' after 40 ns;
s <=  y'stable (10 ns);
q <=  y'quiet (10 ns);

```

Het signaal s zal op tijdstip 40 ns gedurende een tijdsduur van 10 ns false worden, terwijl het signaal q ook op tijdstip 20 ns gedurende een tijdsduur van 10 ns false is.

Er is een belangrijk verschil tussen attributen welke een functie zijn en attributen die een signaal zijn. Bijvoorbeeld in het wait on statement is alleen 'gevoelig' voor signalen, dus:

```

wait on clk'stable;      is correct
wait on clk'event;      is wel correct VHDL, maar het proces zal zijn
                           executie niet vervolgen na een event op het signaal CLK!

```

In een subprogram (functie/procedure) mogen geen attributen worden gebruikt welke een signaal zijn. In de volgende tabel is een overzicht gegeven van de attributen voor signalen.

signaal	functie	type van het attribuut
'transaction		bit
'stable	'event	boolean
'quiet	'active	boolean
	'last_event	time
	'last_active	time
'delayed	'last_value	het zelfde type als het signaal waaraan dit attribuut is gekoppeld.

2.9 Overloading van functies en procedures.

De logische operatoren `and`, `or` etc. zijn standaard gedefinieerd in VHDL voor o.a. types `bit` en `bit_vector`. Maar dit type is te beperkt om hardware mee te modelleren. Daarom werd dan ook een driewaardige logica gegeven: *type drie_waardige_logica is ('0','1','Z');* en *type drie_waardige_vector is array (natural range <>) of drie_waardige_logica;* Voor dit type zijn uiteraard geen operatoren gedefinieerd zoals de `and`, `or`, etc¹¹. Overloading maakt het mogelijk een al gebruikte naam opnieuw te gebruiken. Afhankelijk van de types wordt tijdens de analyse (in sommige VHDL omgevingen ook wel 'compilatie' genoemd) bepaald welk subprogram (functie of procedure) gekozen moet worden. In de standaard omgeving wordt ook al gebruik gemaakt van overloading, immers de `and` operator is gedefinieerd voor het type `bit` maar ook voor `bit_vector`! In figuur 2.26 is een beschrijving gegeven van de operator `and` voor de driewaardige logica. Enkele opmerkingen met betrekking tot deze beschrijving:

- De dubbele quote ' " ' om de functienaam geeft aan dat de operator een infix-operator is. Er is sprake van een infix-operator indien de operator tussen de beide operanden in staat, zoals bijvoorbeeld bij de `and`. Dit betekent niet dat door het plaatsen van dubbele quotes om een functienaam deze automatisch infix wordt! Alle infix-operatoren zijn namelijk al gedefinieerd in VHDL. Het is niet mogelijk om nieuwe infix-operatoren te definiëren. Als een infix-operator overload moet worden moet daarbij altijd gebruik worden gemaakt van de dubbele quote.

Wel mag in een beschrijving voor infix-operatoren de volgende constructie worden gebruikt:

```
y <= "and"(a,b);
```

- De functie *and* met als types `drie_waardige_vector` maakt herhaald gebruik van de *and* op één element van het type `drie_waardige_logica`. Dit herhalen wordt uitgevoerd door de vector `A_INTERN` van links naar rechts te doorlopen. Indien de aliases niet worden gebruikt moet de indexering van de vectoren `a` en `b` gelijk zijn anders ontstaat er een fout in de index! Het gebruik van de aliases converteert a.h.w. willekeurige indices naar een standaard vorm.
- Tijdens de analyse van een VHDL beschrijving wordt gezocht naar een functie en of er een typering is die ermee overeenkomt.

voorbeelden:

```
variable a,b,c : drie_waardige_logica;
```

```
variable av,cv : drie_waardige_vector(1 to 3);
```

```
variable bv : drie_waardige_vector(6 downto 4);
```

¹¹ Als er een type-declaratie wordt gegeven worden ook een aantal impliciete functies gedefinieerd, o.a. de `"="` operator. Dit is ook noodzakelijk om twee waarden met elkaar te kunnen vergelijken!

```

variable d,e,f : bit;
..
f := e and d;
c := e and b;    -- FOUT
cv := av and a;  -- FOUT
c := a and b;
cv := av and bv;

```

```

function "and" (a,b : drie_waardige_logica)
    return drie_waardige_logica is
begin
    if a='0' or b='0'
    then return '0';
    else return '1';
    end if;
end "and";

function "and" (a,b : drie_waardige_vector)
    return drie_waardige_vector is
    alias a_intern : drie_waardige_vector (1 to a'length) is a(a'range);
    alias b_intern : drie_waardige_vector (1 to b'length) is b(b'range);
    variable res : drie_waardige_vector (1 to a'length);
begin
    for i in a_intern'range loop
        res(i) := a_intern(i) and b_intern(i);
    end loop;
    return res;
end "and";

```

figuur 2.26 Overloading van operatoren.

Figuur 2.27 geeft een ander voorbeeld waarbij de fysieke types elektrische stroom, weerstand en spanning worden gebruikt voor de wet van ohm. In de functie vindt een normering plaats door de stroom i te delen door 1 uA, dit levert dan een dimensieloos getal op, evenzo wordt dit voor de weerstand r gedaan. Het product van beide wordt dan vermenigvuldigd met 1 uV, zodat de uitkomst de dimensie heeft van uV. Waarom is de '*' operator tweemaal gedeclareerd?

Op pagina 30 zijn de type-declaraties van weerstand, stroom en spanning gegeven. De primary unit in de type-declaratie voor stroom is 'ua'. In de overloaded function wordt dan ook de stroom gedeeld door een unit van het type 'stroom' zodat een 'dimensieloos' getal overblijft ($i / 1 \text{ ua}$). Ditzelfde gebeurt ook voor de weerstand ($r / 1 \text{ ohm}$). Het product van beide getallen moet echter van het type spanning zijn. Dus moet dit worden vermenigvuldigd met '1 uv'. Waarom '1 uv'? Veronderstel dat i is 1 ma en r is 2 ohm dan is het resultaat gelijk aan:

$(1000 \text{ ua} / 1 \text{ ua}) * (2 \text{ ohm} / 1 \text{ ohm}) = 1000 * 2 = 2000$.

Dit moet uiteindelijk 2000 uv worden, daarom nog vermenigvuldigen met 1 uv.

```

function "*" (i : stroom; r: weerstand)
    return spanning is
begin
    return 1 uv * ( (i / 1 ua) * (r / 1 ohm) );
end "*";

function "*" (r: weerstand; i : stroom)
    return spanning is
begin
    return i * r;
end "*";

```

figuur 2.27 Overloading van de '*' ten behoeve van de wet van ohm.

2.10 Package en package body.

In de vorige paragrafen zijn nieuwe types gedeclareerd en bestaande operatoren opnieuw gedeclareerd door middel van overloading, er is a.h.w. een nieuwe omgeving gecreëerd die later voor de verschillende beschrijvingen van entities en architectures kan worden gebruikt.

Gemeenschappelijke types, functies e.d. kunnen in een *package* worden ondergebracht. Een package bestaat uit een package en een package body (die in grote lijn overeenkomt met de definition module en de implementation module in modula-2). Figuur 2.28 geeft een voorbeeld van een package. De package beschrijft een drie niveau logica en definieert ook de logische operatoren, als voorbeeld is alleen de *and* opnieuw gedefinieerd. Indien een nieuw type wordt gedeclareerd worden ook impliciete functies gedeclareerd. O.a. wordt de '=' operator impliciet gedeclareerd als:

function "=" (a,b : drie_waardige_logica) **return** boolean;

Figuur 2.29 geeft een VHDL beschrijving die gebruik maakt van deze package.

Enkele opmerkingen bij de figuren 2.28 en 2.29:

- Door middel van een *use-clause* in figuur 2.29 wordt aangegeven welke package gebruikt moet worden. Deze use-clause bestaat uit:
 - 1 De library waarin zich de package bevindt. In dit geval de library *work*. Dit is een gereserveerd woord die de huidige werk directory aangeeft. In de volgende paragraaf zal worden aangegeven dat dit ook een andere directory kan zijn.
 - 2 Vervolgens moet de package worden aangegeven binnen deze library.
 - 3 Tenslotte kan worden opgegeven welke objecten uit deze package worden gebruikt. Als alle objecten nodig zijn wordt dit aangegeven met het gereserveerde woord *.ALL*. Dit wordt beschreven door: *use <library>.<package>.<objecten>*. Meerdere objecten kunnen worden opgegeven door ze te scheiden met een komma.
 Wordt alleen het type *drie_waardige_logica* gebruikt dan is het volgende voldoende: *use work.drie_nivo_logic.drie_waardige_logica*;
 Is ook de *and* nodig dan kan het volgende:
use work.drie_nivo_logic.drie_waardige_logica,"and";
 In figuur 2.29 is ook een beschrijving opgenomen met alleen de infix operator *and* opgenomen in de use-clause (ook nu zijn de dubbele quotes weer nodig omdat de operator een infix-operator is). Vervolgens is bij het declareren van de types een pad opgegeven. Soms moet dit wel worden gedaan omdat het gebruik van meerdere packages tegenstrijdigheden kan opleveren, bijvoorbeeld het gebruik van dezelfde naam voor een type. De use-clause geeft dus een pad naar de objecten die nodig zijn (zie figuur 2.30).
 Ook kan de package alleen zichtbaar worden gebruikt door alleen dit op te geven in de use-clause, dus: *use work.drie_nivo_logic*;
 Indien bijvoorbeeld het type *drie_waardige_logica* nodig is uit deze package wordt vanaf deze package het pad opgegeven, dus: *signal y : drie_nivo_logica.drie_waardige_logica*;
- De use-clause staat meestal voor de entity en hoeft dan niet meer herhaald te worden voor de architecture. Indien er in de entity-beschrijving geen gebruik

gemaakt wordt van een package dan mag de use-clause ook voor de architecture worden geplaatst. Dit laatste is zelfs te prefereren immers in een andere architecture-beschrijving behorend bij dezelfde entity wordt deze package misschien niet gebruikt!

- In de package zijn de functies gegeven waarvan anderen gebruik maken. Hoe die functies geïmplementeerd zijn is niet van belang. Het is dus belangrijk dat duidelijk als commentaar in de package wordt vermeld wat een functie of procedure doet. In de package body mogen nog hulp functies, procedures, types e.d. voorkomen welke niet zichtbaar zijn buiten deze package body!
- De package fungeert dus net als een entity als een interface met zijn omgeving. Hieruit volgt dan ook dat een package body niet noodzakelijkerwijs voor een entity, die gebruikt maakt van de daarbij behorende package, geanalyseerd hoeft te worden.
- Component-declaraties zijn meestal in de architecture opgenomen zoals de component-declaratie *nor_gate_2* in figuur 2.21, echter dergelijke declaraties mogen eveneens in een package zijn opgenomen welke door middel van een use clause zichtbaar worden gemaakt voor de desbetreffende architecture.

```
package drie_nivo_logica is
  type drie_waardige_logica is ('0','1','Z');
  type drie_waardige_vector is array (natural range <>) of drie_waardige_logica;
  -- de logische functie 'and'. Een uitgangssignaal is
  -- alleen '0' indien een of allebei ingangen '0' zijn.
  function "and" (a,b : drie_waardige_logica)
    return drie_waardige_logica;
  function "and" (a,b : drie_waardige_vector)
    return drie_waardige_vector;
end drie_nivo_logica;

package body drie_nivo_logica is
  function "and" (a,b : drie_waardige_logica)
    return drie_waardige_logica is
  begin
    if a='0' or b='0'
      then return '0';
    else return '1';
    end if;
  end "and";
  function "and" (a,b : drie_waardige_vector)
    return drie_waardige_vector is
    alias a_intern : drie_waardige_vector (1 to a'length) is a(a'range);
    alias b_intern : drie_waardige_vector (1 to b'length) is b(b'range);
    variable res : drie_waardige_vector (1 to a'length);
  begin
    for i in a_intern'range loop
      res(i) := a_intern(i) and b_intern(i);
    end loop;
    return res;
  end "and";
end drie_nivo_logica;
```

figuur 2.28 Package voor drie_nivo_logica.

```

use work.drie_nivo_logica.all;
entity test_all is
  port ( a : drie_waardige_vector (2 to 4);
        b : drie_waardige_vector (5 downto 3);
        c : out drie_waardige_vector (1 to 3));
end test_all;

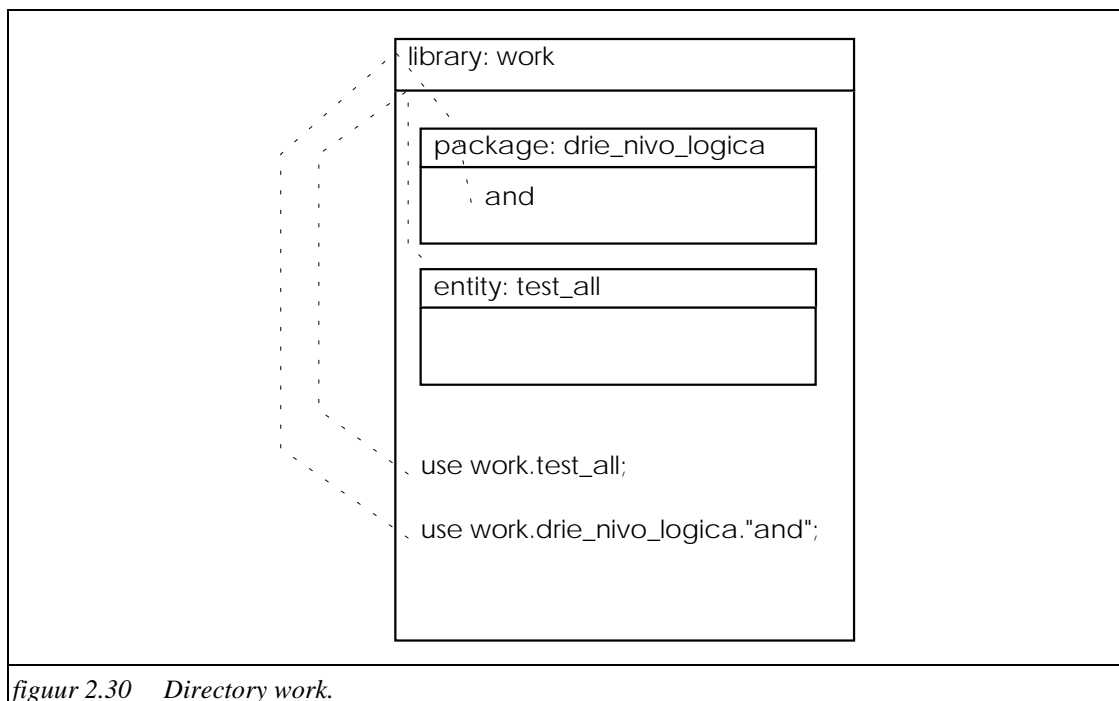
architecture and_test of test_all is
begin
  c <= a and b;
end and_test;

-----
use work.drie_nivo_logica."and";
entity test_select is
  port ( a : work.drie_nivo_logica.drie_waardige_vector (2 to 4);
        b : work.drie_nivo_logica.drie_waardige_vector (5 downto 3);
        c : out work.drie_nivo_logica.drie_waardige_vector (1 to 3));
end test_select;

architecture and_test of test_select is
begin
  c <= a and b;
end and_test;

```

figuur 2.29 Het gebruik van een package.



figuur 2.30 Directory work.

Zoals al eerder is opgemerkt is het type bit te beperkt om hardware mee te beschrijven. In het verleden zijn er dan ook vele packages ontwikkeld die uitgaan van een andere logische types. Zo werd in het begin vaak gebruik van een vier-waardige-logica gebruikt, bijvoorbeeld:

```

type vlbit is ('X','0','1','Z');      -- VIEWLogic
type qsim_state is ('X','0','1','Z'); -- Mentor Graphics

```

De 'Z' wordt gebruikt om de tri-state te modelleren en de 'X' voor onbekend en/of don't care. Daarom staat de 'X' ook als eerste genoemd, immers bij een impliciete initialisatie is de waarde van een signaal gelijk aan het eerste element, en dus onbekend!

De 'X' heeft in deze vier-niveau-logica geen unieke betekenis.

Er is zelfs een 46-niveau-logica¹² beschikbaar die geschikt is voor het modelleren van vele technologieën, zoals TTL, cmos, nmos etc.

Een werkgroep van de IEEE heeft de package `std_logic_1164` gedefinieerd die uitgaat van een negen niveau logica, hierin is o.a. een apart element voor don't care opgenomen en zijn ook *zwakke signalen* gedefinieerd. Dergelijke zwakke signalen komen bijvoorbeeld voor bij een wired-or. Alle CAE systemen ondersteunen deze negen waardige niveau logica.

2.10.1 Component declaraties in een package.

In een structuurbeschrijving staan meestal de te instantieren componenten in het declaratie gedeelte van de desbetreffende architecture. Het is ook mogelijk om deze component declaraties in een package te beschrijven en deze vervolgens door middel van een *use clause* te 'importeren'.

2.10.2 Deferred constant declaration.

In een package kunnen ook constanten worden gedeclareerd. Ook kan een volledige declaratie van een constante worden uitgesteld tot in de package body.

Het volgende is dus toegestaan:

```
package deferred_constant is
    constant test : bit; -- incomplete constant declaration !
end deferred_constant;

package body deferred_constant is
    constant test : bit := '1'; -- complete constant declaration !
end deferred_constant;
```

2.11 Libraries.

In de vorige paragraaf is al aangegeven dat gemeenschappelijke types, functies e.d. ondergebracht kunnen worden in een package. Vaak zullen deze gemeenschappelijke objecten voor meerdere ontwerpers beschikbaar moeten zijn en worden daarom in een aparte library ondergebracht. Hetzelfde geldt voor gemeenschappelijke entiteiten, ook deze zijn meestal ondergebracht in een aparte library. Ongemerkt is er al een library gebruikt namelijk de default library *work* (in figuur 2.29). Dit kan ook in de beschrijving duidelijk worden door de **library** *work* expliciet op te nemen. Voor elke VHDL beschrijving geldt de volgende impliciete context clause:

library *std*, *work*; **use** *std.standard.all*;

Waarbij in de library *std* ook nog een package *textio* aanwezig is.

```
library work;
use work.drie_nivo_logica.all;
entity test_all is
    port .....
    .....
```

figuur 2.31 Default library *work* expliciet opgegeven.

In VHDL worden libraries aangegeven met een logische naam. De werkelijke plaats van libraries in een computer systeem is niet van belang voor de VHDL gebruiker, uiteraard wel voor de systeembeheerder. Het voordeel hiervan is dat de beschrijving

¹² Coelho, D.R., The VHDL Handbook, Kluwer Academic Publishers, 1989

onafhankelijk kan blijven van het operating systeem. In de VHDL omgeving is dan ook meestal een initialisatie file aanwezig die de logische naam koppelt met de werkelijke plaats in het systeem.

Veronderstel dat package drie_nivo_logica (figuur 2.28) zich niet in de huidige werkdirectory bevindt (library work) maar in LIBRARY BASES_LOGICA. Dan komt de beschrijving van figuur 2.29 er uit te zien als gegeven in figuur 2.32, waarin de wijzigingen onderstreept zijn. Het plaatje in figuur 2.33 verduidelijkt dit nog eens.

```

library bases_logica;
use bases_logica.drie_nivo_logica.all;
entity test_all is
  port ( a : drie_waardige_vector (2 to 4);
        b : drie_waardige_vector (5 downto 3);
        c : out drie_waardige_vector (1 to 3));
end test_all;

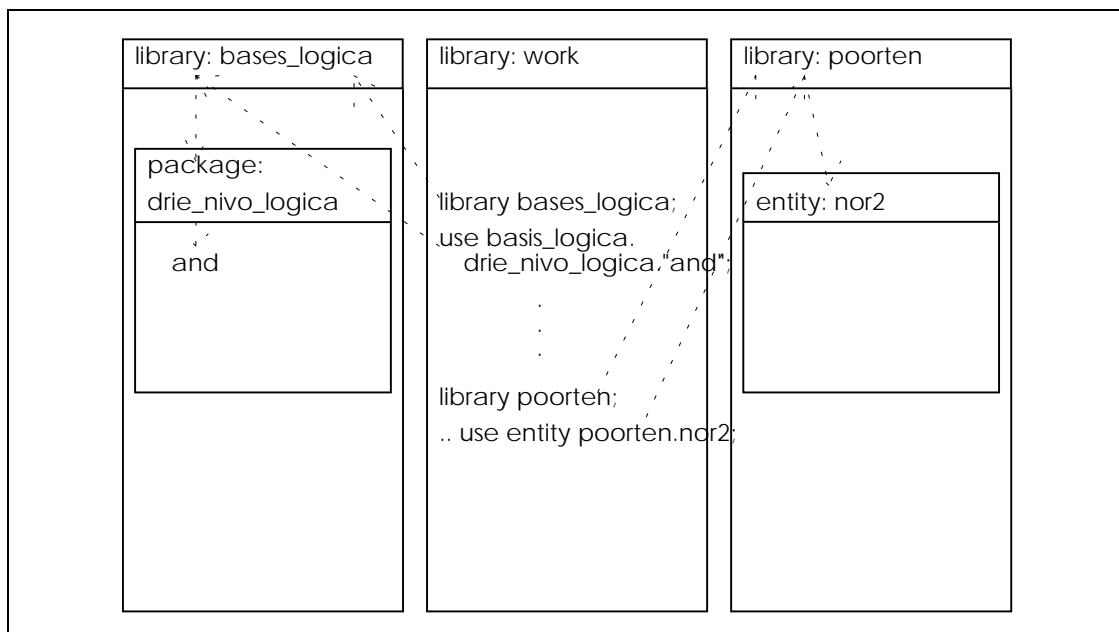
architecture and_test of test_all is
begin
  c <= a and b;
end and_test;

-----
library bases_logica;
use bases_logica.drie_nivo_logica."and";
entity test_select is
  port ( a : bases_logica.drie_nivo_logica.drie_waardige_vector (2 to 4);
        b : bases_logica.drie_nivo_logica.drie_waardige_vector (5 downto 3);
        c : out bases_logica.drie_nivo_logica.drie_waardige_vector (1 to 3));
end test_select;

architecture and_test of test_select is
begin
  c <= a and b;
end and_test;

```

figuur 2.32 Package drie_nivo_logica bevindt zich in library bases_logica.



figuur 2.33 Schematische weergave van de figuur 2.32.

Dat entiteiten zich ook in andere libraries kunnen bevinden dan de work library is eveneens aangegeven in figuur 2.33. In dit figuur is ook nog een library poorten opgenomen, met daarin de entiteit van de nor gate met 2 ingangen uit figuur 2.20 met de daarbij behorende architectuur beschrijvingen. De architecture BIJNA_STRUKTUUR gegeven in figuur 2.21 verandert daardoor iets omdat entity NOR2 zich niet meer in de

default directory bevindt, daarom is expliciet een library clause noodzakelijk. Deze library clause mag direct voor de architecture staan en hoeft niet voor de entity SR_LATCH te staan omdat er in de entity-beschrijving nog geen gebruik van deze library wordt gemaakt. Figuur 2.34 geeft door middel van een onderstreping de wijzigingen aan ten opzichte van figuur 2.21.

```

library poorten
architecture bijna_struktuur of sr_latch is
-- architectural description which uses a
-- process description for each nor-gate.
SIGNAL qi      : bit;
SIGNAL qi_not  : bit := '1';
COMPONENT nor_gate_2
  generic (delay : time);
  port (in1,in2  : in bit;
        outp    : out bit);
END COMPONENT;
for nor_g1:nor_gate_2 use entity poorten.nor2(gedrag);
for nor_g2:nor_gate_2 use entity poorten.nor2(struktuur);
begin
  nor_g1:nor_gate_2
    generic map (delay)
    port map (r,qi_not,qi);
  nor_g2:nor_gate_2
    generic map (delay)
    port map (s,qi,qi_not);
  q <= qi;
  q_not <= qi_not;
end bijna_struktuur;

```

figuur 2.34 Entity 'nor2' bevindt zich in library poorten.

2.12 Samenvatting.

Een beschrijving van hardware bestaat uit een *entity* en één of meerdere *architectures*. De entity legt de communicatie met de omgeving vast, ook kan hierin een generic worden opgenomen. In de entity-beschrijving wordt commentaar opgenomen *wat* de functie van de beschreven hardware is.

De architecture legt het gedrag vast van hardware. Als commentaar wordt globaal aangegeven *hoe* de gewenste functie is vervuld.

VHDL is gebaseerd op communicerende processen. Deze processen zijn impliciet of expliciet aanwezig. De volgorde waarin impliciete of expliciete processen zijn beschreven is niet belangrijk, de volgorde van de statements binnen een expliciete procesbeschrijving wel.

VHDL heeft een drietal objecten: signalen, variabelen en constanten. Communicatie tussen de processen kan alleen plaats vinden door middel van signalen. Alleen binnen een procesbeschrijving, functies en procedures mogen variabelen worden gebruikt.

Entiteiten kunnen als onderdeel van een andere architecture-beschrijving worden gebruikt door componenten te declareren en dit vervolgens door middel van een configuratie-specificatie te koppelen aan de entiteit.

Formele en actuele parameters worden gekoppeld door een positionele koppeling - een actuele parameter staat op de plaats van de formele parameter waarmee het gekoppeld moet worden - of expliciet door de formele en actuele parameters te koppelen m.b.v. de '=>'. Een combinatie is mogelijk, mits eerst de positionele koppeling is aangegeven. Variabelen en signalen worden default geïnitieerd met de meest linkse waarde van het desbetreffende type.

Functies leveren slechts één resultaat op, aan een variabele of signaal. Procedures kunnen meerdere resultaten opleveren maar default alleen maar aan variabelen (zie ook hoofdstuk 3).

In een package en package body kunnen types, functies, procedures, componenten e.d. worden gedeclareerd waarvan door middel van een 'use-clause' gebruik gemaakt kan worden.

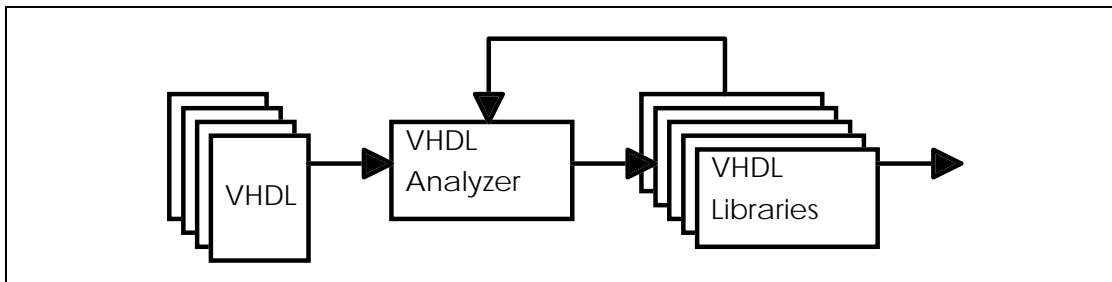
Door middel van de mode 'in', 'out' en 'inout' wordt aangegeven of er resp. sprake is van een in, uit of bidirectionele parameter is. Tevens zijn er nog de modes 'buffer' en 'linkage'. De laatste mode wordt zelden gebruikt.

3. Verdieping in onderwerpen van VHDL.

Een aantal aspecten van VHDL die niet in het inleidende hoofdstuk over VHDL aan de orde zijn gekomen, maar wel van belang zijn voor het ontwerpen en het modelleren van hardware worden in dit hoofdstuk behandeld. De meeste onderwerpen zoals die behandeld worden in dit hoofdstuk kunnen afzonderlijk worden bestudeerd.

3.1 Analyse, 'Elaboration' en Simulatie.

3.1.1 Analyse volgorde



figuur 3.1 Analyse van een VHDL beschrijving.

Tot nu toe zijn vele VHDL beschrijvingen en simulatieresultaten gegeven. Dit werd bereikt door eerst de beschrijvingen te analyseren waarna de resultaten door een simulator ingelezen werden. De commando's voor het analyseren en simuleren behoren niet tot de VHDL standaard en worden daarom ook niet behandeld. Deze paragraaf behandelt de volgorde van de analyse van de design units.

De design units van VHDL zijn:

1. Entity declaration.
2. Architecture body.
3. Package declaration.
4. Package body.
5. Configuration declaration (wordt behandeld op pagina 49).

Voor de analyse geldt dat elke design unit in één file moet staan en dat er in een file meerdere design units mogen staan. Impliciet staat voor elke design unit de volgende library en use clause:

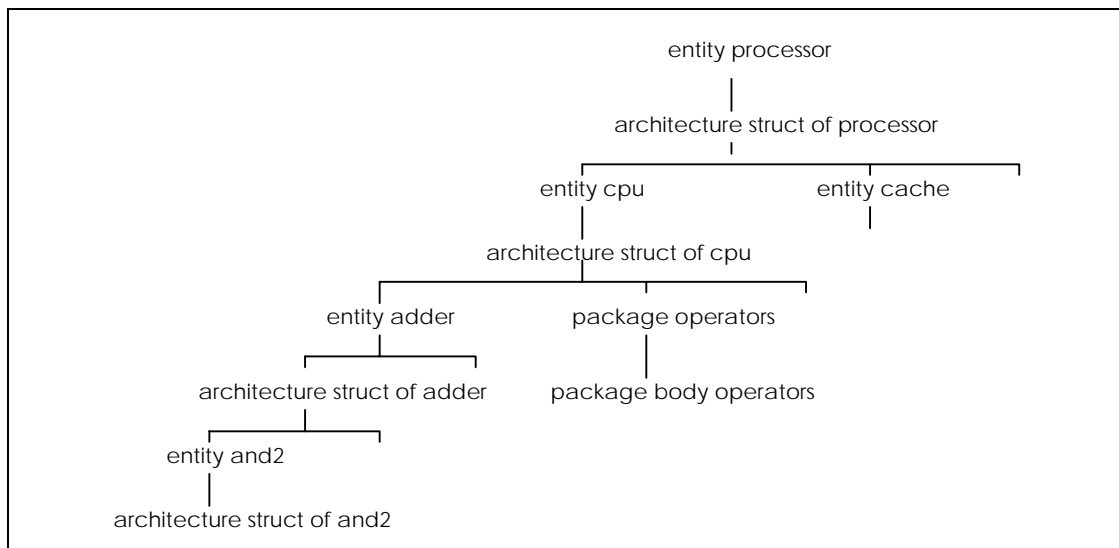
```

library work,std;
use std.standard.all;
  
```

Voordat een design unit wordt geanalyseerd wordt eerst de al eerdere geanalyseerde package standard ingelezen en eventueel ook de expliciet opgegeven packages. Indien een design geen fouten bevat wordt deze opgeslagen in de library structuur, meestal in library work. Indien bijvoorbeeld een architecture geanalyseerd moet worden wordt voordat met de analyse begonnen wordt uit de library de gegevens van de daarbij behorende entity ingelezen. Indien de entity ontbreekt volgt uiteraard een foutmelding. Hieruit volgt direct dat er blijkbaar een volgorde is waarin design units geanalyseerd moeten worden. Figuur 3.1 geeft schematisch het analyse proces weer. Analyse wordt soms ook wel compilatie genoemd, echter bedenk dat het resultaat van de analyse geen executable is! In de volgende paragraaf wordt hierop nader ingegaan.

In figuur 3.2 is een gedeelte aangegeven van de entiteiten en packages waarvan de processor gebruik maakt. Veronderstel dat alles in de correcte volgorde geanalyseerd is en dat de gedragsbeschrijving van de NOR poort niet correct blijkt te zijn. Deze

gedragsbeschrijving wordt aangepast en opnieuw geanalyseerd, maar dan moet ook de ADDER opnieuw geanalyseerd worden, zodat vervolgens ook de CPU opnieuw geanalyseerd moet worden, etc. Dit is uiteraard niet wenselijk en ook niet nodig!



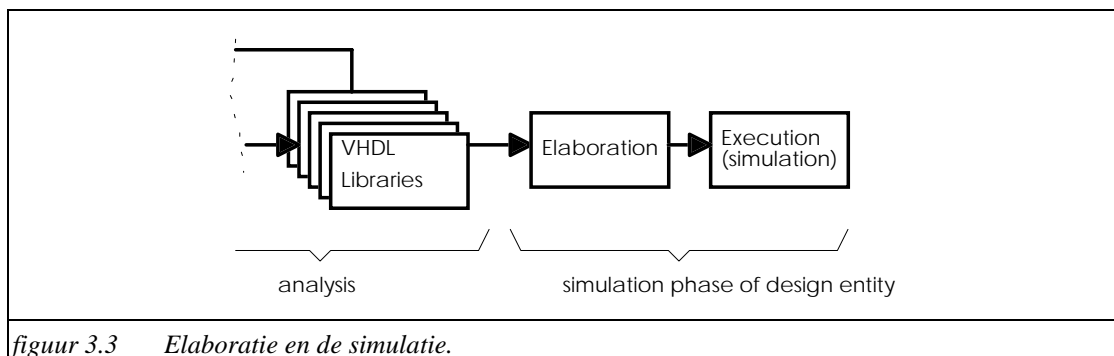
figuur3.2 *Entities, architectures, packages en package bodies van een processor.*

De entity en de package vormen een interface naar respectievelijk een architecture en de package body. Indien alleen een architecture opnieuw wordt geanalyseerd blijft de interface, in dit geval de entity, onveranderd en hoeven de architectures die deze entity gebruiken niet opnieuw geanalyseerd te worden. Ook het analyseren van alleen de package body laat de interface, de package, ongemoeid.

Nu zijn veelal de entity en de verschillende architectures behorende bij deze entity in één file ondergebracht. Indien de architecture wordt gewijzigd en vervolgens deze file opnieuw geanalyseerd wordt, wordt ook de entity opnieuw geanalyseerd! Dit kan worden voorkomen door de entity en de architectures behorende bij deze entity onder te brengen in verschillende files. Om dezelfde reden kan ook de package en de package body worden ondergebracht in verschillende files. Om te voorkomen dat men het overzicht kwijtraakt wordt voor de file naam vaak de naam van de design unit gebruikt, bijvoorbeeld:

- entity SR_LATCH in de file
SR_LATCH.VHD
- architecture GEDRAG van de entity SR_LATCH in de file
SR_LATCH-GEDRAG.VHD.

3.1.2 Elaboratie en simulatie.



figuur 3.3 Elaboratie en de simulatie.

Na de analyse bevinden zich de design units in de VHDL libraries. Echter de simulatie van een design entity kan niet direct plaats vinden. Eerst moet er nog een ‘elaboration’ aan vooraf gaan. In de elaboration fase wordt een simuleerbaar model opgebouwd waarbij o.a.:

1. De componenten geïnstantieerd worden.
2. De globale constanten (de generics) worden ingevuld.
3. De signalen worden gecreëerd.
4. Initialisatie
5. etc.

Nadat dit model is opgebouwd kan het simuleren plaats vinden. De elaboration fase vindt in veel VHDL omgevingen impliciet plaats, namelijk direct nadat is aangegeven welke design entity gesimuleerd moet gaan worden. Alleen tijdens de elaboration fase wordt dus ruimte gecreëerd voor de signalen, dit betekent dat er tijdens het simuleren geen signalen gecreëerd kunnen worden. Vandaar ook dat in een subprogram (functies en procedures) geen signalen gedeclareerd mogen worden.

3.2 Configuratie-declaratie.

Één entity-beschrijving kan meerdere architectures hebben. Als in een architecture een component-declaratie staat wordt door middel van de aanwezige configuratie-specificatie aangegeven welke entity hieraan is gekoppeld, zoals dat bijvoorbeeld gedaan is in figuur 2.21:

```
for nor_g1:nor_gate_2 use entity work.nor2(gedrag);
for nor_g2:nor_gate_2 use entity work.nor2(struktuur);
```

Bij het ontbreken van een dergelijke configuratie-specificatie wordt van een entity de laatst geanalyseerde architecture gebruikt. Voorwaarde hierbij is wel dat de naam en de interface-definitie van de component-declaratie en de entity gelijk moeten zijn en dat de entity zich in de work directory bevindt. Ontbreekt een configuratie-specificatie en er is geen entity met dezelfde naam aanwezig dan zal er tijdens het simuleren een foutmelding worden gegeven.

De configuratie-specificaties die in een architecture-beschrijving staan kunnen ook in een aparte *configuration-declaration* worden geplaatst. Het voordeel hiervan is dat een architecture niet opnieuw geanalyseerd hoeft te worden indien één van de gebruikte entities is veranderd, alleen de configuratie-declaratie moet opnieuw geanalyseerd worden. Voor dezelfde design entity kunnen meerdere configuraties aanwezig zijn.

Voor de entity SR_LATCH waarvan de architecture BIJNA_STRUKTUUR gegeven is in figuur 2.21 ziet de configuration er als volgt uit (N.B. de twee expliciet gegeven configuratie-specificaties, zoals hierboven gegeven, mogen¹³ dan niet meer voor in de architecture):

```
configuration conf_1 of sr_latch is
  for bijna_struktuur -- is architecture van sr_latch
    for nor_g1:nor_gate_2 -- instantiatie label met component
      use entity work.nor2(gedrag);
    end for;
    for nor_g2:nor_gate_2
      use entity work.nor2(struktuur);
    end for;
  end for;
end conf_1;
```

Zoals in het vorig voorbeeld al duidelijk is geworden zijn de labels belangrijk bij configuraties. Bedenk dat mede daarom voor aan aantal taalconstructies een label een vereiste is, zoals bijvoorbeeld voor het generate en het block statement. Ook kunnen binnen configuraties weer andere configuraties worden gebruikt.

Om een indruk te geven van de mogelijkheden zijn in figuur 3.4 een aantal mogelijkheden gegeven. Enkele opmerkingen bij dit figuur zijn:

- Figuur 3.4a zou een beschrijving van een buffer kunnen zijn.
- In figuur 3.4b wordt een component-declaratie gebruikt waarin een component is beschreven welke niet als entity voorkomt met dezelfde naam en interface-definitie in de library work.
- In figuur 3.4c wordt een configuratie gegeven voor entity b1 waarin is aangegeven dat voor component la1 entity a1 gebruikt moet worden.
- In figuur 3.4d is een component-declaratie gebruikt die gelijk is aan de entity b1.
- Figuur 3.4e beschrijft de configuratie van entity c1 waarin eerst wordt aangegeven dat entity b1 wordt gebruikt, en dat voor de component-declaratie in de architecture arch van b1 entity a1 moet worden gebruikt.
- Echter voor entity b1 was al een configuratie aanwezig. Het is ook mogelijk deze configuratie in een configuratie voor c1 te gebruiken (figuur 3.4f)
- De figuren 3.4f en 3.4g geven een voorbeeld waarin getoond wordt hoe het label van een block statement terug komt in de configuratie.
- Met het generate statement kan eenvoudig een regelmatige structuur worden beschreven. Maar het blijft mogelijk om aan elk geïntanceerde component een configuratie-specificatie toe te kennen. Figuur 3.4i geeft aan hoe een configuratie-declaratie daarvoor kan worden gebruikt. Indien de configuratie-specificatie voor een aantal geïntanceerde componenten van toepassing is kan dit door middel van een range achter het label van het generate statement worden aangegeven in de configuratie-declaratie (bijvoorbeeld: **for** width_buffers(1 to 3)) of indien het voor elke geïntanceerde component van toepassing is kan een range geheel ontbreken (bijvoorbeeld: **for** width_buffers).

¹³ In de IEEE Std. 1076-1993 (VHDL'93) mag een configuratie-specificatie wordt overruled door een andere configuratie-specificatie.

<pre> entity a1 is port (a: in bit; b : out bit); end a1; architecture arch of a1 is begin b <= a; end arch; </pre> <p>a)</p>	<pre> entity b1 is port (a: bit; b : out bit); end b1; architecture arch of b1 is component la1 port (a: in bit; b : out bit); end component; begin z : la1 port map(a,b); end arch; </pre> <p>b)</p>
<pre> configuration confb of b1 is for arch for z : la1 use entity work.a1(arch); end for; end for; end confb; </pre> <p>c)</p>	<pre> entity c1 is port(a: bit; b : out bit); end c1; architecture arch of c1 is component b1 port (a: in bit; b : out bit); end component; begin z : b1 port map(a,b); end arch; </pre> <p>d)</p>
<pre> configuration confc1 of c1 is for arch for z : b1 use entity work.b1(arch); for arch for z : la1 use entity work.a1(arch); end for; end for; end for; end confc1; </pre> <p>e)</p>	<pre> configuration confc2 of c1 is for arch for z : b1 use configuration work.confb; end for; end for; end confc2; </pre> <p>f)</p>

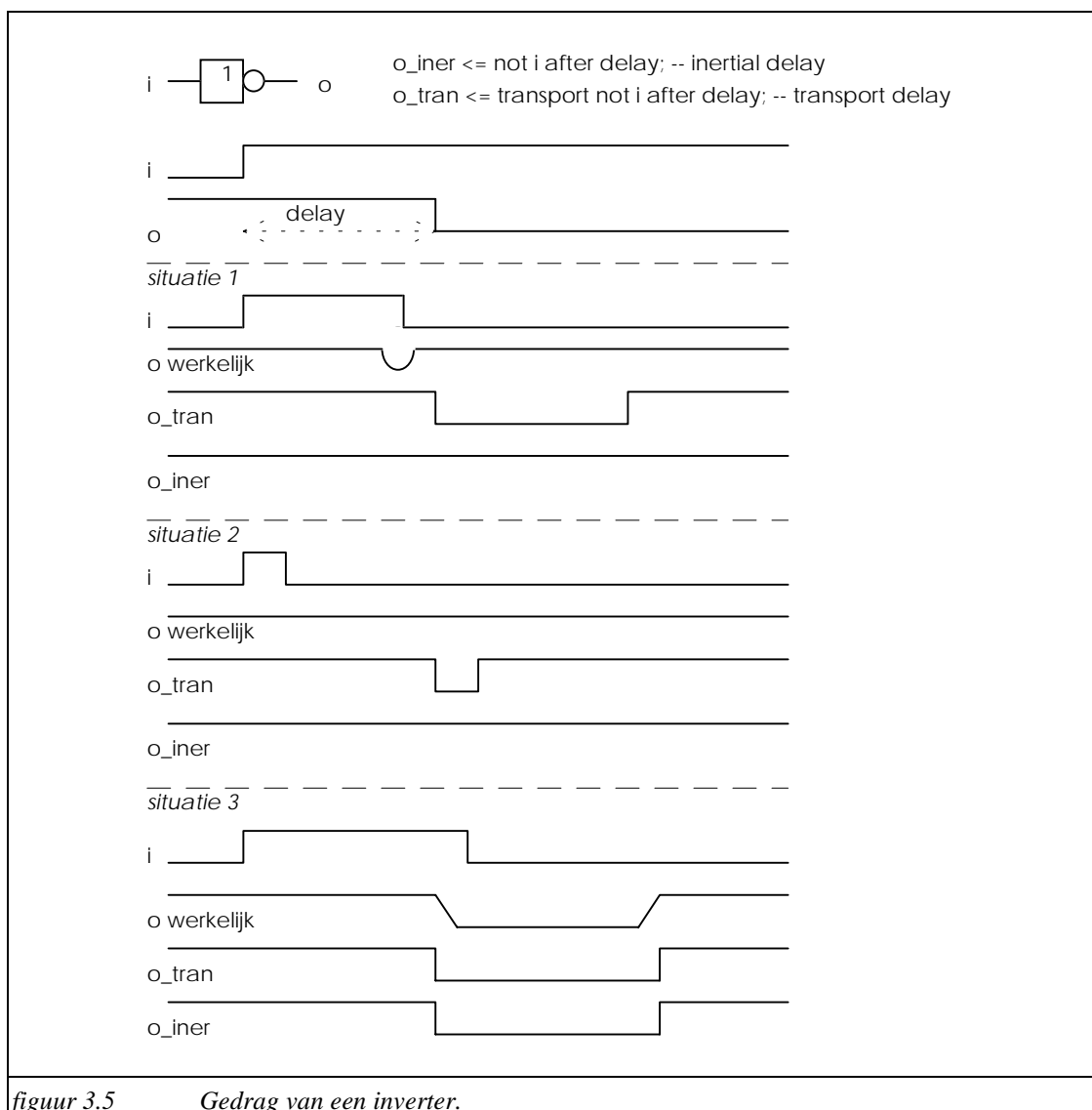
<pre> entity d1 is port(a: bit; b : out bit); end d1; architecture arch of d1 is begin demo:block component b1 port (a: in bit; b : out bit); end component; begin z : b1 port map(a,b); end block; end arch; </pre> <p>g)</p>	<pre> configuration confd of d1 is for arch for demo for z : b1 use configuration work.confb; end for; end for; end confd; </pre> <p>h)</p>
<pre> entity n_output_buffers is generic (width : natural :=8); port (inp : in bit_vector(width downto 1); outp : out bit_vector(width downto 1)); end n_output_buffers; architecture struktuur of n_output_buffers is component output_buffer port (input : in bit; output: out bit); end component; begin width_buffers:for i in width downto 1 generate outp_buf:output_buffer port map(inp(i),outp(i)); end generate width_buffers; end struktuur; configuration buffer1 of n_output_buffers is for struktuur for width_buffers(1) for outp_buf : output_buffer use entity work.output_buffer; end for; end for; end buffer1; </pre> <p>i)</p>	

figuur 3.4 Voorbeelden van configuratie-declaraties.

3.3 Modelleren van vertraging.

Bij geen enkel component heeft een verandering van een ingangssignaal direct een verandering van het uitgangssignaal tot gevolg, tussen beide zit altijd een zekere vertragingstijd. Van een inverter waarvan het ingangssignaal verandert van een '0' naar een '1' zal het uitgangssignaal iets later veranderen van '1' naar '0'. Maar wat gebeurt er als vóór de verandering van het uitgangssignaal het ingangssignaal weer naar '0' gaat? Figuur 3.5 schets globaal het verloop van het uitgangssignaal bij de verschillende ingangssignalen. Het modelleren van dergelijke realistische situaties is zonder gekunstelde beschrijvingen niet mogelijk in VHDL. VHDL ondersteunt een drietal geïdealiseerde modellen om vertraging te modelleren:

- delta-delay
- transport-delay
- inertial-delay



figuur 3.5 Gedrag van een inverter.

Delta-delay kan worden opgevat als een bijzondere vorm van zowel de transport- als inertial-delay, namelijk de situatie waarbij de opgegeven vertragingstijd 0 ns is. Dit betekent dat het signaal één iteratie wordt vertraagd. In deze paragraaf worden dan ook alleen de geïdealiseerde vertragingmodellen *transport-delay* en *inertial-delay*

behandeld. Globaal is het verschil tussen beide modellen dat een verandering, hoe kort ook, altijd wordt doorgegeven bij een transport-delay en dat voor inertial-delay moet gelden dat de verandering tenminste gedurende de aangegeven vertragingstijd stand moet houden. In figuur 3.5 zijn deze geïdealiseerde vertragingmodellen afgebeeld met een schets van de werkelijke vertraging voor de inverter.

Om het verschil aan te geven tussen transport en inertial-delay wordt gebruik gemaakt van het gereserveerde woord **transport**. Een gereserveerd woord om inertial-delay aan te geven is er niet, het default vertragingmodel is dan ook de inertial-delay.

In figuur 2.6 is aangegeven dat een signaal niet alleen een huidige waarde heeft maar ook toekomstige waarden met de daarbij behorende tijdstippen. Deze lijst wordt dan een *driver* voor een signaal genoemd. Veronderstel dat een signaal *y* van het type character een driver heeft met een aantal toekomstige transitie's, bijvoorbeeld de driver van signaal *y* is: 'a', 'c' @ 5 ns, 'b' @ 10 ns, 'c' @ 20 ns, 'e' @ 40 ns, 'f' @ 60 ns. Wat gebeurt er nu als een van de volgende twee toekenningen wordt gedaan aan signaal *y*:

y <= **transport** 'c' **after** 30 ns; -- transport-delay;

y <= 'c' **after** 30 ns; - inertial-delay;

De procedure voor het bepalen van de driver op basis van een transport-delay is:

- 1 Verwijder van de aanwezige driver alle transitie's die groter of gelijk zijn aan de nieuw toe te voegen vertragingstijd. De nieuw toe te voegen vertragingstijd is 30 ns, dus tijdelijk wordt de driver van signaal *y* gelijk aan:
'a', 'c' @ 5 ns, 'b' @ 10 ns, 'c' @ 20 ns
- 2 Voeg de nieuwe transitie toe aan de ontstane driver:
'a', 'c' @ 5 ns, 'b' @ 10 ns, 'c' @ 20 ns, 'c' @ 30 ns

De procedure voor het bepalen van de driver op basis van een inertial-delay is:

- 1 Verwijder van de aanwezige driver alle transitie's die groter of gelijk zijn aan de nieuw toe te voegen vertragingstijd. De nieuw toe te voegen vertragingstijd is 30 ns, dus tijdelijk wordt de driver van signaal *y* gelijk aan:
'a', 'c' @ 5 ns, 'b' @ 10 ns, 'c' @ 20 ns
- 2 Voeg de nieuwe transitie toe aan de ontstane driver:
'a', 'c' @ 5 ns, 'b' @ 10 ns, 'c' @ 20 ns, 'c' @ 30 ns
- 3 Markeer de toegevoegde transitie (*):
'a', 'c' @ 5 ns, 'b' @ 10 ns, 'c' @ 20 ns, 'c' @ 30 ns*
- 4 Markeer (1) de huidige waarde en (2) alle opeenvolgende waarden voorafgaand aan de laatst toegevoegde waarde die hieraan gelijk zijn.
'a' *, 'c' @ 5 ns, 'b' @ 10 ns, 'c' @ 20 ns *, 'c' @ 30 ns*
- 5 Verwijder de niet gemarkeerde transitie's:
'a', 'c' @ 20 ns, 'c' @ 30 ns

Veronderstel dat voor een signaal *o* de volgende driver aanwezig is:

o : 'x', 'z' @ 5 ns, 'y' @ 30 ns

Indien dan in een procesbeschrijving de volgende statements zijn gegeven in de gegeven volgorde:

o <= **transport** 'a' **after** 10 ns;

o <= **transport** 'b' **after** 20 ns;

o <= **transport** 'c' **after** 30 ns;

o <= **transport** 'e' **after** 50 ns;

dan wordt de volgende driver samengesteld:

o : 'x', 'z' @ 5 ns, 'a' @ 10 ns, 'b' @ 20 ns, 'c' @ 30 ns, 'e' @ 50 ns (1)

Deze vier opeenvolgende statements zijn equivalent met het statement:

o <= **transport** 'a' **after** 10 ns, 'b' **after** 20 ns, 'c' **after** 30 ns, 'e' **after** 50 ns;

Zou er geen sprake van transport maar van inertial-delay zijn, dus:

o <= 'a' **after** 10 ns;

o <= 'b' **after** 20 ns;

o <= 'c' **after** 30 ns;

o <= 'e' **after** 50 ns;

dan wordt de driver voor signaal o gelijk aan: 'x', 'e' @ 50 ns! De eerste statement zal de transitie 'y' @ 30 ns verwijderen en de transitie 'a' @ 10 ns toevoegen en ten gevolge van de markering de transitie 'z' @ 5 ns verwijderen, etc.

Voor de statement o <= 'a' after 10 ns, 'b' after 20 ns, 'c' after 30 ns, 'e' after 50 ns; geldt alleen voor de eerste vertraging het inertial vertragingsmodel voor de overige vertragingen geldt het transport vertragingsmodel.

Dus uitgaande van de driver voor signaal o : 'x', 'z' @ 5 ns, 'y' @ 30 ns geeft dit de volgende driver:

'x', 'a' @ 10 ns, 'b' @ 20 ns, 'c' @ 30 ns, 'e' @ 50 ns.

Merk op dat het resultaat van de driver voor signaal o afwijkt van alleen maar transport-delays, zie (1)!

3.4 Procedures en functies.

Functies en procedures worden subprograms genoemd en worden o.a. gebruikt om een beschrijving beter leesbaar te maken en om gemeenschappelijke stukken programma tekst mee te beschrijven.

In procedures en functies mogen ook de attributen worden gebruikt, echter het is niet toegestaan om die attributen te gebruiken welke een signaal zijn.

De verschillen tussen een functie en een procedure zijn in de volgende tabel gegeven en zullen worden toegelicht.

	<u>functie</u>	<u>procedure</u>
1	kan slechts een resultaat opleveren	kan willekeurig aantal resultaten opleveren, ook nul.
2	dezelfde functie kan zowel aan een variabele als aan een signaal een toekenning doen	dezelfde procedure kan alleen aan een signaal of alleen aan een variabele een toekenning doen
3	expressie	statement
4	mag geen wait statement bevatten	mag ook meerdere wait statements bevatten

Stel er is een procedure demo met de volgende declaratie:

procedure demo(a,b,c,d : bit, e : **out** bit);

In deze procedure zijn a, b, c en d van mode **in**, dit is de default mode. Echter er is nog een default aanwezig in deze beschrijving namelijk de default voor het object. Hiervoor geldt:

mode	default object
in	constant
out	variable
inout	variable

M.a.w. de procedure demo is identiek aan:

```
procedure demo(constant a,b,c,d : in bit, e : variable out bit);
```

Uiteraard kan ook expliciet het object worden opgegeven bijvoorbeeld:

```
procedure demo1(signal a,b,c,d : in bit, e : signal out bit);
```

In procedure demo1 zijn de objecten dus van het type signaal. Het object bepaald tevens het gebruik van een procedure:

formele object	toegestane actuele object
constant	expressie
variable	variable
signal	signal

Indien het formele object een signaal is wordt niet de waarde overgedragen maar een referentie naar het actuele signaal zelf.

Veronderstel dat de volgende procedures gedeclareerd zijn:

```
procedure x1 (a : in bit);
```

```
procedure x2 (variable a : in bit);
```

```
procedure x3 (signal a : in bit);
```

```
procedure x4 (constant a : in bit);
```

De procedures x1 en x4 zijn identiek.

Veronderstel tevens dat de volgende objecten gedeclareerd zijn:

```
signal sgn1, sgn2 : bit;
```

```
variable v1, v2 : bit;
```

Dan zijn de volgende procedure aanroepen correct:

```
x1(sgn1);           -- expressie, alleen de 'waarde' wordt overgedragen
```

```
x1(sgn1 or sgn2);   -- idem.
```

```
x1(v1);
```

```
x1(v1 or v2);
```

```
x2(v1);
```

```
x3(sgn1);
```

Onjuist zijn:

```
x2(v1 or v2)        -- een expressie, moet echter een variable zijn.
```

```
x3(v1)              -- actuele parameter is geen signaal
```

Het zal duidelijk zijn dat mode out nooit van het object type constant kan zijn.

3.4.1 Procedures en het gebruik van signalen.

```
procedure wire(a :bit; b : out bit);
```

```
begin
```

```
  b := a;
```

```
end wire;
```

In de procedure wire is al te zien dat de default mode van de formele parameter b een variabele is, immers als toekennings operator wordt de ':=' gebruikt. Indien de actuele parameter voor de formele parameter b een signaal is zal er tijdens de analyse een fout worden gegeven. Om deze procedure geschikt te maken voor een toekenning aan een signaal zou de procedure herschreven moeten worden naar:

```
procedure wire(a :bit; signal b : out bit);
begin
  b <= a;
end wire;
```

Echter de laatste procedure kan weer niet worden gebruikt als het actuele object een variabele is. Als nu beide declaraties worden gegeven kan dan niet door middel van overloading de juiste procedure worden bepaald? Voor overloading moet gelden dat de naam en de typering van de parameters voor de keuze van een subprogram gelijk moeten zijn. Deze zijn voor beide procedures gelijk en dus werkt overloading hier niet.

Voor een functie doen zich deze problemen niet voor, immers in een functie is het niet bekend of er een toekenning aan een variabele of aan een signaal wordt gedaan, dankzij de return statement.

3.4.2 Procedure met wait statements.

Een procedure mag een of meerdere wait statements bevatten. Een voorbeeld van een dergelijke procedure is:

```
procedure flipflop (data : in bit; clk : in bit; signal q : out bit) is
begin
  wait until clk='1';
  q <= data;
end flipflop;
```

Na het aanroepen van deze procedure wordt gewacht totdat het signaal clk '1' is geworden, althans dat was waarschijnlijk de bedoeling! Aangezien de formele parameter van het object een constante is wordt tijdens de aanroep de waarde op dat moment doorgegeven en een referentie naar het signaal. De procedure zal zijn executie dan ook niet vervolgen.

De gewenste beschrijving is:

```
procedure flipflop (data : in bit; signal clk : in bit; signal q : out bit) is
begin
  wait until clk='1';
  q <= data;
end flipflop;
```

Moet de formele parameter data ook niet van het object signaal zijn, of heeft dat geen invloed op het gedrag omdat dit signaal niet in een wait statement wordt gebruikt? Om aan te geven dat ook dit het gedrag kan beïnvloeden worden de volgende twee procedures gebruikt:

```
procedure delay1 (data : in bit; signal q : out bit) is
begin
  wait for 100 ns;
```

```

    q <= data;
end delay1;

procedure delay2 (signal data : in bit; signal q : out bit) is
begin
    wait for 100 ns;
    q <= data;
end delay2 ;

```

Na het aanroepen van de procedure delay1 wordt de waarde van dat moment doorgegeven aan het constante formele object data. Na 100 ns wordt dit dan toegekend aan het signaal q. Bij procedure delay2 wordt niet de waarde, maar een referentie naar het actuele signaal doorgegeven. Na 100 ns wordt dan de actuele waarde van het signaal doorgegeven aan het signaal q. Dus is er wel degelijk sprake van een verschillend gedrag.

3.4.3 Een functie mag geen wait statements bevatten.

Een functie is een expressie welke een waarde oplevert. De executie van een functie is gelijk aan die van alle andere operatoren. Het resultaat wordt direct bepaald, er mogen dus geen wait statements aanwezig zijn in een functie.

Een functie is meestal zodanig gedefinieerd dat het resultaat altijd hetzelfde is als de invoer hetzelfde is, zoals bijvoorbeeld voor de operator and.

In de package standard is één functie aanwezig die hieraan niet voldoet: de functie **now** (deze geeft het simulatietijdstip als resultaat).

3.4.4 Concurrent procedure call.

Figuur 3.6 geeft een procedure voor het maximum, dat het resultaat aan een signaal geeft, vergelijk deze beschrijving met figuur 2.17.

```

architecture max_procedure of maximum is
begin
    process(in1,in2)
        procedure find_maximum (input1,input2: in integer;
                               signal outp : out integer) is
            -- levert het maximum van input1 en input2
            begin
                if input1>input2
                    then outp <= input1;
                    else outp <= input2;
                end if;
            end find_maximum;
        begin
            find_maximum(in1,in2,max);
        end process;
    end max_procedure;

```

figuur 3.6 Procedure met expliciete toekenning aan een signaal.

In figuur 3.6 wordt de procedure nog steeds in een sequentiële beschrijving aangeroepen, dit wordt dan ook (*sequential*) *procedure call* genoemd. Het is ook mogelijk procedures, die alleen aan signalen een resultaat geven, voor een *concurrent procedure call* te gebruiken. Een concurrent procedure call bevindt zich niet in een procesbeschrijving maar is zelf een proces!

Veronderstel dat procedure demo de volgende vorm heeft:

```

procedure demo ( a,b : in bit, signal c,d : out bit);

```

De volgende beschrijving is dan toegestaan:

```
architecture test of concurrent_procedure_call is
  signal u,v,w,x,y,z : bit;
begin
  demo(u,v,w,x);
  z <= y;
end test;
```

en is equivalent met:

```
architecture test of concurrent_procedure_call is
  signal u,v,w,x,y,z : bit;
begin
  process
  begin
    demo(u,v,w,x);
    wait on u,v;
  end process;
  z <= y;
end test;
```

3.5 Signaal met meerdere sources.

```
process(s)
begin
  a<='0';
  if s='1' then a<='1'; end if;
end process;
```

figuur 3.7 *Signaal a heeft één source.*

```
proces_0:process(s)
begin
  a<='0';
end process;

proces_1:process(s)
begin
  a<='1';
end process;
```

figuur 3.8 *Twee sources voor signaal a.*

Ieder proces waarin een toekenning aan een signaal wordt gedaan creëert één driver voor dat signaal. Ook al zijn er meerdere toekenningen binnen een proces aan hetzelfde signaal, er wordt toch maar één driver gecreëerd. In de tot nu toe gegeven VHDL beschrijvingen had elk signaal maar één source. In figuur 3.7 is een proces gegeven dat, indien s='1' waar is, tweemaal een toekenning wordt gedaan aan het signaal A. Toch is er slechts sprake van één driver. Er is ook geen onduidelijkheid over de betekenis van dit proces dankzij het sequentiële karakter ervan: eerst wordt aan signaal A met een vertraging van een delta-delay de waarde '0' toegekend, maar als s='1' waar is vervalt deze toekenning en wordt met een vertraging van een delta-delay een '1' aan signaal A toegekend.

Figuur 3.8 geeft twee processen die beide een driver creëren voor signaal A. Als signaal s verandert moet signaal A van proces PROCES_0 na delta-delay de waarde '0'

krijgen terwijl het van het proces PROCES_1 na delta-delay de waarde '1' krijgt! De waarde van signaal A is dus onbepaald. VHDL zal dit ook niet accepteren!

Toch zijn er situaties waarin het wenselijk is dat één signaal meerdere sources¹⁴ heeft, bijvoorbeeld voor het beschrijven van een wired_or, of het beschrijven van een bidirectionele bus. M.b.v. een *resolved signal* is dit probleem op te lossen.

3.5.1 Resolved signal.

Een signaal met meerdere sources, zoals geschetst is in figuur 3.8, moet een *resolutiefunctie* hebben. Een resolutiefunctie verzamelt de waarden van deze sources van een signaal en bepaalt op basis hiervan een resulterende waarde voor het signaal. In figuur 3.8 zou dus de resolutiefunctie op basis van de toekenning van een '1' en een '0' aan signaal A een resulterende waarde moeten bepalen voor dit signaal.

Een resolutiefunctie is een normale functie, en krijgt pas een bijzondere betekenis in combinatie met een type. Veronderstel dat het de bedoeling is van de beschrijving in figuur 3.8 dat een wired_or gemodelleerd moet worden. Er moet dan een functie aanwezig zijn die dit beschrijft. De functie WIRED_OR, figuur 3.9, wordt hiervoor gebruikt. Door het declareren van een subtype kan worden aangegeven welke functie als resolutiefunctie wordt gebruikt. Het subtype WIRED_OR_BIT koppelt de functie WIRED_OR aan het type BIT (figuur 3.9). Ook nu erft het subtype WIRED_OR_BIT alle gedefinieerde operatoren van het type BIT. Ook mag een signaal van een dergelijk subtype weer worden toegekend aan signalen van het type waarvan het is afgeleid.

Beide processen in figuur 3.8 hebben een driver voor signaal A. Het signaal is van het type bit en wordt geconcateneerd tot een bit_vector. De ontstane bit_vector is invoer van de functie WIRED_OR en het functieresultaat is de resulterende waarde voor signaal A.

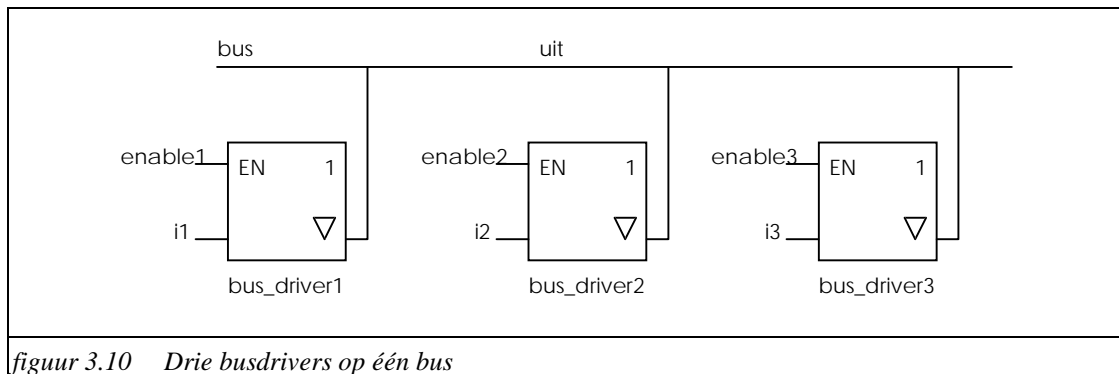
```
function wired_or (inp : bit_vector)
    return bit is
begin
    for i in inp'range loop
        if inp(i)='1' then return '1'; end if;
    end loop;
    return '0';
end wired_or;

-- subtype <type resolved signal > is <function> <type>
subtype wired_or_bit is wired_or bit;
signal a : wired_or_bit;
```

figuur 3.9 Modelleren van een wired-or.

¹⁴ Een source van een signaal kan zijn a) een driver (vanuit een proces) of b) een *out*, *inout*, *buffer* of *linkage* port van een component-instantiatie of van een block statement.

3.5.2 Modelleren van een bidirectionele bus.



figuur 3.10 Drie busdrivers op één bus

Figuur 3.10 toont een één bit brede bus met drie busdrivers. Uiteraard moet voor het modelleren van de hardware bus een *resolved signal* worden gebruikt immers er zijn drie mogelijke drivers voor deze bus. De busdrivers zijn alle voorzien van een tri-state uitgang met de logische niveaus '0', '1' en 'Z', waarbij de 'Z' aangeeft dat de tri-state uitgang hoogohmig is. In hoofdstuk 2 is hiervoor al het aangepaste type `drie_waardige_logica` gedeclareerd waarvan ook nu weer gebruik is gemaakt. De resolutiefunctie krijgt op elk moment een vector van dit type aangeboden en gaat na of er maximaal één busdriver actief is. In de package `RESOLVE` (figuur 3.11) is dit uitgewerkt. De signalen die de busdriver ingaan zijn van het type `bit`, terwijl de bus van het type `drie_waardige_logica` is. Daarom is er ook een conversie-functie `BIT2DWL` geschreven die deze conversie uitvoert.

```
use work.drie_nivo_logica.all; -- zie figuur 2.28
package resolve is
  function bus_resolve (inp: drie_waardige_vector)
    return drie_waardige_logica;
  -- modellering van de wired_or
  subtype bus_bit is bus_resolve drie_waardige_logica;
  function bit2dwl (inp : bit)
    return drie_waardige_logica;
  -- type bit geconverteerd naar drie_waardige_logica;
  -- '0' -> '0' en '1' -> '1'
end resolve;

package body resolve is
  function bus_resolve (inp: drie_waardige_vector)
    return drie_waardige_logica is
  variable one_active : boolean := false;
  variable res : drie_waardige_logica := 'Z';
  begin
    for i in inp'range loop
      if inp(i)='0' or inp(i)='1' then
        res := inp(i);
        assert not one_active
          report "2 of meer tri-states actief" severity warning;
        one_active := true;
      end if;
    end loop;
    return res;
  end bus_resolve;
  function bit2dwl (inp : bit)
    return drie_waardige_logica is
  begin
    if inp='1' then return '1'; else return '0'; end if;
  end bit2dwl;
end resolve;
```

figuur 3.11 Package resolve.

Elke busdriver is beschreven met een proces dat afhankelijk van de enable-ingang de tri-state hoogohmig maakt of het ingangssignaal, na conversie van het type, doorgeeft aan de bus. Dit is, met enkele simulatieresultaten, gegeven in figuur 3.12.

In de volgende paragraaf wordt nog een andere beschrijving van hetzelfde probleem gegeven.

```

use work.drie_nivo_logica.all;
use work.resolve.all;
entity tri_states is
  port (enable1, enable2, enable3 : bit;
        i1,i2,i3                  : bit;
        uit                       : out bus_bit);
end tri_states;

architecture demo of tri_states is
begin
  bus_driver1:process(enable1,i1)
  begin
    if enable1='0' then uit <= 'Z'; else uit <= bit2dwl(i1); end if;
  end process;

  bus_driver2:process(enable2,i2)
  begin
    if enable2='0' then uit <= 'Z'; else uit <= bit2dwl(i2); end if;
  end process;

  bus_driver3:process(enable3,i3)
  begin
    if enable3='0' then uit <= 'Z'; else uit <= bit2dwl(i3); end if;
  end process;
end demo;

--- enkele simulatieresultaten
ns enable1 enable2 enable3 i1 i2 i3 uit
0      0      0      0      0 0 0 0
0      0      0      0      0 0 0 Z
0      0      0      0      0 1 0 Z
10     1      0      0      0 1 0 Z
10     1      0      0      0 1 0 1
20     0      0      0      0 1 0 1
20     0      0      0      0 1 0 Z
30     0      1      0      0 1 0 Z
30     0      1      0      0 1 0 0
40     1      1      0      0 1 0 0
40     1      1      0      0 1 0 1
# ** Warning: 2 of meer tri-states actief
#   Time: 40 ns   Iteration: 1

```

figuur 3.12 Modelleren van een bus met drie drivers.

3.5.3 Guarded signal.

Elk proces dat een toekenning doet aan een signaal creëert één driver voor dat signaal en verder bepalen alle drivers de resulterende waarde van het signaal. Echter het is ook mogelijk dat bepaalde drivers worden ‘uitgeschakeld’. Dit wordt bereikt met een guarded (resolved) signal. Op pagina 61 zijn bijvoorbeeld drie busdrivers aanwezig op een fysieke bus, waarbij de busdrivers actief of hoogohmig zijn. Dit hoogohmig zijn geeft al duidelijk aan hoe dit gerealiseerd gaat worden, in plaats van nu al naar de realisatie te kijken kan ook op een hoger abstractie niveau worden aangegeven dat een driver niet altijd meedoet bij het bepalen van de resulterende waarde van een signaal. Dit 'uitschakelen' van een driver wordt bereikt door gebruik te maken van een guarded signal.

Een guarded (resolved) signal is een signaal waarvan het type gekoppeld is aan een resolutiefunctie en bovendien is een guarded signal van het soort 'bus' of 'register'.

Declaratie van een guarded signal (met gebruikmaking van het subtype bus_bit, figuur 3.11) is:

signal uit1 : bus_bit **bus**;

signal uit2 : bus_bit **register**

Het onderscheid tussen *bus* en *register* wordt later behandeld. In de entity-declaratie mag ook een guarded signal worden opgenomen, echter deze mag alleen van het soort

bus zijn. Van een guarded signal kan de driver van een proces worden uitgeschakeld m.b.v. een *null-transaction*. Voorbeelden:

```
uit1 <= null after 10 ns;
```

```
uit2 <= null;
```

In het eerste voorbeeld wordt de driver na 10 ns 'uitgeschakeld' en in het tweede voorbeeld na een delta-delay. Op een andere manier kijkend naar het voorbeeld zoals op pagina 61 is uitgewerkt kan worden gezegd dat de niet-actieve busdrivers, die als hoogohmig gemodelleerd zijn, niet meedoen bij het bepalen van de resulterende waarde van de bus. Dit alternatief is uitgewerkt en gegeven in figuur 3.13.

Enkele opmerkingen met betrekking tot figuur 3.13:

- Door middel van onderstreping is het verschil aangegeven met de figuren 3.10 t/m 3.12. De functie BUS_RESOLVE is geheel herschreven.
De lokale declaratie van het signaal UIT is niet noodzakelijk, immers guarded signalen van het soort bus mogen ook in de interface worden opgenomen:

```
port ( ..  
      uit : out bus_bit bus;
```
- Omdat het probleem op een hoger abstractie niveau is beschreven is het type drie_waardige_logica weer vervangen door het type bit.
- De resolutiefunctie is zodanig opgezet dat er van uitgegaan is dat er maximaal één driver is, d.w.z. dat de bit_vector hooguit één element mag bevatten! Indien er geen enkele driver actief is komt er een '0' op de bus.
- Indien ENABLE1='0' waar is, draagt de driver van dit signaal van proces BUS_DRIVER1 niet bij aan het bepalen van de resulterende waarde van het signaal UIT.
- Merk op dat de use-clause voor de entity-beschrijving (**use** work.resolve_bit.bus_bit;) alleen maar het type hoeft te bevatten en niet de functie BUS_RESOLVE.

```

package resolve_bit is
  function bus_resolve (inp: bit_vector)
    return bit;
  subtype bus_bit is bus_resolve bit;
end resolve_bit;

package body resolve_bit is
  function bus_resolve (inp: bit_vector)
    return bit is
  begin
    assert inp'length <= 1 report "2 of meer drivers actief" severity warning;
    if inp'length = 1 then return inp(0); else return '0'; end if;
  end bus_resolve;
end resolve_bit;

-----
use work.resolve_bit.bus_bit;
entity tri_states1 is
  port (enable1, enable2, enable3 : bit;
        i1,i2,i3                : bit;
        o                        : out bit);
end tri_states1;

architecture demo of tri_states1 is
  signal uit : bus_bit bus;
begin
  o <= uit;

  bus_driver1:process(enable1,i1)
  begin
    if enable1='0' then uit <= null; else uit <= i1; end if;
  end process;

  bus_driver2:process(enable2,i2)
  begin
    if enable2='0' then uit <= null; else uit <= i2; end if;
  end process;

  bus_driver3:process(enable3,i3)
  begin
    if enable3='0' then uit <= null; else uit <= i3; end if;
  end process;
end demo;

enkele simulatieresultaten
ns enable1 enable2 enable3 i1 i2 i3 o
0      0      0      0      0 0 0 0
0      1      0      0      0 0 0 0
10     1      0      0      0 1 0 0
10     1      0      0      0 1 0 0
20     1      1      0      0 1 0 0
20     1      1      0      0 1 0 0
# ** Warning: 2 of meer drivers actief
# Time: 20 ns Iteration: 1
30     1      0      0      0 1 0 0
30     1      0      0      0 1 0 0
40     0      0      0      0 1 0 0
40     0      0      0      0 1 0 0

```

figuur 3.13 Guarded signalen voor het modelleren van de drie tri-states uit figuur 3.10.

Het tijdstip 40 ns is interessant om het verschil aan te geven tussen guarded signalen van het soort **bus** en het soort **register**. Voor dat tijdstip geldt dat het guarded signaal UIT de waarde '1' heeft, vervolgens worden *alle* drivers uitgeschakeld. Dit betekent dat de resolutiefunctie wordt aangeroepen met een bit_vector met lengte 0.

Indien een guarded signal van het soort register is, wordt de resolutiefunctie niet aangeroepen indien alle drivers uitgeschakeld zijn. De waarde van een guarded signal blijft onveranderd. Dit verklaart de naam 'register'.

Indien een guarded signal van het soort bus is, wordt de resolutiefunctie altijd aangeroepen. De resolutiefunctie bepaalt ook bij een bit-vector met lengte 0 de nieuwe waarde van het guarded signaal.

Figuur 3.14 geeft de enige wijziging ten opzichte van de VHDL beschrijving uit figuur 3.13 met de simulatieresultaten. Op tijdstip 40 ns zijn alle drivers uitgeschakeld, het uitgangssignaal blijft dus de waarde '1' behouden.

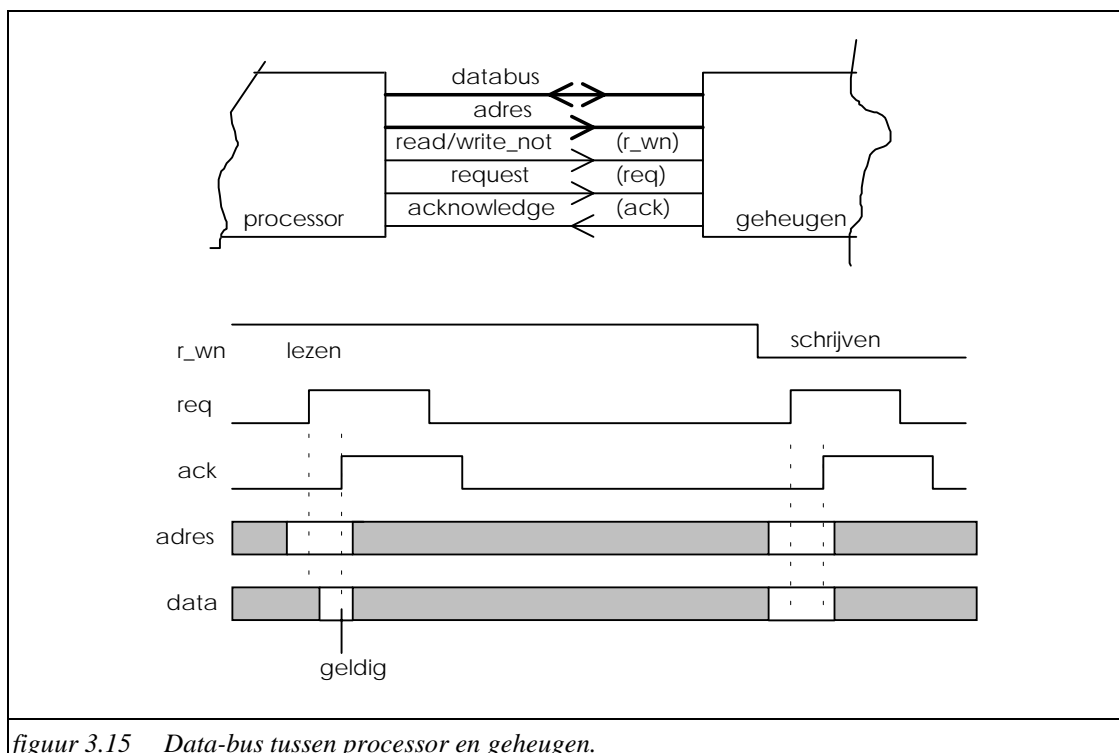
```
architecture demo of tri_states is
  signal uit : bus_bit register;
begin

  enkele simulatieresultaten.
  ns enable1 enable2 enable3 i1 i2 i3 o
  0      0      0      0      0 0 0 0
  0      1      0      0      0 0 0 0
  10     1      0      0      0 1 0 0
  10     1      0      0      0 1 0 1
  20     1      1      0      0 1 0 1
  20     1      1      0      0 1 0 0
  # ** Warning: 2 of meer drivers actief
  #   Time: 20 ns   Iteration: 1
  30     1      0      0      0 1 0 0
  30     1      0      0      0 1 0 1
  40     0      0      0      0 1 0 1
```

figuur 3.14 Guarded signal is van het soort register

3.5.4 Modelleren van een bus tussen processor en geheugen.

In deze paragraaf wordt aangegeven hoe de data-bus tussen een processor en een geheugen beschreven kan worden.



figuur 3.15 Data-bus tussen processor en geheugen.

Een data-bus tussen processor en geheugen is fysiek meestal uitgevoerd als een bidirectionele bus, figuur 3.15 geeft dit schematisch weer. Vanuit het standpunt van VHDL heeft signaal DATA_BUS dus twee drivers, de processor en het geheugen. Dit signaal is een resolved signal! Tevens is schematisch het communicatieprotocol weergegeven.

Twee VHDL beschrijvingen worden gegeven om dit probleem te beschrijven waarbij gebruik gemaakt zal worden van een *resolved signal* en een *guarded (resolved) signal*

voor het signaal DATA_BUS. De essentie ligt bij beide beschrijvingen op het 'aansturen' van de bidirectionele bus en niet op de beschrijving van de processor of het geheugen.

3.5.4.1 Resolved signal voor het beschrijven van de bus.

Figuur 3.16 geeft een beschrijving van de bidirectionele bus waarbij het signaal DATA_BUS een resolved signal is. Ook zijn de simulatieresultaten weergegeven waarin het communicatieprotocol is terug te vinden.

Enkele opmerkingen bij de figuren 3.16a t/m 3.16e:

- De package types (figuur 3.16a)
Signalen op de data-bus zijn bytes, en zijn in dit voorbeeld beschreven door middel van een subrange van het integer bereik. De resolutiefunctie BUS_RESOLVE sommeert de waarden van vector INP. Verder is nog een subtype voor het adres gegeven.
Op een lager abstractie niveau kan het probleem als volgt worden beschreven:
subtype byte **is** drie_waardige_vector (7 **downto** 0);
type byte_vector **is** **array** (natural **range** <>) **of** byte;
function bus_resolve (inp : byte_vector);
 return byte;
subtype bus_byte **is** bus_resolve byte;
Merk op dat één element van de byte_vector dus ook een array is.
- De processor (figuur 3.16b)
De processor wacht tot er opnieuw data moet worden gelezen van of geschreven worden naar het geheugen, daarom *wait on read*;. Ga na dat de beschrijving tussen de gereserveerde woorden **then** en **else** overeenkomen met het lezen van data uit het geheugen en tussen de gereserveerde woorden **else** en **end if** met het schrijven van data naar het geheugen. Indien de processor data naar het geheugen schrijft is dat de data van het signaal INP_CPU.
- Het geheugen (figuur 3.16c)
Het geheugen is eveneens niet gemodelleerd, data uit het geheugen is afkomstig van het signaal INP_GEH.
- De simulatie (figuur 3.16e)
Op tijdstip 100 ns wordt het signaal READ '1' gemaakt. Dit betekent dat de processor data van het geheugen wil hebben. Op tijdstip 105 ns wordt het signaal REQ '1' gemaakt, het geheugen plaatst vervolgens de data op de DATA_BUS en maakt het signaal ACK '1'. Daarna leest de processor de data en maakt het signaal REQ weer '0', vervolgens zal het geheugen ook het signaal ACK weer '0' maken.

```

package types is
  subtype byte is integer range 0 to 255;
  type byte_vector is array (natural range <>) of byte;
  function bus_resolve (inp : byte_vector)
    return byte;
  subtype bus_byte is bus_resolve byte;
  subtype address is bit_vector (15 downto 0);
end types;

package body types is
  function bus_resolve (inp : byte_vector)
    return byte is
  variable res : byte := 0;
  begin
    for i in inp'range loop
      res := res + inp(i);
    end loop;
    return res;
  end bus_resolve;
end types;

```

figuur 3.16a Package met types ten behoeve van modellering data-bus.

```

use work.types.all;
entity cpu is
  port (data_bus : inout bus_byte;
        adres : out address;
        r_wn : out bit; -- read/write not
        req : out bit; -- request
        ack : bit; -- acknowledge

        inp_cpu : byte;
        read : bit);
end cpu;

architecture gedrag of cpu is
  signal cpu_data : byte;
begin
  process
  begin
    wait on read;
    if read='1'
    then -- lees data uit geheugen
      -- adres <= geldig adres
      r_wn <= '1';
      req <= '1' after 5 ns;
      wait until ack='1'; -- data geldig
      cpu_data <= data_bus;
      req <= '0' after 5 ns;
      wait until ack='0';
    else -- schrijven naar geheugen
      data_bus <= inp_cpu;
      -- adres <= 'geldig adres'
      r_wn <= '0';
      req <= '1' after 5 ns;
      wait until ack='1'; -- data is gelezen
      data_bus <= 0;
      req <= '0' after 5 ns;
      wait until ack='0';
    end if;
  end process;
end gedrag;

```

figuur 3.16b Beschrijving communicatie deel processor.

```

use work.types.all;
entity geheugen is
  port (data_bus : inout bus_byte;
        adres    : address;
        r_wn     : bit;      -- read/write not
        req      : bit;      -- request
        ack      : out bit;  -- acknowledge

        inp_geh  : byte);
end geheugen;

architecture gedrag of geheugen is
  signal mem_data : byte;
begin
  process
  begin
    wait until req='1';
    if r_wn='1'
    then -- data from memory
      -- lees data uit geheugen adres
      data_bus <= inp_geh;
      ack <= '1' after 5 ns;
      wait until req='0';
      data_bus <= 0;
      ack <= '0' after 5 ns;
    else -- data in geheugen plaatsen
      mem_data <= data_bus;
      ack <= '1' after 5 ns;
      wait until req='0';
      ack <= '0' after 5 ns;
    end if;
  end process;
end gedrag;

```

figuur 3.16c Beschrijving communicatie deel geheugen.

```

use work.types.all;
entity test_cpu_geheugen is
  port (inp_cpu : byte;
        read    : bit;
        inp_geh : byte;
        data_bus : inout bus_byte);
end test_cpu_geheugen;

architecture structuur of test_cpu_geheugen is
  component cpu
  port (data_bus : inout bus_byte;
        adres    : out address;
        r_wn     : out bit; -- read/write not
        req      : out bit; -- request
        ack      : bit;    -- acknowledge

        inp_cpu  : byte;
        read     : bit);
  end component;

  component geheugen
  port (data_bus : inout bus_byte;
        adres    : address;
        r_wn     : bit;    -- read/write not
        req      : bit;    -- request
        ack      : out bit; -- acknowledge

        inp_geh  : byte);
  end component;

  signal adres : address;
  signal r_wn, req, ack : bit;
begin
  processor:cpu port map (data_bus,adres,r_wn,req,ack,inp_cpu,read);
  memory:geheugen port map (data_bus,adres,r_wn,req,ack,inp_geh);
end structuur;

```

figuur 3.16d Beschrijving koppeling tussen processor en geheugen.

enkele simulatieresultaten							
ns	inp_cpu	inp_geh	read	req	ack	data_bus	-- lezen uit geheugen
0	0	0	0	0	0	0	
0	10	20	0	0	0	0	
100	10	20	1	0	0	0	
105	10	20	1	1	0	0	
105	10	20	1	1	0	20	
110	10	20	1	1	1	20	
115	10	20	1	0	1	20	
115	10	20	1	0	1	0	
120	10	20	1	0	0	0	
ns	inp_cpu	inp_geh	read	req	ack	data_bus	-- schrijven naar geheugen
200	10	20	1	0	0	0	
200	10	20	0	0	0	0	
200	10	20	0	0	0	10	
205	10	20	0	1	0	10	
210	10	20	0	1	1	10	
210	10	20	0	1	1	0	
215	10	20	0	0	1	0	
220	10	20	0	0	0	0	

figuur 3.16e Simulatieresultaten

3.5.4.2 Guarded signal voor het beschrijven van de bus.

Figuur 3.17 geeft eveneens een beschrijving voor de bidirectionele bus maar nu is de bus beschreven met een guarded signal. De wijzigingen zijn minimaal ten opzichte van de beschrijving in figuur 3.16a t/m 3.16e:

- In de bus_resolve functie is nog een assert opgenomen om aan te geven dat er 2 of meer drivers zijn, dit om aan te kunnen geven of de processor en het geheugen misschien tegelijk data op de bus willen plaatsen.
- De guarded signals moeten nog van het soort 'bus' zijn.
- In de vorige paragraaf werd de data-bus als het ware 'vrij' gegeven, door daaraan 0 toe te kennen. Dit kan fraaier worden aangegeven door de driver uit te schakelen met `data_bus <= null;`

In figuur 3.17 zijn de wijzigingen gegeven ten opzichte van de figuren 3.16a t/m 3.16e.

<pre> package body types is function bus_resolve (inp : byte_vector) return byte is variable res : byte := 0; begin assert inp'length<=1 report "2 of meer drivers actief" severity warning; .. end bus_resolve; end types; de overige wijzigingen: - data_bus : inout bus_byte bus; in plaats van data_bus : inout bus_byte; - data_bus <= null; in plaats van data_bus <= 0; </pre>

figuur 3.17 Wijzigingen ten opzichte van figuur 3.16a t/m 3.16e.

3.5.5 Nogmaals de resolutiefunctie.

In figuur 3.10 is een bus gegeven met drie drivers. Voor deze beschrijving zijn twee manieren gegeven om dit te modelleren in VHDL:

- door gebruik te maken van een resolved signaal, figuur 3.12.
- door gebruik te maken van een guarded signaal, figuur 3.13.

Beide beschrijvingen hebben gemeenschappelijk dat het beschreven is met drie processen in één architecture. Een alternatieve beschrijving kan worden verkregen

door voor een busdriver eerst een entity en architecture te bepalen. Bijvoorbeeld voor de oplossing met de guarded signalen:

```

use work.resolve_bit.bus_bit;
entity driver1 is
  port (enable : in bit;
        inp    : in bit;
        uit    : out bus_bit bus);
end driver1;

architecture gedrag of driver1 is
begin
  process(enable,inp)
  begin
    if enable='1' then uit <= inp; else uit <= null; end if;
  end process;
end gedrag;

```

In de port declaratie is aangegeven dat het uitgangssignaal een guarded signaal is van het soort bus.

Vervolgens wordt de structuur beschrijving van figuur 3.10 herschreven in :

```

architecture structuur of tri_states1 is
  component driver1
    port (enable : in bit;
          inp    : in bit;
          uit    : out bus_bit bus);
  end component;
  signal uit : bus_bit bus;
begin
  o <= uit;
  bus_driver1: driver1 port map (enable1,i1,uit);
  bus_driver2: driver1 port map (enable2,i2,uit);
  bus_driver3: driver1 port map (enable3,i3,uit);
end structuur;

```

De lezer verwacht uiteraard dat het gedrag van deze beschrijving gelijk is aan dat van figuur 3.13, dit is echter niet het geval. Er is een verschil in het aantal keren dat de resolutiefunctie wordt aangeroepen. In de oorspronkelijk beschrijving, figuur 3.13, wordt slechts eenmaal de resolutiefunctie aangeroepen. In de alternatieve beschrijving wordt de resolutiefunctie veel vaker aangeroepen. Veronderstel dat alle enable signalen van '1' naar '0' gaan, dan wordt driemaal de resolutiefunctie aangeroepen, voor de drie entities, om een resulterende waarde te bepalen. Driemaal wordt deze functie aangeroepen met een lege lijst (dankzij de *null*-toekenning). Voor alle drie de uitgangssignalen wordt dus een resulterende waarde '0' bepaald. Deze drie '0'-en worden dan in de architecture-beschrijving 'structuur' gebruikt. In deze architecture wordt de resolutiefunctie dus met een vector van drie elementen aangeroepen (drie maal een '0'). En dus zal steeds de volgende melding worden afgedrukt (zie de resolutiefunctie):

"2 of meer drivers actief" **severity level** warning

Indien de beschrijving uit figuur 3.12 als uitgangspunt wordt gebruikt en deze op dezelfde wijze wordt omgezet, dan wordt het voorgaande probleem voorkomen dankzij de introductie van het niveau 'Z', immers iedere entity genereert een 'Z' indien de enable-ingang de waarde '0' heeft!

De volgorde waarin de lijst wordt samengesteld die aan de resolutiefunctie wordt aangeboden is niet vastgelegd in de standaard. Dit betekent dat de resolutiefunctie zowel *commutatief* en *associatief* moet zijn om een portable beschrijving te garanderen.

3.6 Het block statement.



figuur 3.18 Schematisch de interface met een block weergegeven.

Decompositie van een ontwerp is mogelijk door gebruik te maken van entiteiten en deze als component in een andere beschrijving te gebruiken. Decompositie is ook mogelijk door expliciet gebruik te maken van een block. Een block bakent een deelprobleem af van de omgeving waarin het is geplaatst. Door middel van een

interface, de port en de generic, kan worden gecommuniceerd met de omgeving waarin het block is geplaatst. Schematisch is dit weergegeven voor een VHDL beschrijving in figuur 3.18.

De syntax-beschrijving van een block statement is:

```
block [boolean expressie ]
  [ generic ]
  [ generic map ]
  [ port ]
  [ port map ]
  [ declaraties ]
begin
  [ concurrent statements ]
end block;
```

Op de boolean expressie achter het gereserveerde woord block wordt later nog nader ingegaan. De interface beschrijving is een optie, evenals het declareren van lokale signalen, types etc. Merk tevens op dat de statements in het block statement concurrent statements zijn (en geen sequentiële statements).

3.6.1 De relatie tussen een block en instantiatie van een component.

Deze paragraaf wil de relatie tussen een component-declaratie en een block toelichten. In hoofdstuk 2 is aangegeven hoe door middel van een component-declaratie gebruik gemaakt werd van al bestaande entiteiten. Opvallend daarbij was het dat de naamgeving van de formele parameters van de component-declaratie niet overeen hoefden te komen met de declaratie van de entiteit. Het zal zelfs blijken dat het *aantal* parameters van de generic en de port niet overeen moeten komen. Aan de hand van het nu volgende voorbeeld wordt duidelijk gemaakt dat een component-declaratie eigenlijk alleen maar een interface vastlegt. In figuur 3.19 is o.a. de VHDL beschrijving gegeven van de entity SR_LATCH met architecture GEDRAG1. In de component-declaratie zijn slechts drie parameters aanwezig. De mapping van entity SR_LATCH op deze component-declaratie vindt plaats door middel van de configuratie-specificatie:

```
for l:latch use entity work.sr_latch (gedrag1)
  generic map (delay=>vertraging)
  port map (s=>set,r=>reset,q=>q,q_not=>open);
```

Deze configuratie-specificatie kan achterwege blijven indien de component-declaratie en entity dezelfde naam en parameters hebben. In feite blijkt de algemeen gesuggereerde configuratie-specificatie -zie bijvoorbeeld figuur 2.21- niet algemeen, er is immers nog altijd sprake van een default generic map en een default port map! Indien de aantallen parameters niet met elkaar overeenkomen moet er ook nog expliciet een generic map en een port map worden opgenomen in de configuratie-specificatie.

Figuur 3.20 geeft een equivalente beschrijving waarbij de component-declaratie en de daaraan gekoppelde entity SR_LATCH vervangen zijn door een block statement. Deze beschrijving laat ook duidelijk zien dat de component-declaratie, die overeenkomt met de interface declaratie van block L, op zich zelf niets te maken heeft met de entity SR-LATCH.

Ook in de configuration kan deze algemenere koppeling worden ondergebracht (zie pagina 49).

```

entity sr_latch is
  generic (delay : time := 10 ns);
  port ( s,r      : bit;
        q, q_not : out bit);
end sr_latch;

architecture gedrag1 of sr_latch is
  signal qi      : bit;
  signal qi_not  : bit := '1';
begin
  qi <= r nor qi_not after delay;
  qi_not <= s nor qi after delay;
  q <= qi;
  q_not <= qi_not;
end gedrag1;

-----
entity q_output_latch is
  port ( setten, resetten : bit;
        uit : out bit);
end q_output_latch;

architecture demo of q_output_latch is
  component latch
    generic (vertraging : time);
    port (set,reset : bit;q : out bit);
  end component;
  signal tijd : time := 20 ns;
  for l:latch use entity work.sr_latch (gedrag1)
    generic map (delay=>vertraging)
    port map (s=>set,r=>reset,q=>q,q_not=>open);
begin
  l:latch generic map (vertraging => tijd)
    port map (set=>setten,reset=>resetten,q=>uit);
end demo;

  ns setten resetten uit
  0      0      0      0
  50     1      0      0
  90     1      0      1
  100    0      0      1
  150    0      1      1
  170    0      1      0
  200    0      0      0

```

figuur 3.19 Component-declaratie met minder parameters dan de entity.

```

entity q_output_latch is
  port ( setten, resetten : bit;
        uit : out bit);
end q_output_latch;

architecture demo of q_output_latch is
  signal tijd : time := 20 ns;
begin
  l:block
    generic (vertraging : time);
    generic map (vertraging=>tijd);
    port (set,reset : bit;q : out bit);
    port map (set=>setten,reset=>resetten,q=>uit);
  begin
    sr_latch:block
      generic (delay : time := 10 ns);
      generic map (delay=>vertraging);
      port (s,r : bit; q,q_not : out bit);
      port map (s=>set, r=>reset, q=>q, q_not=>open);
      signal qi      : bit;
      signal qi_not  : bit := '1';
      begin
        qi <= r nor qi_not after delay;
        qi_not <= s nor qi after delay;
        q <= qi;
        q_not <= qi_not;
      end block;
    end block;
  end demo;

```

figuur 3.20 Equivalente beschrijving van figuur 3.19 met een block.

3.6.2 Block met een boolean expressie (de guard).

Decompositie van het ontwerp door middel van een block wordt zelden toegepast. Echter een veel gebruikte beschrijving is een block met een boolean expressie, deze boolean expressie wordt guard genoemd. De guard heeft *alleen* betrekking op de *guarded signal assignment statements* in dat block, en is voorzien van het gereserveerde woord **guarded**. Indien de guard waar is worden de daarbij behorende signal assignment statements uitgevoerd. Is de guard niet waar dan worden de daarbij behorende drivers uitgeschakeld bij het bepalen van de resulterende waarde indien het signaal waaraan wordt toegekend een guarded signal is, en het signaal blijft onveranderd indien het geen guarded signaal is.

Figuur 3.21 geeft een voorbeeld van een guarded block: is de guard waar (GRD='1') dan wordt het signal assignment statement uitgevoerd en als de guard niet waar is krijgt het guarded signal een *null transaction*. De equivalente procesbeschrijving, eveneens gegeven in figuur 3.21, geeft dit duidelijk aan.

Enkele opmerkingen bij figuur 3.21:

- Merk op dat ook nu in het block *concurrent* signal assignment statements staan.
- De guards hebben geen invloed op de niet guarded signal assignment statements. Dit blijkt ook uit de equivalente procesbeschrijving. Ook de simulatieresultaten laten duidelijk zien dat signaal O1 het signaal I altijd volgt.
- Ook nu moet een guarded signal van het type bus of register zijn.

```

use work.resolve_bit.all;
entity guarded_block is
  port (i : bit; grd : bit;
        o1 : out bit; o2 : out bit);
end guarded_block;

architecture test of guarded_block is
  signal o : bus_bit bus;
begin
  b:block (grd='1')
  begin
    o <= guarded i;
    o1 <= i;
  end block;

  o2 <= o;
end test;

architecture equivalent of guarded_block is
  signal o : bus_bit bus;
begin
  b:block
  begin
    process
    begin
      if grd='1' then o <= i; else o <= null; end if;
      wait on grd,i;
    end process;
    o1 <= i;
  end block;

  o2 <= o;
end equivalent;

ns i grd o1 o2
0 0 0 0 0
10 1 0 0 0
10 1 0 1 0
20 0 0 1 0
20 0 0 0 0
30 0 1 0 0
40 1 1 0 0
40 1 1 1 0
40 1 1 1 1
50 0 1 1 1
50 0 1 0 1
50 0 1 0 0
60 1 1 0 0
60 1 1 1 0
60 1 1 1 1
70 1 0 1 1
70 1 0 1 0

```

figuur 3.21 Guarded signal assignment met equivalente procesbeschrijving.

Door gebruik te maken van een guarded signal assignment statement kan een expliciete toekenning van een null transaction achterwege blijven. In figuur 3.13 zijn procesbeschrijvingen gegeven voor de busdrivers. Figuur 3.22 geeft een andere, maar equivalente, beschrijving van proces BUS_DRIVER1 uit figuur 3.13.

```

bus_driver1:block (enable1='1')
begin
  uit <= guarded i1;
end block;

```

figuur 3.22 Busdriver beschreven met een guarded block.

```

entity reg is
  port (i : bit;
        enable : bit;
        o : out bit);
end reg;

architecture latch of reg is
begin
  b:block (enable='1')
  begin
    o <= guarded i;
  end block;
end latch;

architecture latch_equivalent of reg is
begin
  b:block
  begin
    process
    begin
      if enable='1'
        then o <= i;
        else null;
        end if;
      wait on enable,i;
    end process;
  end block;
end latch_equivalent;

```

figuur 3.23 Latch beschreven met een block.

Figuur 3.23 geeft een toepassing van een block waarbij een guarded signal assignment wordt gebruikt welke aan een niet guarded signaal een toekenning doet. Eveneens is nu ook de equivalente procesbeschrijving gegeven. Merk op dat er nu in de equivalente beschrijving geen acties worden uitgevoerd in het 'else' deel (de code *else null*; kan ook worden verwijderd).

3.6.2.1 Disconnectie-specificatie.

Bij het behandelen van het guarded block is aangegeven dat aan een guarded signal in een guarded signal assignment statement een null transaction wordt toegekend indien de guard false is. Dit betekent dat de desbetreffende driver niet meedoet bij het bepalen van de resulterende waarde door de resolutiefunctie. Al eerder is beschreven dat bij een null-transaction ook nog een after statement kan worden gebruikt (bijvoorbeeld *null after 10 ns*;). Dit is ook mogelijk bij een guarded signal assignment statement door middel van een expliciete disconnectie- specificatie. Figuur 3.24 geeft dit aan voor nagenoeg dezelfde architecture-beschrijving uit figuur 3.21. Bij het ontbreken van een disconnectie-specificatie, zoals in figuur 3.21, wordt impliciet uitgegaan van een disconnectie-specificatie met tijd 0 ns, voor figuur 3.21:

disconnect o : bus_bit **after** 0 ns;

```

architecture test of guarded_block is
signal o : bus_bit bus;
disconnect o : bus_bit after 10 ns;
begin
b:block (grd='1')
begin
o <= guarded i;
o1 <= i;
end block;
o2 <= o;
end test;

architecture equivalent of guarded_block is
signal o : bus_bit bus;
begin
b:block (grd='1')
begin
process
begin
if grd='1'
then o <= i;
else o <= null after 10 ns;
end if;
wait on grd,i;
end process;
o1 <= i;
end block;
o2 <= o;
end equivalent;

ns i grd o1 o2
0 0 0 0 0
100 1 0 0 0
100 1 0 1 0
200 1 1 1 0
200 1 1 1 1
300 1 0 1 1
310 1 0 1 0
400 0 0 1 0

```

figuur 3.24 Expliciete disconnectie-specificatie in block met guard.

3.7 Passieve processen in de entity-declaratie.

```

entity flipflop is
generic (setup : time := 5 ns;
hold : time := 10 ns;
delay : time := 20 ns);
port (d,clk : bit;
q : out bit);
begin
process(d,clk)
begin
-- setup violation
if clk'event and clk='1' then
assert d'stable(setup) report "setup violation" severity warning;
end if;
-- hold violation
if d'event and clk='1' then
assert clk'stable(hold) report "hold violation" severity warning;
end if;
end process;
end flipflop;

architecture gedrag of flipflop is
begin
process
begin
wait until clk='1';
q <= d after delay;
end process;
end gedrag;

```

figuur 3.25 Passieve process statements in de entity-declaratie van een flipflop.

In de entity-declaratie mogen ook nog zogenaamde *passieve processen* worden opgenomen. Een proces is passief indien noch in het proces, noch in de daarin

gebruikte subprogramma's, signaal-toekenningen voorkomen. Een voorbeeld hiervan is gegeven in figuur 3.25. Hier wordt de setup violation en hold violation gecontroleerd door middel van een passief proces welke opgenomen is in de entity-declaratie.

3.8 Access types.

Het dynamisch alloceren, en het vrijgeven, van geheugen ruimte tijdens de executie van een beschrijving is mogelijk in VHDL. Daarvoor heeft VHDL, net als ADA, het access type (in PASCAL en C een pointer genoemd). In VHDL kan dit o.a. worden toegepast om lijsten te modelleren. Ook kan het handig worden gebruikt om een groot geheugen te modelleren, vooral als slechts een klein deel van het geheugen wordt gebruikt. Voor een geheugen zijn de volgende opeenvolgende type-declaraties bruikbaar:

```

type mem;
type pointer is access mem;
type mem is
  record
    address : natural;
    content : integer;
    nxt     : pointer;
  end record;

```

Bij dergelijke type-declaraties moet voor het declareren van het type pointer het type mem bekend zijn, en omgekeerd. Daarom voorziet de taal in een zogenaamde *incomplete type-declaratie*. De eerste regel (**type** mem;) geeft aan dat er een type mem bestaat, maar is niet volledig. Nadat het type pointer is gedeclareerd volgt een volledige definitie van het type mem.

```

package memory_types is
  type mem;
  type pointer is access mem;
  type mem is
    record
      address : natural;
      content : integer;
      nxt     : pointer;
    end record;
  procedure search_address(variable current : inout pointer; addr : in natural);
  procedure read_memory (variable location : inout pointer; signal data : out
integer);
  procedure write_memory (variable location : inout pointer; data : in integer);
end memory_types;

package body memory_types is
  procedure search_address(variable current : inout pointer; addr : in natural) is
    variable previous : pointer := null;
  begin
    while current.address < addr and current.nxt /= null loop
      previous := current;
      current := current.nxt;
    end loop;
    -- current.address >= addr or end reached
    if current.address /= addr
    then
      if current.address < addr
      then -- append at end of list
        current.nxt := new mem'(addr,0,null);
        current:=current.nxt;
      else -- insert in list
        previous.nxt := new mem'(addr,0,previous.nxt);
        current := previous.nxt;
      end if;
    end if;
  end search_address;

  procedure read_memory (variable location : inout pointer; signal data : out integer)
is
  begin
    data <= location.content;
  end read_memory;

  procedure write_memory (variable location : inout pointer; data : in integer) is
  begin
    location.content:=data;
  end write_memory;
end memory_types;

use work.memory_types.all;
entity memory is
  port (RWn      : in bit;
        address  : in natural;
        data_in  : in integer;
        data_out : out integer);
end memory;

architecture behaviour of memory is
begin
  process (RWn,address,data_in)
    variable location : pointer := null;
    variable root : pointer := new mem'(0,0,null);
  begin
    location:=root;
    search_address(location,address);
    if RWn='0' then -- write to memory
      write_memory(location,data_in);
    end if;
    read_memory(location,data_out);
  end process;
end behaviour;

```

figuur 3.26 Modelleren van een geheugen met een access type.

Figuur 3.26 geeft een volledige beschrijving van een geheugen waarbij gebruik is gemaakt van het access type. De procedure `search_address` creëert eventueel een nieuwe geheugen locatie door gebruik te maken van de functie `new`.

De geheugenruimte kan worden vrijgegeven door de impliciet gedeclareerde procedure `deallocate`.

3.9 Files.

VHDL geeft een beperkte ondersteuning voor het gebruik van files. Tijdens een simulatiesessie kan een file niet gemakkelijk geopend en gesloten worden, bijvoorbeeld om eerst data weg te schrijven naar een file om deze vervolgens weer in te lezen. Voor een geformateerde file i/o, in de volgende paragraaf, zal getoond worden hoe een file meerdere malen geopen en gesloten kan worden tijdens het simuleren.

VHDL maakt onderscheid in twee soorten files:

- De geformateerde file.
- De tekst file.

3.9.1 Geformateerde file i/o.

```
entity write_file is
end write_file;

architecture test of write_file is
  type data is file of integer;
  file out_int : data is out "int.dat";
begin
  process
  begin
    for i in 0 to 10 loop
      write(out_int,i);
    end loop;
    wait;
  end process;
end test;

-----

entity read_file is
end read_file;

architecture test of read_file is
  type data is file of integer;
  file in_int : data is in "int.dat";
begin
  process
    variable data : integer;
  begin
    loop
      exit when endfile(in_int);
      read(in_int,data);
    end loop;
    wait;
  end process;
end test;
```

figuur 3.27 Schrijven naar en lezen van een geformateerde file.

In figuur 3.27 zijn twee beschrijvingen gegeven waarbij een file wordt geschreven en gelezen, respectievelijk de entity `write_file` en de entity `read_file`.

Voor het schrijven van een file zijn twee declaraties nodig:

type data is file of integer;

file out_int : data is out "int.dat";

De eerste declaratie geeft het type van de file aan, en de daarop volgende declaratie koppelt de logische naam met de fysieke naam van de file.

Tevens wordt impliciet de functie `write` gedeclareerd, waarmee het mogelijk is om data weg te schrijven naar de file. De gegenereerde file is niet leesbaar voor de gebruiker, althans dit is niet noodzakelijk, en ook potentieel niet portable. In dit voorbeeld zijn de getallen 0 t/m 10 opgeslagen in de file. Tevens zorgt de file-declaratie er voor dat de file op 'dat moment' wordt geopend voor schrijven. Pas nadat de file-declaratie is beëindigd wordt de file gesloten en dat is dus op het moment dat

de simulatie wordt beëindigd. Pas tijdens een volgende sessie kan deze file weer worden gelezen. Ook dan zijn twee declaraties nodig waarbij de tweede declaratie aangeeft dat van de file gelezen gaat worden (de mode is *in*). Deze declaratie genereert eveneens impliciete subprogramma's, o.a. *read*.

```
entity write_and_read_same_file is
end write_and_read_same_file;

architecture demo of write_and_read_same_file is
  procedure write_file (FileOut : in String) is
    type data is file of integer;
    file OutInt : data is out FileOut;
  begin
    for i in 0 to 10 loop
      write(OutInt,i);
    end loop;
  end write_file;

  procedure add_contents_of_file (FileIn : in String;
                                total : out integer) is
    type data is file of integer;
    file InInt : data is in FileIn;
    variable int, sum : integer;
  begin
    sum := 0;
    while not endfile (InInt) loop
      read(InInt,int);
      sum := sum + int;
    end loop;
    total := sum;
  end add_contents_of_file;

begin
  process
    variable s : integer;
  begin
    write_file("demo.int");
    add_contents_of_file("demo.int", s);
    wait;
  end process;
end demo;
```

figuure 3.28

In figuur 3.28 is aangegeven hoe een file tijdens een simulatiesessie meerdere malen geopend en gesloten kan worden, voor zowel lezen van als schrijven naar de file. In de procedures staan de file-declaraties, dat wil zeggen dat pas tijdens de executie van de declaraties in deze procedures de file wordt gecreëerd voor lezen of schrijven. Na het beëindigen van de procedure zijn ook de file-declaraties niet meer geldig en dus wordt ook de file afgesloten.

3.9.2 Tekst file i/o.

Een file gecreëerd met geformateerde file i/o is niet uitwisselbaar met een andere VHDL omgeving. Indien dit gewenst is moet gebruik worden gemaakt van een tekst file.

VHDL is regel georiënteerd als het gaat om het gebruik van files. Een file bestaat dus uit een aantal regels, die op hun beurt weer bestaat uit een aantal karakters. Daarom is in de package *textio*, die onderdeel uitmaakt van de standaard, de volgende type-declaratie aanwezig:

type text is file of string;

Met in de package *standard*:

type string is array (positive range <>) of characters;

Data wordt per regel van een file gelezen of naar een file geschreven door middel van de volgende procedures (ook in de package *textio*):

```

procedure readline(f : in text; l: out line);    -- lezen van file
procedure writeline(f : out text; l: in line);    -- schrijven naar file

```

Vervolgens kan dan per regel de gewenste data worden gelezen. Eveneens zijn hiervoor in de package `textio` een aantal procedures opgenomen. Voor het type `bit` zijn de aanwezige procedure declaraties:

```

procedure read (l : inout line; value: out bit);
procedure read (l : inout line; value: out bit; good : out boolean);

```

Deze procedures zijn eveneens gedeclareerd voor de types:

- `bit_vector`
- `boolean`
- `character`
- `integer`
- `real`
- `string`
- `time`

Voor het schrijven zijn eveneens een aantal procedures beschreven in de package `textio`. In figuur 3.29 is een voorbeeld gegeven van een VHDL beschrijving die een file `data.inp` leest. De eerste 4 posities van een regel vormen de naam en vervolgens wordt een integer gelezen. De invoerfile zou bijvoorbeeld het volgende kunnen zijn:

```

abc    12
xyz    24

```

```

entity read_file is
end read_file;

use std.textio.all;
architecture test of read_file is
begin
  process
    variable data : integer;
    variable naam : string(1 to 4);
    variable inp_line : line;
    file in_data : text is "data.inp";
  begin
    loop
      exit when endfile(in_data);
      readline(in_data,inp_line);
      read(inp_line,naam);
      read(inp_line,data);
    end loop;
    wait;
  end process;
end test;

```

figuur 3.29 Lezen van een tekst file.

3.10 Koppeling van niet compatibele types in component instantiaties

Door bedrijven worden ook modellen gemaakt van bestaande componenten, o.a. beschreven in VHDL. Hierdoor kan een compleet systeem worden beschreven en gesimuleerd waarbij de ontwerper zelf niet alle componenten hoeft te beschrijven. Echter het is mogelijk dat de gebruikte typering afwijkt van het door U gekozen type. Veronderstel dat de ontwerper de volgende entity wil gebruiken:

```

entity dummy is
  port (a : in bit; b : out bit; c : inout bit);
end dummy;

```

Echter in zijn beschrijving wordt consequent gebruik gemaakt van het *type mvl is* ('0','1','Z');. Dus er moet een type-conversie plaats vinden van het type mvl naar het type bit en omgekeerd. In de package conversion heeft de gebruiker deze conversie-functies geplaatst, namelijk:

```
package conversion is
  type mvl is ('0','1','Z');
  function mvl2bit (a : in mvl) return bit;
  function bit2mvl (a : in bit) return mvl;
end conversion;

package body conversion is
  function mvl2bit (a : in mvl) return bit is
  begin
    if a='1' then return '1'; else return '0'; end if;
  end mvl2bit;
  function bit2mvl (a : in bit) return mvl is
  begin
    if a='1' then return '1'; else return '0'; end if;
  end bit2mvl;
end conversion;
```

Figuur 3.30 geeft aan hoe de type-conversie-functies gebruikt worden om van type bit naar mvl te gaan en omgekeerd. Let hier vooral op de mode van de formele parameters en de plaats waar de conversie-functie komt. Zo mag een formele parameter van de mode out, zoals parameter b, niet worden gelezen en dus is het volgende niet toegestaan:

```
b      => mvl2bit(y),
```

Eveneens mag aan formele parameters van de mode in, zoals parameter a, geen waarde worden toegekend. Het volgende is niet toegestaan:

```
bit2mvl(a) => x,
```

```
use work.conversion.all;
entity incompatible_types is
  port (x : in mvl; y : out mvl; z : inout mvl);
end incompatible_types;

architecture demo of incompatible_types is
  component dummy
    port (a : in bit; b : out bit; c : inout bit);
  end component;
begin
  good : dummy port map ( a      => mvl2bit(x),
                        bit2mvl(b) => y,
                        bit2mvl(c) => mvl2bit(z));

end demo;
```

figuur 3.30 Gebruik van type-conversie in de component binding.

Merk op dat bij de mode inout de conversie-functie zowel bij de formele en/of actuele parameter mag staan. De plaats waar deze conversie-functie staat bepaald echter mede het gedrag. Om dit duidelijk te maken is in figuur 3.31 een voorbeeld opgenomen waarin de conversie-functie op de drie mogelijke manieren is gebruikt. Tevens is ook mode buffer opgenomen. Op pagina 25 is al aangegeven dat mode

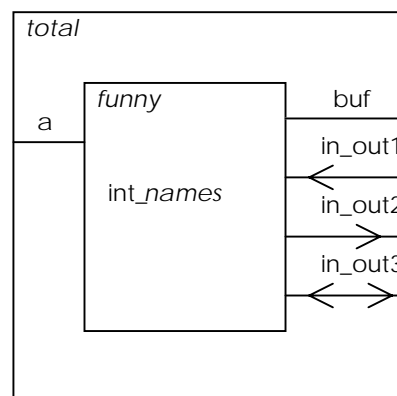
buffer op die plaatsen gebruikt kan worden waar een uitgangssignaal ook intern weer gelezen moet worden.

Het is niet toegestaan om type-conversie-functies voor unconstrained arrays op te nemen in de port map.

In figuur 3.31 zijn ook enkele simulatieresultaten gegeven. Veronderstel dat op een gegeven moment het signaal *a* van 0 naar 100 gaat, de iteraties (delta delays) die ten gevolge van deze verandering, op hetzelfde tijdstip, worden doorlopen zijn in dit figuur gegeven.

Enkele opmerkingen met betrekking tot de simulatieresultaten:

- signalen *in_out1* en *int_in_out1*
Het signaal *a* (=100) wordt via de entity funny doorgegeven aan poort *in_out1*. Dus na één iteratie wordt *in_out1* gelijk aan 100. Aan de zijde van de actuele parameter, welke tevens als input fungeert voor entity funny (zie ook mode in), wordt dit signaal met 10 opgehoogd, zodat het signaal *int_in_out1* na de volgende iteratie gelijk wordt aan 110.
- signalen *in_out2* en *int_in_out2*
Het signaal *a* (=100) wordt via de entity funny doorgegeven aan poort *in_out2*. Aan de van de zijde formele parameter bevindt zich de conversie-functie. Dus na één iteratie wordt *in_out2* gelijk aan 100+10=110. Aan de zijde van de aktuele parameter, welke tevens als input fungeert voor entity funny (zie ook mode in), is geen conversie-functie aanwezig, zodat het signaal *int_in_out2* na de volgende iteratie gelijk wordt aan 110.
- signalen *in_out3* en *int_in_out3*
Het signaal *a* (=100) wordt via de entity funny doorgegeven aan poort *in_out3*. Aan de zijde van de formele parameter bevindt zich de conversie-functie. Dus na één iteratie wordt *in_out3* gelijk aan 100+10=110. Aan de zijde van de aktuele parameter, welke tevens als input fungeert voor entity funny (zie ook mode in), is ook een conversie-functie aanwezig, zodat het signaal *int_in_out2* na de volgende iteratie gelijk wordt aan 110+10=120.
- signalen *buf* en *int_buf*
Mode buffer wijkt in het geheel af van mode inout als het gaat om de conversie-functie. Het signaal dat aan een uitgang van mode buffer wordt toegekend houdt wel rekening met de conversie-functie, echter als dat uitgangssignaal intern wordt gelezen wordt altijd direct de source gelezen, en niet de waarde die verkregen is via een conversie-functie.



pijlrichting geeft aan naar welke richting de conversie funktie werkt

```

package funny_conversion is
    function increment_10 ( inp : in natural) return natural;
end funny_conversion;

package body funny_conversion is
    function increment_10 ( inp : in natural) return natural is
    begin
        return inp+10;
    end increment_10;
end funny_conversion;

entity funny is
    port ( a : natural;
          buf : buffer natural;
          in_out1 : inout natural;
          in_out2 : inout natural;
          in_out3 : inout natural);
end funny;

architecture demo of funny is
    signal int_buf, int_in_out1, int_in_out2, int_in_out3 : natural;
begin

    buf <= a;
    int_buf <= buf;

    in_out1 <= a;
    int_in_out1 <= in_out1;

    in_out2 <= a;
    int_in_out2 <= in_out2;

    in_out3 <= a;
    int_in_out3 <= in_out3;

end demo;

entity total is
    port ( a : in natural;
          buf : buffer natural;
          in_out1 : inout natural;
          in_out2 : inout natural;
          in_out3 : inout natural);
end total;

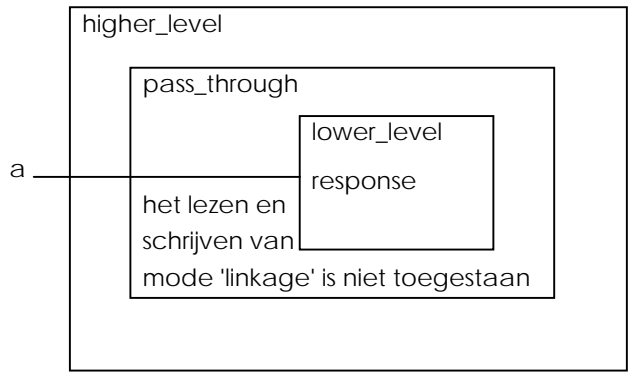
use work.funny_conversion.all;
architecture test of total is
    component funny
        port ( a : natural;
              buf : buffer natural;
              in_out1 : inout natural;
              in_out2 : inout natural;
              in_out3 : inout natural);
    end component;
begin
    i : funny port map ( a,
        increment_10(buf)      => increment_10(buf),
        in_out1               => increment_10(in_out1),
        increment_10(in_out2) => in_out2,
        increment_10(in_out3) => increment_10(in_out3) );
end test;

```


simulatieresultaat:									
delta	a	buf	in_out1	in_out2	in_out3	int_in_out1	int_in_out2	int_in_out3	int_buf
+0	100	10	0	10	10	10	10	20	0
+1	100	110	100	110	110	10	10	20	0
+2	100	110	100	110	110	110	110	120	100

figuur 3.31 Conversie-functies voor de modes inout en buffer.

3.11 Mode linkage



```

entity lower_level is
  port ( a : in bit);
end lower_level;

architecture echo of lower_level is
  signal response : bit;
begin
  response <= a;
end echo;
-----

entity pass_through is
  port ( a : linkage bit);
end pass_through;

architecture structure of pass_through is
  component lower_level
    port (a : in bit);
  end component;
  signal fout : bit;
begin
  -- fout <= a; -- Van een mode linkage mag niet geschreven noch gelezen worden
  l : lower_level port map (a);
end structure;
-----

entity higher_level is
  port ( a : in bit);
end higher_level;

architecture stimuli of higher_level is
  component pass_through
    port ( a : linkage bit);
  end component;
begin
  z : pass_through port map (a);
end stimuli;

```

Enkele simulatieresultaten:

ns	delta	a	response
0	+0	0	0
100	+0	1	0
100	+1	1	1
200	+0	0	1
200	+1	0	0

figuur 3.32 Toepassing van mode linkage.

In figuur 3.32 is schematisch weergegeven dat data (signaal a) van component higher_level, via component pass_through, doorgegeven moet worden naar component lower_level. In pass_through is het niet toegestaan het object a te lezen noch te schrijven. Dit wordt bereikt door hiervoor mode linkage te gebruiken.

Het is nu niet toevallig dat de naam van de formele parameter voor pass_through (=a) gelijk is aan die van de formele parameter van higher_level. Zou dit niet het geval zijn, bijvoorbeeld de formele parameter van pass_through is b, dan zou de architecture stimuli gewijzigd moeten worden in:

```
architecture stimuli of higher_level is
  component pass_through
    port ( b : linkage bit);
  end component;
begin
  z : pass_through port map (b => a);
end stimuli;
```

Echter de gegeven port map "b => a" is niet toegestaan.

3.12 Attributen gedefinieerd door de gebruiker

Door middel van attributen kan op eenvoudige wijze informatie worden verkregen over objecten. In de standaard omgeving zijn een groot aantal attributen gedefinieerd. Ook is het mogelijk dat de gebruiker zelf attributen definieert.

In figuur 3.33 is een attribuut gedeclareerd van het type string (**attribute** str : string;). Vervolgens vindt de specificatie van het attribuut plaats door het te koppelen aan het signaal sg en met de waarde %hello% (een andere schrijfwijze voor "hello").

Vervolgens kan dit attribuut later worden gebruikt op dezelfde wijze zoals dat voor de voorgedefinieerde attributen geldt.

```
entity attr is
  port(a : in integer;
       b : out integer;
       c : out string(1 to 5));
end attr;

architecture demo of attr is
  signal sg : integer;
  attribute str : string;
  attribute str of sg : signal is %hello%;
begin
  sg <= a;
  b <= sg'str'length;
  c <= sg'str;
end;
```

figuur 3.33 Attribuut voor een signaal.

De door de gebruiker gedefinieerde attributen hebben als nadeel dat ze lokaal blijven. Dus indien de informatie op een hoger niveau gewenst is, moet dit door middel van signalen worden doorgegeven; bijvoorbeeld signaal c in figuur 3.33.

Figuur 3.34 geeft een beschrijving waarin de attributen breedte en hoogte worden gebruikt om de afmetingen van een component vast te leggen. Deze afmetingen hebben alleen betrekking op de architecture. D.w.z. dat een andere architecture bij dezelfde entity andere afmetingen kan hebben. Ook de entity merkwaardige_inverter laat zien hoe deze informatie doorgegeven kan worden. Is het wenselijk om dat op deze manier te doen?

De door de gebruiker gedefinieerde attributen kunnen zinvol zijn voor andere tools. Zo kan een tool die de componenten moet plaatsen de attributen lengte en breedte

inspecteren. Voor synthese-tools wordt soms door middel van een attribuut aangegeven welk signaal als kloksignaal fungeert, of welke codering (binair, gray, one-hot, ..) voor een toestandsmachine gebruikt moet worden.

```
package lgt is
  type lengte is range 0 to 10e9
  units
    um;          -- micro meter
    mm  = 1000 um; -- milli meter
    m   = 1000 mm; -- meter
  end units;
  attribute hoogte : lengte;
  attribute breedte : lengte;
end lgt;

use work.lgt.all;
entity inverter is
  port ( a : in bit; b : out bit);
end inverter;

architecture model of inverter is
  attribute hoogte of model : architecture is 10 um;
  attribute breedte of model : architecture is 5 um;
begin
  b <= not a after 10 ns;
end model;

use work.lgt.all;
entity merkwaardige_inverter is
  port ( a : in bit; b : out bit;
        hgt, brd : out lengte);
end merkwaardige_inverter;

architecture model of merkwaardige_inverter is
  attribute hoogte of model : architecture is 10 um;
  attribute breedte of model : architecture is 5 um;
begin
  b <= not a after 10 ns;
  hgt <= model'hoogte;
  brd <= model'breedte;
end model;
```

figuur 3.34 Attributen voor de afmetingen van een component.

In figuur 3.33 is de attribuut specificatie gebruikt voor een signaal, en in figuur 3.34 voor een architecture. In het algemeen kan de gebruiker attributen specificeren voor een:

- entity
- architecture
- configuration
- procedure
- function
- package
- type
- subtype
- constant
- signal
- variable
- component
- label

3.13 Samenvatting.

Indien een entity of een package opnieuw geanalyseerd is moeten alle architecture-beschrijvingen die deze entity of package gebruiken ook opnieuw geanalyseerd

worden. Dus als alleen de architecture of package body is gewijzigd compileer dan *niet* de daarbij behorende entity of package.

Een configuration maakt het mogelijk om de koppeling van entity en package uit te stellen tot het laatste moment.

Drie vertragsmodellen worden ondersteund: delta, inertial en transport-delay. Het verschil tussen inertial- en transport-delay is dat een verandering, hoe kort ook, altijd wordt doorgegeven bij een transport-delay terwijl voor inertial-delay moet gelden dat de verandering tenminste gedurende de aangegeven vertragingstijd stand moet houden. Inertial-delay is het default vertragsmodel.

Functies kunnen een resultaat geven aan een variabele en aan een signaal. Een procedure kan default alleen een toekenning doen aan een variabele. Het is ook mogelijk dat een procedure alleen resultaten aan een signaal kan geven, dit moet dan expliciet in de beschrijving worden aangegeven.

Indien een signaal meerdere sources heeft moet het een resolved signaal zijn. Het gebruikte subtype voor een dergelijk signaal heeft naast een type ook een resolutiefunctie. De resolutiefunctie bepaalt dan de resulterende waarde voor dit signaal.

Met een guarded (resolved) signaal kunnen ook nog drivers worden uitgeschakeld d.m.v een null-toekenning of door gebruik te maken van een guarded signal assignment. Guarded signalen moeten van het type bus of register zijn. Indien alle drivers zijn afgesloten en het signaal is van het soort register dan blijft de waarde onveranderd. Een guarded signaal van het soort bus maakt altijd gebruik van de resolutiefunctie om de resulterende waarde te bepalen.

Decompositie van een beschrijving is mogelijk door middel van het block statement. Een block met een guard wordt altijd in combinatie met een guarded signaal gebruikt. Indien een guard niet waar is krijgt de driver van de guarded signal assignment statement een null-toekenning.

4. VHDL 1076-1993

4.1 Onderhoud aan de standaard.

VHDL is een complexe taal en de LRM (Language Reference Manual) probeert deze compleet en zonder fouten te beschrijven. Echter de VASG (VHDL Analysis and Standardization Group van de IEEE) voorzag dat er zeker fouten en onduidelijkheden in de LRM zouden zitten. Na 6 maanden heeft de VASG een groep samengesteld welke verantwoordelijk is voor het onderhoud aan de standaard: ISAC (Issues Screening and Analysis Committee). De ISAC heeft tot doel om te komen tot oplossingen voor aangedragen problemen met de taal en de LRM. Oplossingen worden doorgegeven aan de VASG, alwaar zij worden besproken en mogelijk goedgekeurd.

In juni 1990 begon de VASG te werken aan de nieuwe versie van de standaard, met als doel om in december 1992 opnieuw tot goedkeuring van de VHDL standaard over te kunnen gaan (iedere vijf jaar ‘verloopt’ een standaard automatisch).

Eerst werden de wensen van gebruikers en voorstellen tot wijzigingen van de taal verzameld. Er zijn circa 500 voorstellen (inclusief een aantal al eerder vastgestelde zaken) ingediend. Vervolgens werd langdurig over de consequenties van voorstellen gesproken. O.a. werd voorgesteld dat een ‘variant record’ ook mogelijk moest zijn. Dit gaf echter problemen met de taal indien er op een event van zo’n veld werd gewacht. Daarom is dit niet opgenomen in de nieuwe standaard. Ook zijn er voorstellen geweest die een significante wijziging van de taal zouden inhouden, maar mogelijk op termijn wel wenselijk werden geacht; deze zijn doorgeschoven naar de volgende standaard.

Uiteindelijk werd in september een draft versie 1076-1992/A voor goedkeuring aan geïnteresseerde IEEE leden rondgezonden. Hoewel 82% van de stemmers voor deze standaard hebben gestemd, is er toch nog een nieuwe draft versie 1076-1992/B geschreven i.v.m. niet te implementeren eigenschappen in de voorgestelde versie. Dat slechts 82% voor was is met name veroorzaakt door het toevoegen van een ‘shared variable’ (globale variable) aan de taal (zie ook later in dit hoofdstuk).

Uiteindelijk zijn fouten en correcties aangebracht zodat de IEEE op 15 september 1993 de nieuwe standaard heeft goedgekeurd, met als grote verrassing dat het niet de Std 1076-1992 is gaan heten maar Std 1076-1993. Boeken die het dus hebben over de VHDL’92 kunnen ook doelen op een draft versie en dat komt voor!

Gelukkig is de taal in de meeste opzichten upwards compatible. Wel zijn er enkele gereserveerde woorden toegevoegd waardoor soms een ‘oude’ beschrijving moet worden aangepast.

Dit hoofdstuk probeert globaal een beeld te geven van belangrijke wijzigingen. Bedenk daarbij wel dat vele CAE tools nog enkele jaren nodig zullen hebben om deze standaard te ondersteunen. Voorzichtigheid is dus geboden bij het gebruik van al het nieuws.

4.2 Aanpassingen in de syntax van de taal.

De syntax van LRM 1076-1987 is op een aantal plaatsen niet consequent als het gaat om het gebruik van de gereserveerde woorden. Zo wordt tijdens het invoeren van een component-declaratie vaak uitgegaan van een kopie van de entity-declaratie. Bijvoorbeeld:

entity demo is

```

    port (a : in bit; b : out bit);
end demo;

```

en de volgende hiervan afgeleide component-declaratie wordt vaak gebruikt:

```

component demo
    port (a : in bit; b : out bit);
end component;

```

De volgende aanpassingen moesten worden gemaakt om van de entity-declaratie een correcte component-declaratie te maken:

- het gereserveerde woord *is* is verwijderd.
- de herhaling van de naam van de entity is gewijzigd in het gereserveerde woord *component*.
- het gereserveerde woord *entity* is gewijzigd in *component*.

In LRM 1076-1993 is het wel toegestaan om gereserveerde woorden zoals entity (maar ook architecture, package, function, procedure, configuration, etc.) te herhalen; dus *end entity*. Bovendien mag het gereserveerde woord *is* in de component-declaratie blijven staan. Ook mag de naam van de component worden herhaald; *end component demo*;

In de volgende syntax-beschrijving voor een entity en de component-declaratie is dit nogmaals aangegeven. Merk op dat de uitbreiding optioneel is; hiermee voldoen VHDL beschrijvingen gebaseerd op LRM 1076-1987 ook aan de LRM 1076-1993.

```

component_declaration ::=
    component identifier [ is ]
        [ local_generic_clause ]
        [ local_port_clause ]
    end component [ component_simple_name ] ;

```

```

entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity ] [ entity_simple_name ] ;

```

4.3 Globale variabelen.

Als er één onderwerp is geweest wat de gemoederen bezig heeft gehouden met betrekking tot de LRM 1076-1993, dan is dat wel de 'globale variabele' (shared variable).

```

architecture ..
  signal global : resolved_integer;
begin
  prc1 : process (a)
  begin
    global <= global +1;
    outp1 <= global;
  end process;

  prc2 : process (a)
  begin
    outp2 <= global;
  end process;
end ..

```

figuur 4.1 Dankzij signalen als communicatie medium is de volgorde van de executie processen niet belangrijk.

In LRM 1076-1987 kunnen alleen signalen worden gebruikt als communicatie medium tussen verschillende processen. Dankzij het delta-delay-mechanisme voor signaal-toekenningen is het onbelangrijk welk proces door de simulator het eerste wordt geëxecuteerd tijdens de simulatie van figuur 4.1. Echter wordt signaal GLOBAL vervangen door een variabele, wat niet is toegestaan in de LRM 1076-1987, dan is het gedrag afhankelijk van de volgorde van de executie van de processen, figuur 4.2. waardoor de beschrijving niet meer overdraagbaar is naar andere omgevingen.

```

architecture ..
  shared variable global : resolved_integer;
begin
  prc1 : process (a)
  begin
    global := global +1;
    outp1 <= global;
  end process;

  prc2 : process (a)
  begin
    outp2 <= global;
  end process;
end ..

```

figuur 4.2 Globale variabelen zijn funest als het gaat om de invloed van de executie volgorde van processen (niet correct volgens de LRM 1076-1987).

Veronderstel dat voor de verandering van signaal A de variabele GLOBAL de waarde 100 heeft, dan zijn de volgende twee simulaties mogelijk:

- Eerst wordt proces PRC1 uitgevoerd. D.w.z. dat de globale variabele nog tijdens dezelfde iteratie de waarde 101 krijgt. Vervolgens zal het signaal OUTP1 na een delta-delay ook 101 worden. Vervolgens wordt proces PRC2 geëxecuteerd, de globale variabele heeft inmiddels al de waarde 101 gekregen, en dus wordt ook aan het signaal OUTP2 na een delta-delay de waarde 101 toegekend.
Dus na een delta-delay: OUTP1 101 en OUTP2 101.
- Eerst wordt proces PRC2 uitgevoerd. De globale variabele heeft de waarde 100, en dus zal na een delta-delay het signaal OUTP2 de waarde 100 krijgen. Vervolgens wordt proces PRC1 geëxecuteerd. De globale variabele zal nog tijdens dezelfde iteratie de waarde 101 krijgen. Vervolgens zal het signaal OUTP1 na een delta-delay ook 101 worden.
Dus na een delta-delay geldt: OUTP1 is 101 en OUTP2 is 100.

Uiteraard is dit een ongewenste situatie. Echter veel gebruikers hebben te kennen gegeven dat zij toch een vorm van globale variabelen willen hebben (er zijn ook vele tegenstanders). De wijze waarop dit in de taal verwerkt moest worden gaf aanleiding

tot veel discussies. Zo zou het probleem geschetst in figuur 4.2 niet voor mogen komen. Dus is er een vorm van protectie nodig.

De beschrijving in figuur 4.2 is correct volgens de LRM 1076-1993. Maar tevens is in het voorjaar van 1993 door de ISAC (Issues Screening and Analysis Committee), een subgroep van VASG (VHDL Analysis and Standardization Group), een commissie "shared variable working group" ingesteld die met een passende oplossing moet komen.

4.3.1 Generatie van random getallen.

Om aan te geven dat een vorm van globale variabelen wenselijk kan zijn wordt als voorbeeld gegeven het genereren van random getallen. Voor het genereren van random getallen is het wenselijk dat, elke keer als er een random getal wordt gevraagd, er een nieuw getal gegenereerd wordt. In figuur 4.3 is aangegeven hoe dat in LRM 1076-1987 beschreven kan worden.

```
package random is
  signal current_state : integer;
  procedure random_nmb (new_nmb : out integer; signal state : inout integer);
end random;

package body random is
  procedure random_nmb (new_nmb : out integer; signal state : inout integer) is
  begin
    ..... -- a random generator algorithm!
  end;
end random;

use work.random.all;
entity use_random is
  port (a : integer; b : out integer);
end use_random;

architecture test of use_random is
begin
  process (a)
    variable tmp : integer;
  begin
    random_nmb(tmp,current_state);
    b <= tmp;
  end process;
end test;
```

figuur 4.3 Genereren van random getallen.

De procedure *random_nmb* genereert op basis van de huidige STATE een nieuw random getal. Echter merk op dat voor de huidige toestand een signaal noodzakelijk is. Dus als op hetzelfde tijdstip én iteratie nog een random getal wordt gevraagd dan zal dit hetzelfde getal zijn! Merk ook op dat de CURRENT_STATE niet gedeclareerd kan worden binnen de package body zodat deze buiten de package zichtbaar is.

Alleen een globale variabele kan deze probleem oplossen. De package random wordt dan (LRM 1076-1993):

```
package random is
  procedure random_nmb (new_nmb : out integer; state : inout integer);
end random;

package body random is
  shared variable current_state : integer;
  procedure random_nmb (new_nmb : out integer; state : inout integer) is
  begin
    ..... -- a random generator algorithm!
```

```

end;
end random;

```

4.4 Pure en impure functions.

In LRM 1076-1987 zijn bijna alle functies *pure*, d.w.z. dat zij altijd hetzelfde resultaat geven als de waarden van de parameters waarmee de functie wordt aangeroepen gelijk zijn. Er is één uitzondering te vinden in de package standard van de LRM 1076-1987, namelijk de functie *now*¹⁵. Deze functie heeft als resultaat het simulatie tijdstip!

In de LRM 1076-1993 kan een functie zowel *pure* als *impure* zijn. Een functie is *pure* als het voldoet aan hetgeen hiervoor is gesteld. Een functie is *impure* indien het resultaat niet hetzelfde hoeft te zijn.

```

subprogram_specification ::=
    procedure designator [ ( formal_parameter_list ) ]
    | [ pure | impure ] function designator
      [ ( formal_parameter_list ) ] return type_mark

```

Indien voor een functie geen *pure* of *impure* staat wordt *pure* verondersteld. Aangezien de functie *now* niet *pure* is, is deze in de LRM 1076-1993 voorzien van het gereserveerde woord *impure*.

In figuur 4.3 werd een procedure *random_nmb* gebruikt om een random getal te genereren. Hierbij moest ook de huidige *state* worden meegegeven. Door een impure functie te gebruiken hoeft ook deze state niet meer te worden opgegeven. Daarmee wordt de package random gelijk aan:

```

package random is
    impure function random_nmb return integer;
end random;

package body random is
    shared variable current_state : integer;
    impure function random_nmb return integer is
    begin
        ..... -- a random generator algorithm!
    end;
end random;

```

Het type *time* is zodanig gedefinieerd dat het ook negatief kan zijn. Dit is noodzakelijk in verband met het vergelijken van tijdstippen, waarvan het resultaat ook negatief kan zijn. Echter een simulatietijdstip kan niet negatief zijn. In de package standard van de LRM 1076-1993 is daarom een extra subtype *delay_length* opgenomen die gebruikt wordt in de functie *now*.

```

subtype delay_length is time range 0 fs to time'high;
impure function now return delay_length;

```

¹⁵ In de package *textio* van de LRM 1076-1987 komt ook nog de functie *endline* voor die niet 'pure' is. Echter deze functie is op 10 november 1988 door de ISAC uit de standaard verwijderd. ("The Sense of the VASG", October, 1989, VHDL Issue Number 0032.)

4.5 Group.

Het ook mogelijk om bij elkaar behorende zaken te groeperen in een *group*. Indien bijvoorbeeld de signalen SIG1 en SIG2 altijd dezelfde vertraging hebben, en datzelfde geldt ook voor de signalen SIG3, SIG4 en SIG5, dan kan dit worden beschreven door deze signalen eerst te groeperen.

Eerst is een *group_template_declaration*¹⁶ nodig:

Voor het voorbeeld van de signalen bijvoorbeeld:

group vertraging_gelijk **is** (**signal** <>)

Vervolgens volgt een group declaratie¹⁷

group groep1 : vertraging_gelijk (sig1,sig2);

group groep2 : vertraging_gelijk (sig3,sig4,sig5);

En vervolgens kan bijvoorbeeld een door de gebruiker gedefinieerd attribuut worden gebruikt om de vertragingstijd aan te geven:

attribute delay : time;

attribute delay **of** groep1 : group **is** 10 ns;

attribute delay **of** groep2 : group **is** 15 ns;¹⁸

4.6 Bit string literals.

In the LRM 1076-1987 kon door middel van een base_specifier (B, O, X) worden aangegeven of de string geïnterpreteerd moet worden als respectievelijk binaire, octale of hexadecimale representatie. In de stringrepresentatie mogen 'underlines' voorkomen en het resultaat is van het type bit_vector.

Dit laatste, het type van het resultaat, is gewijzigd in de LRM 1076-1993. Elke digit in de string (dus niet de underlines) wordt vervangen door een string van resp. 1, 3 of 4 lang (resp. binair, octaal en hexadecimaal). Het type van het resultaat is niet alleen beperkt voor bit_vectoren maar ook bruikbaar voor ander arrays, mits uiteraard deze de elementen '0' en '1' bevatten.

4.7 Component instantiatie.

In de LRM 1076-1987 kan alleen een entity worden gebruikt door eerst een component te declareren en vervolgens deze te instantiëren. Door middel van een configuratie-specificatie kan heel eenvoudig voor de component een andere entity worden gekozen.

¹⁶ group_template_declaration ::=
group identifier **is** (entity_class_entry_list);
entity_class_entry_list ::=
entity_class_entry { , entity_class_entry }
entity_class_entry ::= entity_class [<>]
entity_class
entity | **architecture** | **configuration** | **procedure** | **function** | **package** | **type** | **subtype**
| **constant** | **signal** | **variable** | **component** | **label** | **literal** | **units** | **group** | **file**

¹⁷ group_declaration ::=
group identifier : group_template_name (group_constituent_list);
group_constituent_list ::= group_constituent { , group_constituent }
group_constituent ::= name | character_literal

¹⁸ Dit suggereert dat nu ook de signalen van group GROEP2 het attribuut delay hebben. Dan zou het volgende zijn toegestaan: SIG3'DELAY; Dit blijkt echter niet in overeenstemming met de standaard te zijn.

In de LRM 1076-1993 zijn ook 'directe' instantiaties van entities en configuraties mogelijk¹⁹. In een architecture-beschrijving kan dus ook het volgende voorkomen:

```
architecture ..
    component demo ..
begin
    inst : component demo ..
    insta : entity work.xor(behaviour) ..
    instb : configuration test ..
..
```

Het gebruik van het gereserveerde woord *component* is optioneel i.v.m. de compatibiliteit met de LRM 1076-1987.

4.8 Expressies in de association list in de port map.

De regels met betrekking tot de association list in de port map zijn gewijzigd. In de vorige versie van de standaard mochten alleen signalen worden gebruikt voor de actuele parameters. In de nieuwe standaard mogen voor de mode zowel signalen als expressies worden gebruikt.

Dit betekent bijvoorbeeld dat een component waarvan een ingang bijvoorbeeld altijd '1' geen hulp signaal nodig is. Nu is het volgende toegestaan:

```
instantiatie : component_naam port map ( '1', -- altijd '1', voorheen niet toegestaan
.... );
```

4.9 Configuration.

In de LRM 1076-1987 kan een configuratie niet worden overruled door een andere configuratie. Dit is wel mogelijk in de LRM 1076-1993.

Het volgende fragment is volgens de LRM 1076-1993 wel correct en niet volgens de LRM 1076-1987:

```
architecture test of conf is
    component demo ..
    for inst : demo use work.demo(gedrag);
begin
    inst : component demo ..
..
end test;

configuration overruling of conf is
    for test
        for inst : demo use lib.demo(structuur);
```

¹⁹ component_instantiation_statement ::=
 instantiation_label :
 instantiated_unit
 [generic_map_aspect]
 [port_map_aspect] ;
 instantiated_unit ::=
 [component] component_name
 | entity entity_name [(architecture_identifier)]
 | configuration configuration_name

end for;
end overruling;

In de configuratie-specificatie wordt opnieuw een configuratie gegeven voor de instantiatie *inst*.

4.10 Nieuwe operatoren.

In het onderstaande overzicht zijn de nieuwe operatoren onderstreept. De volgorde waarin de regels zijn gegeven geeft ook de volgorde van executie aan.

logical_operator	::=	and		or		nand		nor		xor		<u>xnor</u>
relational_operator	::=	=		/=		<		<=		>		>=
shift_operator	::=	sll		srl		sla		sra		rol		<u>ror</u>
adding_operator	::=	+		−		&						
sign	::=	+		−								
multiplying_operator	::=	*		/		mod		rem				
miscellaneous_operator	::=	**		abs		not						

4.10.1 Xnor operator.

De *exclusive nor* operator ontbrak in de LRM 1076-1987.

4.10.2 Schuif operatoren.

In een digitaal systeem moet vaak met data worden geschoven. Echter in de LRM 1076-1987 zijn standaard geen schuifoperaties aanwezig. Hoewel deze eenvoudig te beschrijven zijn in deze standaard zijn schuifoperaties als infix-operatoren toegevoegd aan de LRM 1076-1993 voor een-dimensionale arrays met elementen van het type bit en boolean.

Operator		type linker operand	type rechter operand	type resultaat
sll	shift left logical	eendimensionaal array met elementen van het type bit of boolean	integer	zelfde type als linker operand
srl	shift right logical	idem	idem	idem
sla	shift left arithmetic	idem	idem	idem
sra	shift right arithmetic	idem	idem	idem
rol	rotate left logical	idem	idem	idem
ror	rotate right logical	idem	idem	idem

Er van uitgaande dat L de linker operand is (de te verschuiven vector) en R de rechter operand (het aantal te verschuiven posities) zijn de schuif operaties als volgt gedefinieerd (indien L een null array is het resultaat altijd L) :

sll	$R \geq 0$	De vector L wordt R maal één positie naar links verschoven waarbij op de rechter positie steeds T' ^{left20} wordt ingeschoven.
	$R < 0$	Het resultaat is: L srl -R

²⁰ "T" is het type van de elementen van de vector: boolean of bit.

srl	$R \geq 0$	De vector L wordt R maal één positie naar rechts verschoven waarbij op de linker positie steeds T'left wordt ingeschoven.
	$R < 0$	Het resultaat is: L srl -R
sla	$R \geq 0$	L wordt R maal één positie naar links verschoven waarbij op de rechter positie wordt L'(L'right) ingeschoven.
	$R < 0$	Het resultaat is: L sla -R
sra	$R \geq 0$	L wordt R maal één positie naar rechts verschoven waarbij op de rechter positie L(L'left) wordt ingeschoven.
	$R < 0$	Het resultaat is: L sla -R
ror	$R \geq 0$	L wordt R maal één positie naar rechts verschoven waarbij op de rechter positie L(L'right) wordt ingeschoven.
	$R < 0$	Het resultaat is: L rol -R
rol	$R \geq 0$	L wordt R maal één positie naar links verschoven waarbij op de rechter positie L(L'left) wordt ingeschoven.
	$R < 0$	Het resultaat is: L ror -R

4.11 Labels.

In de LRM 1076-1987 is aangegeven dat sommige concurrent statements gelabeld moeten worden, zoals het block statement, en andere concurrent statements gelabeld mogen worden, zoals het proces statement.

In de LRM 1076-1993 mogen ook alle sequentiële statements gelabeld worden.

4.12 Report statement.

In de LRM 1076-1987 is het mogelijk om door middel van assert statements condities te controleren, bijvoorbeeld:

assert a > b **report** "a is kleiner of gelijk aan b" **severity** error;

De praktijk heeft laten zien dat heel vaak de volgende toepassing voorkomt:

assert false **report** "passende melding" **severity** error;

De conditie is altijd false en dus wordt de melding altijd afgedrukt. In de LRM 1076-1993 is een report statement opgenomen welke identiek is aan de *assert false ..* statement, namelijk:

report "passende melding" **severity** error;

4.13 Uitbreiding van het delay mechanisme.

De LRM 1076-1987 ondersteund een drietal delay modellen:

- Delta-delay
- Inertial-delay
- Transport-delay

Globaal geeft een inertial-delay alleen de veranderingen door die groter zijn dan de opgegeven tijd, terwijl transport elke verandering doorgeeft. (Zie hoofdstuk 3 voor een meer nauwkeurige beschrijving.)

inrt <= x **after** 4 ns;

trns <= **transport** inrt **after** 6 ns;

In de LRM 1076-1993 mag ook het nieuwe gereserveerde woord *inertial* worden gebruikt om expliciet aan te geven dat er sprake is van inertial-delay. Maar belangrijker is nog dat een delay opgegeven kan worden welke tussen dat van inertial en transport in ligt²¹.

```
rej <= reject 4 ns inertial x after 10 ns;
```

In deze beschrijving wordt een verandering die langer duurt dan 4 ns doorgegeven, maar met een vertraging van 10 ns. Dit statement heeft hetzelfde gedrag als de twee *concurrent* statements hiervoor beschreven.

4.14 Postponed process.

In de LRM 1076-1993 is het ook mogelijk de executie van een proces uit te stellen tot de laatste iteratie op dat tijdstip. In het volgende fragment wordt duidelijk waarvoor dit gebruikt kan worden:

```
architecture ..
```

```
..
```

```
begin
```

```
    y1 <= x after 10 ns;
```

```
    y2 <= y1;
```

Zijn de signalen Y1 en Y2 gelijk? Strikt genomen niet, immers door het delta-delay mechanisme krijgt signaal y2 één delta-delay later de nieuwe waarde dan signaal Y1. Dit blijkt ook als de volgende statement in de beschrijving aanwezig is:

```
    gelijk1 <= y1 = y2 after 10 ns;
```

Er zijn echter ook toepassingen waarbij alleen de stabiele situatie aan het einde van de iteraties op hetzelfde tijdstip voor het model interessant is. En dit is mogelijk door de executie van dit laatste proces uit te stellen tot het einde van alle iteraties op een tijdstip.

```
    gelijk2 <= postponed y1 = y2 after 10 ns;
```

Als extra voorwaarde voor een *postponed* proces geldt dat het geen extra event mag genereren voor de volgende delta-delay (volgende iteratie). Dus de volgende statements zijn niet juist:

```
    gelijk2 <= postponed y1 = y2; -- niet correct!
```

```
    gelijk2 <= postponed y1 = y2 after 0 ns; -- niet correct!
```

4.15 Concurrent conditional signal assignment statement.

In de LRM 1076-1987 moest elk conditional assignment statement eindigen met een *waveform*. In de LRM 1076-1993 is dit optioneel:

```
conditional_signal_assignment ::=
    target <= options conditional_waveforms ;
```

```
conditional_waveforms ::=
    { waveform when condition else }
    waveform [ when condition ]
```

²¹ signal_assignment_statement ::=
 [label :] target <= [delay_mechanism] waveform ;
 delay_mechanism ::=
 transport
 | [**reject** time_expression] **inertial**

```

y1 <= x when a='1' else y1;    -- correct LRM 1076-1987, 1076-1993.
y2 <= x when a='1';             -- correct LRM 1076-1993

```

Hoewel beide statements equivalent lijken te zijn zijn ze dit niet! Indien het signaal A niet gelijk is aan '1' wordt aan de waveform van het signaal Y1 een transaction toegevoegd met dezelfde waarde en blijft de waveform van het signaal Y2 onveranderd. Attributen die gevoelig zijn voor een transactie, wel of niet met dezelfde waarde, zullen vervolgens een ander resultaat opleveren. Zo geldt in het algemeen dat impliciete signaal Y1 **stable** niet gelijk is aan het impliciete signaal Y2 **stable**.

4.16 Unaffected.

In een proces is het mogelijk een toekenning **null** te doen aan een guarded resolved signal. Bijvoorbeeld:

```

process
begin
    if a='1'
        then y <= '1';
        else y <= null;
    end if;
    . . . .
end process;

```

Echter in een concurrent signal assignment statement is een null-waveform-element niet toegestaan daarom werd dan de volgende statement gebruikt:.

```

y <= (not y) when a='1' else y;

```

Merk op dat dit statement niet gelijk is aan de process beschrijving. Dankzij het gereserveerde unaffected kan een identiek gedrag worden verkregen:

```

y <= (not y) when a='1' else unaffected;

```

Is er een verschil tussen de volgende twee statements?

- a) q <= i **when** a='1' **else** **unaffected**;
- b) q <= i **when** a='1';

Indien het signaal q geen guarded signal is, is er geen verschil. Indien q een guarded signal is wordt in situatie a) een null-transitie toegekend aan q indien a ongelijk aan '1' is en in situatie b) blijft de waarde van de driver voor q onveranderd.

4.17 Generate statement.

In het generate statement van de LRM 1076-1993 kunnen ook declaraties worden opgenomen. Hiermee is het mogelijk om signalen die alleen binnen het generate statement nodig zijn ook alleen daar te declareren.

De syntax voor het generate statement is:


```

generate_statement ::=
  generate_label :
    generation_scheme generate
      [ { block_declarative_item }
      begin ]
      { concurrent_statement }
    end generate [ generate_label ] ;

```

4.18 Karakterset.

De karakterset is veranderd van ASCII (128 karakters) naar ISO 8859-1:1087(E) met 256 karakters.

4.19 Extended identifiers.

Gereserveerde woorden maken integratie met andere omgevingen, zoals bijvoorbeeld synthese-tools, soms moeilijk. Een synthese-tool heeft bijvoorbeeld een aantal componenten zoals inverter, buffer, adder, etc. Indien de buffer gebruikt moet worden ziet de structuurbeschrijving er als volgt uit:

```

architecture
  component buffer ..
begin
  buf : component buffer..
  ..
end ..

```

Buffer is echter een gereserveerd woord en hier dus niet toegestaan. Door middel van een backslash is het nu mogelijk om ook gereserveerde woorden te gebruiken. Correct volgens de LRM 1076-1993 is:

```

architecture
  component \buffer\ ..
begin
  buf : component \buffer\..
  ..
end ..

```

Nu geldt ook dat:

- \BUS\ niet gelijk is aan \Bus\.
- \dit is een identifier\ is één identifier.
- een backslash is \.

4.20 Alias.

In de LRM 1076-1987 kan een alias alleen worden gebruikt voor objecten. In de LRM 1076-1993 kan ook voor een niet-object een alias worden gebruikt. Voor een functie of procedure dient daarbij ook de signature worden opgegeven.

Een alias kan niet gebruikt worden voor labels, loop parameters en generate parameters.

Let er op dat de eerste “[“ en de laatste “]” in de signature letterlijk aanwezig moet zijn in een VHDL beschrijving en niet ‘optioneel’ betekent.

```

alias_declaration ::
  alias alias_designator [ : subtype_indication ] is name [ signature ] ;

```

```
alias_designator ::= identifier | character_literal | operator_symbol
```

```
signature ::= [ [ type_mark { , type_mark } ] [ return type_mark ] ]
```

Voorbeeld:

```
package demo is
    function fun (a: bit) return bit;
end demo;

entity x is
end x;

architecture b of x is
    signal r, s : bit;
    alias f is work.demo.fun [ bit return bit ];
begin
    s <= f (r);
end b;
```

4.21 File.

In LRM 1076-1987 kon zeer beperkt met files worden gewerkt. Zo kon nooit expliciet een file worden geopend of gesloten, dit gebeurt resp. met het starten en beëindigen van een simulatie. Gevolg hiervan is dat gegevens die naar een file worden geschreven pas aan het einde van de simulatie beschikbaar is. Ook werd 'file' niet als een object gezien. Hierdoor was het ook niet mogelijk een procedure te schrijven met een file als object: bijvoorbeeld procedure naam (file f : ...).

In LRM 1076-1993 is file ook een object (naast de variable, signaal en constante). Verder zijn er subprograms aan de taal toegevoegd waarmee het openen en sluiten van een file tijdens de simulatie kan plaatsvinden. Om dit mogelijk te maken zijn de volgende type-declaraties aan de package standard toegevoegd:

```
type file_open_kind is (
    read_mode,           -- Alleen lezen is mogelijk.
    write_mode,          -- Alleen schrijven is mogelijk.
    append_mode);        -- Alleen schrijven is mogelijk en het resultaat
                        -- wordt aan de file toegevoegd.

type file_open_status is (
    open_ok,             -- File met succes geopend.
    status_error,        -- File was al geopend.
    name_error,          -- File niet gevonden of niet toegankelijk.
    mode_error);         -- File met opgegeven mode kan niet
                        -- worden geopend.
```

Tevens zijn ook de volgende procedures impliciet aanwezig voor een file type FT na de volgende declaratie: **type** ft **is** **file of** tm; waarin TM een scalair type is, een record, of een constrained array.):

```
procedure file_open (
    file f:                ft;
    external_name:        in string;
```

```

        open_kind:          in file_open_kind := read_mode);
procedure file_open (
    status:                out file_open_status;
    file f:                ft;
    external_name:         in string;
    open_kind:             in file_open_kind := read_mode);
procedure file_close (file f: ft);
procedure read (file f: ft; value: out tm);
procedure write (file f: ft; value: in tm);
impure function endfile (file f : ft) return boolean;22

```

Enkele voorbeelden (met **type** int_file **is** file of integer;):

- **file** f1 : integer_file; -- geen impliciete 'file_open'.
- **file** f2 : integer_file **is** "test.dat"; -- impliciet openen met leesmode
- **file** f3 : integer_file **open** write_mode **is** "test.dat";
-- impliciet openen met schrijfmode

Concreet betekent dit dat beschrijvingen welke gebruik van maken van een file gebaseerd op de vorige versie van de standaard (LRM'87) niet upwards compatible zijn.

De volgende wijzigingen worden gemaakt:

file f3 : int_file is out "my_fyle";	LRM 1076-1987
file f3 : int_file open write_mode is "my_fyle";	LRM 1076-1993
file vec : text is in "test.vec";	LRM 1076-1987
file vec : text open read_mode is "test.vec";	LRM 1076-1993
Voeg het gereserveerd woord impure toe aan alle functions	LRM 1076-1993

4.22 Attributen.

4.22.1 Attributen verwijderd.

In de LRM 1076-1987 komen de attributen 'structure en 'behavior voor. Beide attributen komen niet meer voor in LRM 1076-1993

4.22.2 Attributen toegevoegd.

4.22.2.1 Foreign.

Soms is het wenselijk dat de architecture niet in VHDL is geschreven maar in een andere taal om:

- de simulatie tijd te verkorten, of
- de implementatie te verbergen voor de gebruiker van bijvoorbeeld commerciële modellen.

In de package standard is het volgende attribuut gedeclareerd:

²² Op de LRM 1076-1993 is inmiddels door de VASG (VHDL Analysis and Standardization Group) van de IEEE een correctie aangebracht. In de huidige tekst van de LRM ontbreekt het gereserveerde woord **impure**.

attribute foreign : string;

Dit attribuut mag alleen worden geassocieerd met een architecture of de body van een subprogram. In het laatste geval moet de attribuutspecificatie staan in het declaratiedeel waarin het subprogram is gedeclareerd.

Op welke wijze dit verder is geïmplementeerd is niet in de standaard beschreven. Dit kan dus ook portabiliteitsproblemen veroorzaken.

4.22.2.2 Ascending.

Dit attribuut e'ascending[(n)] geeft het boolean resultaat die true is als de gegeven dimensie (indien aanwezig) stijgend is.

4.22.2.3 Image en value.

Een image-attribuut levert een stringrepresentatie op van de parameter met de volgende eigenschappen:

- In het geval van een 'extended identifier' wordt een backslash om de string geplaatst. Hoofdletters en kleine letters blijven zoals ze zijn. Voor de overige identifiers worden kleine letters genomen.
- Een character literal wordt omgeven met de '.
- Infix-operatoren worden omgeven met de ".
- Voor een exponent wordt de kleine letter "e" gebruikt.
- Voor een fysiek (sub)type wordt het resultaat uitgedrukt in de primary unit. Voor het type tijd wordt echter het resultaat uitgedrukt in de 'resolution limit'. (een simulator hoeft niet als kleinste tijd 1 fs te accepteren, dit mag bijvoorbeeld ook 1 ns zijn).

Het value attribuut kan een string weer omzetten in het daarbij behorende type.

4.22.2.4 Driving en driving_value.

Het attribuut driving is false indien aan de driver een null toekenning is gedaan. Het attribuut driving_value geeft de waarde van de driver. Van een signaal mag alleen de driving_value worden gevraagd indien er geen null-toekenning is gedaan. dus bijvoorbeeld in de volgende constructie:

```
if sig'driving
  then res := s'driving_value;
  else null; -- hier mag s'driving_value niet worden gebruikt!
end if;
```

4.22.2.5 Simple_name, instance_name en path_name.

Deze attributen geven een stringrepresentatie van een object, subprogram etc. Bij simple_name wordt geen pad meegegeven. Bij instance_name en path_name wel. Bij instance_name wordt ook de geïnstatieerde design entity meegegeven, dit ontbreekt bij de path_name. De ':' wordt gebruikt als scheidingskarakter. Ook worden weer in de opgeleverde string kleine letters gebruikt (tenzij er sprake is van een extended identifier).

Een voorbeeld verduidelijkt het een en ander. Verondersteld is dat de package en de entity zich in library lib bevinden.

```
package p is
    -- p'path_name = ":lib:p"
    -- p'instance_name = ":lib:p"
```

```

procedure proc (f: inout integer);
    constant c: integer := 42;
end p;

package body p is
    procedure proc (f: inout integer) is
        variable x: integer;
    ...
end;

library lib;
use lib.p.all;
entity e is
    end e;

architecture a of e is
    signal s: bit_vector (1 to g);
    ..
end;

```

-- p'simple_name = "p"
-- proc'path_name = ":lib:p:proc"
-- proc'instance_name = ":lib:p:proc"
-- proc'simple_name = "proc"
-- c'path_name = ":lib:p:c"
-- c'instance_name = ":lib:p:c"
-- c'simple_name = "c"

-- x'path_name = ":lib:p:proc:x"
-- x'instance_name = ":lib:p:proc:x"

-- entity e is in in library lib en
-- is the top-level design entity:
-- e'path_name = ":e:"

-- s'path_name = ":e:s"
-- s'instance_name = ":e(a):s"

5. Van een VHDL beschrijving naar een realisatie

Het ontwerpen van digitale systemen met behulp van VHDL is mogelijk van de specificatie tot en met een implementatie op logisch niveau. De realisatie zelf wordt in het algemeen niet in VHDL beschreven. Hiervoor waren ten tijde van de ontwikkeling van VHDL in 1980 reeds voldoende hulpmiddelen aanwezig. Na een aantal ontwerpstappen uitgevoerd te hebben in VHDL blijft er een beschrijving over die door een commerciële synthese-tool gerealiseerd kan worden. De VHDL beschrijving wordt omgezet in een realisatie (full custom, gate array, programmeerbare logica, etc.). De commerciële synthese-tools zijn grofweg te verdelen in twee groepen:

- logische synthese
- hoog niveau synthese

Bij een synthese systeem dat uitgaat van logische synthese wordt vaak een één op één afbeelding van VHDL naar een technologie uitgevoerd met een beperkt aantal combinatorische minimalisaties. Dit betekent dat de VHDL beschrijving al zoveel detail moet bevatten dat voor het synthetiseren al in grote lijn bekend is hoeveel registers e.d. gebruikt gaan worden. Aan de hand van een dergelijk synthese systeem zal in dit hoofdstuk synthese worden toegelicht.

Synthese systemen die hoog niveau synthese uitvoeren zijn in staat ook een aantal ontwerpbeslissingen op een hoger abstraktie niveau mee te nemen. Zo wordt bijvoorbeeld nagegaan of meerdere optellers zoals die beschreven zijn in de VHDL beschrijving niet gecombineerd kunnen worden, uiteraard zonder het gewenste gedrag te veranderen. Ook kunnen beschrijvingen waarin de toestanden impliciet aanwezig zijn gerealiseerd worden.

Vaak kan een ontwerp ingedeeld worden in een data-pad en een besturing voor dit data-pad. De besturing kan vaak beschreven worden in de vorm van een toestandsmachine, terwijl in een data-pad vaak weer componenten als optellers, multiplexers e.d. voorkomen. Er zijn dan ook synthese systemen die verschillende algoritmen gebruiken voor het synthetiseren van een data-pad en een besturing.

Het is niet de intentie van dit hoofdstuk om alle synthese aspecten of de algoritmen die hierbij worden gebruikt te behandelen. Het is bedoeld om de lezer een indruk te geven van de mogelijkheden en onmogelijkheden van synthese systemen zodat duidelijk wordt op welk niveau gestopt kan worden bij het beschrijven in VHDL, de voorbeelden zijn uitgewerkt met een logische synthese-tool. Inmiddels zal het duidelijk zijn dat er veel alternatieve VHDL beschrijvingen mogelijk zijn om hetzelfde gedrag te beschrijven. Synthese systemen ondersteunen echter slechts een subset van de mogelijkheden van VHDL en, helaas, niet dezelfde subset.

Lees dit hoofdstuk met de nodige terughoudendheid; morgen is synthese van VHDL beschrijvingen mogelijk waarvan nu nog alleen kan worden gedroomd.

Verder wordt in dit hoofdstuk ook met name stil gestaan bij de gevaren bij het gedachteloos gebruiken van synthese-tools omdat simulatie en synthese wel een verschillende resultaten kunnen opleveren.

De voorbeelden uitgewerkt in dit hoofdstuk zijn gesynthetiseerd met de ViewSyn 5.0.1, ViewLogic. De gebruikte technologie is Xilinx X4000.

5.1 Wat is niet synthetiseerbaar?

In veel hardware beschrijvingstalen zijn alle constructies af te beelden op hardware. Het is de intentie van VHDL dat ook een specificatie²³ moet kunnen worden beschreven. Taalconstructies die in een dergelijke beschrijving worden gebruikt hoeven niet synthetiseerbaar te zijn. Nagenoeg alle synthese systemen hebben geheel of gedeeltelijk problemen met:

- expliciet opgegeven vertragingstijden
- asynchrone beschrijvingen
- wait statements
- dynamische structuren, recursie, loops en files
- typering voor signalen
- het event-driven simulatiemodel

Het volgende hoofdstuk geeft een groot aantal synthese-resultaten voor complexere voorbeelden dan de voorbeelden die in dit hoofdstuk voor educatieve doeleinden zijn opgenomen.

5.1.1 Dynamische structuren, recursie en loops.

Voordat een VHDL beschrijving gesimuleerd kan worden wordt deze eerst geanalyseerd, vindt de elaboration plaats en pas daarna kan de executie (simulatie) plaats vinden. De analyse en de elaboration zijn voor synthese en simulatie nagenoeg gelijk. Echter daarna is er een groot verschil, tijdens de simulatie kan nog willekeurig ruimte gecreëerd worden (access types) en kan een for loop willekeurige grenzen hebben. Maar voor synthese moet na de elaboration bekend zijn welke hardware nodig is!

Dit betekent dat dynamische structuren (access types) niet worden ondersteund. Hoewel, met reconfigureerbare FPGA's zou men een heel eind kunnen komen!

Voor recursieve aanroepen van subprogramma's (functies en procedures) geldt ongeveer hetzelfde. Indien na de elaboration geen bovengrens is vast te stellen zal ook dit niet te synthetiseren zijn.

Ook een loop statement kent soortgelijke problemen. De volgende loop statement is voor simulatie geen enkel probleem, maar voor synthese ...

```
for i in 0 to input_signaal loop
```

```
....
```

```
end loop;
```

In deze loop wordt de bovengrens bepaald door een ingangssignaal. Voor de aanvang van de simulatie kan niet worden vastgesteld hoe vaak deze loop 'uitgevouwen' moet worden. Dit is dus ook voor een synthese tool niet eenvoudig, immers hoe moet dit worden gerealiseerd? Mogelijk dat een synthese tool in staat is om een (pessimistische) bovengrens te bepalen, bijvoorbeeld door gebruik te maken van de typering en op basis hiervan hardware te genereren. Door uit te gaan van de bovengrens, als dat al wordt ondersteund, is de kans op te veel hardware zeer waarschijnlijk.

Ook de manier waarmee met de loop-variabele wordt omgegaan kan de synthese-resultaten aanzienlijk beïnvloeden. Bijvoorbeeld:

```
for loop_var in 1 to 15 loop
```

²³ Merk op dat het 'simuleerbare specificaties' zijn. Dit betekent ook dat lang niet alles op een eenvoudige wijze met VHDL te specificeren valt.

```

    if loop_var > a+1 then .... end if;
end loop;

```

De conditie in het if-statement zal waarschijnlijk resulteren in een comparator en een opteller. Hetzelfde gedrag kan ook als volgt worden beschreven:

```

for loop_var in 1 to 15 loop
    if loop_var-1 > a then .... end if;
end loop;

```

Afhankelijk van de synthese tool zal nu waarschijnlijk geen aftrekker worden gebruikt. Immers bedenkt dat bij het uitvouwen van de loop door de synthese tool *loop_var* een constante is en dus ook *loop_var - 1*.

Zoals al eerder is opgemerkt is dit sterk afhankelijk van de synthese tool. Het is dan ook aan te raden de richtlijnen, suggesties en dergelijke nauwkeurig te lezen. Ook bij het verschijnen van een nieuwe versie!

5.1.2 Expliciet opgegeven vertragingstijden

VHDL kent een drietal delay modellen, delta, inertial en transport-delay²⁴. Synthese systemen maken (nog) geen onderscheid tussen deze delay modellen. Elke opgegeven vertraging wordt terug gebracht tot een delta-delay. M.a.w de constructie *y <= a and b after 10 ns;* wordt tijdens synthese teruggebracht tot *y <= a and b;*. Dit betekent tevens dat de simulatieresultaten afwijken van het gedrag van de realisatie.

```

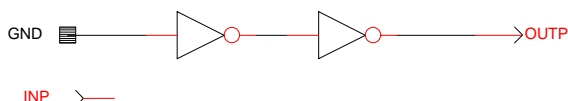
entity invloed_delay is
  port (inp : in bit;
        outp : out bit);
end invloed_delay;

```

```

architecture voorbeeld of invloed_delay is
  signal y_vroeg,y_laag : bit;
begin
  y_vroeg <= inp after 10 ns;
  y_laag <= inp after 20 ns;
  outp <= y_vroeg xor y_laag;
end voorbeeld;

```



figuur 5.1 Gerealiseerde wijkt af van de simulatie.

Figuur 5.1 beschrijft een schakeling welke signaleert dat het ingangssignaal INP verandert. Indien deze schakeling gesynthetiseerd gaat worden zullen alle (?) synthese systemen tot de conclusie komen dat het uitgangssignaal outp altijd '0' blijft! De twee inverters in deze beschrijving zijn ontstaan ten gevolge van een eigenschap van de gebruikte tool: elke in- en uitgang is voorzien van een buffer.

Indien de opgegeven vertragingstijden essentieel zijn voor de werking van het circuit kan beter geen gebruik worden gemaakt van synthese systemen.

²⁴ In de Std 1076-1993 is ook nog een tussenvorm tussen inertial- en transport-delay opgenomen: 'reject'.

5.1.3 Asynchrone schakelingen

Figuur 5.2 geeft een beschrijving van een schakeling waarbij alleen een verandering van ingangssignaal INP1 een nieuw uitgangssignaal tot gevolg kan hebben. Merk op dat er dus geen *and* poort is beschreven. Synthese van een dergelijke beschrijving zal door veel systemen niet worden ondersteund, of er wordt (hopelijk) een waarschuwing gegeven dat er terugkoppeling aanwezig is in het circuit. In dit laatste geval zal de ontwerper zelf moeten nagaan of het gerealiseerde correct is. Indien helemaal geen waarschuwing wordt gegeven is de kans toch vrij groot dat de realisatie afwijkt van het gewenste gedrag. Indien een ontwerp een asynchrone schakeling beschrijft is de realisatie zelden 'correct by construction'. Trouwens hoe zou u dit realiseren? In een industriële omgeving wordt ook zelden met asynchrone schakelingen gewerkt. Voor synthese van synchrone schakelingen geldt vaak wel dat de realisatie 'correct by construction' is. 'Vaak' omdat vrijwel geen enkel systeem vrij is van fouten. En soms interpreteert de synthese-tool de beschrijving anders dan de simulator (pagina 112). Dit betekent dat het gerealiseerde altijd geverifieerd moet worden tegen het gewenste gedrag.

```
entity asynchroon is
  port ( inpl, inp2 : in bit;
        outp : out bit);
end asynchroon;

architecture gedrag of asynchroon is
begin
  process(inpl)
  begin
    outp <= inpl and inp2;
  end process;
end gedrag;
```

figuur 5.2 Beschrijving van een asynchrone schakeling.

5.1.4 Wait statements

In VHDL kunnen processen impliciet of expliciet beschreven zijn. Synthese systemen kunnen impliciet beschreven processen vaak goed synthetiseren. In een expliciet beschreven proces kunnen een viertal wait statements voorkomen: *wait for*, *wait*, *wait on*, *wait until* (en een combinatie hiervan).

wait for 10 ns;

Een *wait for* statement stopt de executie van het proces gedurende de opgegeven tijdsduur. Dit is voor simulatie geen enkel probleem, echter voor synthese een groot probleem!

wait

Het *wait* statement geeft aan dat het proces nooit meer geëxecuteerd gaat worden. In een procesbeschrijving wordt dit o.a. gebruikt om de inhoud van een ROM te initialiseren met data die bijvoorbeeld in een file staat. Ook een dergelijk wait statement kan vaak niet gesynthetiseerd worden.

wait on a,b;

Het *wait on* statement wordt o.a. gebruikt om combinatorische schakelingen mee te beschrijven. Soms mag dit statement niet expliciet worden gebruikt in een VHDL beschrijving, maar is het impliciet aanwezig door de sensitivity list achter het keyword process (pagina 116).

wait until clk='1';

Een procesbeschrijving met een wait until statement wordt in het algemeen gesynthetiseerd als een synchroon circuit met als kloksignaal CLK.

5.1.5 Dynamische structuren, recursie en files

Voor het beschrijven van een specificatie kan het gebruik van dynamische structuren en recursieve functies soms handig zijn. Echter het zal duidelijk zijn dat beide taalconstructies niet gesynthetiseerd kunnen worden. Voor recursie geldt mogelijk een uitzondering als tijdens de analysefase en elaboratiefase het aantal recursieve aanroepen bepaald kan worden. Uiteraard kunnen files ook niet worden gerealiseerd in hardware.

5.1.6 Typering op "bit niveau"

In VHDL worden een drietal²⁵ objecten onderscheiden: constanten, variabelen en signalen. Het in de standaard beschreven type bit is een enumeratie van een '0' en een '1'. Dit is vaak te beperkt omdat bijvoorbeeld het nog *onbekend* zijn van signaal waarden niet te beschrijven is. Daarom is er dan ook een 'wildgroeï' ontstaan van andere typeringingen om het 'bit niveau' beter te kunnen beschrijven. Uiteindelijk heeft een werkgroep van de IEEE heeft een negenwaardige logica opgesteld, welke geaccepteerd is (IEEE Std. 1164-1193).

```
TYPE std_ulogic is (
    'U',    -- Unitialized
    'X',    -- Forcing 0 or 1
    '0',    -- Forcing 0
    '1',    -- Forcing 1
    'Z',    -- High Impedance
    'W',    -- Weak 0 or 1
    'L',    -- Weak 0 ( for ECL open emitter )
    'H',    -- Weak 1 ( for open Drain or Collector )
    '-'     -- Don't care
);
```

Deze typering verdient zeker geen schoonheidsprijs. Het onderscheid tussen de logische niveaus en de fysieke waarden is door elkaar gehaald. In feite is dit alleen goed te gebruiken indien als uitgangspunt de *positieve logica* wordt gebruikt, m.a.w. een logische '0' wordt op ongeveer 0 V afgebeeld etc.. Het zou beter geweest zijn als er onderscheid was gemaakt tussen de logische niveaus en de fysieke waarden, waardoor ook met negatieve en gemengde logica gewerkt kan worden. Bijvoorbeeld door het type te veranderen in *type std_ulogic is ('U','X','L','H','Z','WL','WH','-');*

In de std_logic_1164 package is ook nog een resolved signal opgenomen: *std_logic*. Dit type is uiteraard nodig om bidirectionele bussen te kunnen beschrijven. Echter, in de standaard is opgenomen dat *std_logic* het voorkeurstype is. Dit betekent ook dat packages die weer gebaseerd zijn op deze package soms alleen uitgaan van de std_logic en std_logic_vector. Een groot nadeel is dat in veel situaties een signaal maar één driver heeft en als dit signaal van het type std_logic is en per ongeluk vanuit meerdere processen een waarde krijgt dit pas tijdens (lang) simuleren wordt opgemerkt in plaats van bij de analysefase.

²⁵ In VHDL'93 is ook 'file' een object.

De negenwaardige logica wordt tegenwoordig door elke CAE tool ondersteund. Steeds minder vaak wordt gebruik gemaakt van een ‘eigen’ vierwaardige logica, bijvoorbeeld *type vlbit is ('X','0','1','Z');*, waarbij 'vlbit' staat voor ViewLogic's bit. Ook andere CAE systemen gebruik(t)en vaak een vierwaardige logica, met uiteraard een andere naam voor het type. De betekenis van deze typering is:

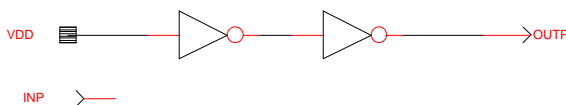
'X' niet geïnitieerd, '0' of '1', of don't care
 '0' logische '0'
 '1' logische '1'
 'Z' hoogohmige impedantie voor een tri-state

Elke omgeving heeft meestal een package waarin conversiefuncties zijn gedefinieerd tussen vectoren en integers etc. Daarvoor wordt nog geen unieke naamgeving gebruikt. Daar komt in het voorjaar van 1996 verandering in als de “Standard VHDL Synthesis Packages, Std. 1076.3-1996” (package *numeric_std* en package *numeric_bit*) door de IEEE wordt goedgekeurd. Deze standaard gaat alleen uit van het type *std_logic* (en ondersteunt niet het type *std_ulogic*). Tot die tijd blijft de defacto standaard de packages van SYNOPSYS, de marktleider in 1995.

```
library ieee;
use ieee.std_logic_1164.all;
entity verwarring is
  port (inp  : in std_ulogic;
        outp : out std_ulogic);
end verwarring;

architecture gedrag of verwarring is
  signal int_outp : std_ulogic;
begin
  process(inp,int_outp)
  begin
    if inp='X'
      then int_outp <='1';
    else int_outp <= not int_outp;
    end if;
  end process;
  outp <= int_outp;
end gedrag;
```

ns	inp	outp
0	X	X
0	X	1
100	1	1
100	1	0
200	0	0
200	0	1



figuur 5.3 Simulatie en synthese verschillen.

Met name de 'X' is nogal verwarrend in deze context. Als tijdens simulatie van *verwarring(gedrag)* (figuur 5.3) het signaal INP de waarde 'X' heeft wordt het uitgangssignaal '1', in alle overige situaties wordt het ingangssignaal geïnverteerd.

Maar wat is de betekenis van 'X'?

Tijdens het simuleren wordt meestal²⁶ geen bijzonderen betekenis aan het 'X' toegekend. Echter tijdens synthese wordt dit meestal beschouwd als een don't care. M.a.w. de procesbeschrijving uit figuur 5.3 wordt tijdens synthese als volgt geïnterpreteerd: *if "inp is een '0' of een '1' " then int_outp <= '1'; else ..* Maar het signaal INP is altijd een '0' of een '1' zodat het uitgangssignaal OUTP altijd '1' blijft!

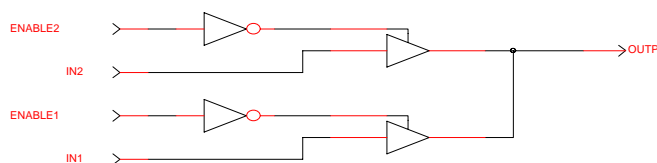
In een meer reële beschrijving zou bijvoorbeeld de volgende conditie kunnen voorkomen *if Y="0X1" then ..* Tijdens simulatie wordt dan gekeken of Y(0)='0' én Y(1)='X' én Y(2)='1', maar bij synthese wordt de volgende conditie gerealiseerd Y(0)='0' én Y(2)='1'.

Het 'logische' niveau 'Z'

Het niveau 'Z' wordt geïnterpreteerd als het hoogohmig zijn van een tri-state uitgang.

```
entity tri_state is
  port (in1, in2, enable1, enable2 : in std_ulogic;
        outp : out std_ulogic);
end tri_state;

architecture test of tri_state is
begin
  outp <= in1 when enable1='1' else
    'Z';
  outp <= in2 when enable2='1' else
    'Z';
end test;
```



figuur 5.4 Tri-state uitgangen.

In figuur 5.4 is aangegeven hoe een tweetal tri-state uitgangen met elkaar verbonden zijn. Inderdaad toont de realisatie het verwachte resultaat. Toch is dit niet zo vanzelfsprekend, het signaal OUTP heeft twee sources! Dit betekent dat er een resolutiefunctie aanwezig moet zijn. Een willekeurige resolutiefunctie zal niet gerealiseerd kunnen worden. Meestal ondersteunt een CAE systeem een aantal eigen resolutiefuncties en tevens de functies die zijn beschreven in de package STD_LOGIC_1164 van de IEEE.

²⁶ Indien in VHDL een nieuw type wordt gedeclareerd is er o.a. ook een impliciete functie voor de "=" aanwezig. Uiteraard geldt voor deze impliciete functie dat 'X' niet gelijk is aan '1'. Je kunt echter ook expliciet de functie "=" overladen. In deze laatste functie kan natuurlijk wel rekening worden gehouden met de betekenis don't care.

```

library ieee;
use ieee.std_logic_1164.all;
entity simulation_versus_realization is
  port (inp : in std_ulogic;
        outp : out std_ulogic);
end simulation_versus_realization;

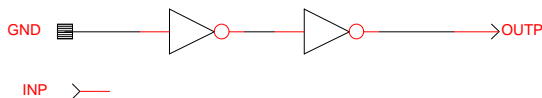
architecture test of simulation_versus_realization is
  signal y1,y2, sgnl : std_ulogic;
begin
  p1:process(inp,sgnl)
  begin
    sgnl <= inp;
    y1 <= sgnl;
  end process;

  p2:process(inp)
  variable var : std_ulogic;
  begin
    var := inp;
    y2 <= var;
  end process;

  outp <= y1 xor y2;
end test;

ns inp sgnl y1 y2 outp
0 X X X X X
10 0 X X X X
10 0 0 X 0 X
10 0 0 0 0 X
10 0 0 0 0 0
20 1 0 0 0 0
20 1 1 0 1 0
20 1 1 1 1 1
20 1 1 1 1 0

```



figuur 5.5 Uitgangssignaal gegenereerd ten gevolge van het event-driven simulatiemodel.

5.1.7 Het event-driven simulatiemodel

Figuur 5.5 geeft een beschrijving met twee expliciete processen. Proces P2 beschrijft niets anders dan een 'verbinding' tussen INP en Y2. Ten gevolge van het simulatiemodel wordt hetingangssignaal één delta-delay 'vertraagd'. Proces P1 beschrijft eveneens een verbinding tussen INP en Y1, echter het input signaal wordt tweemaal een delta-delay opgehouden. Na de eerste delta-delay krijgt signaal SGNL een nieuwe waarde, een verandering van dit signaal start het proces opnieuw zodat na de volgende delta-delay het signaal Y1 verandert. Dit betekent dus dat tijdens het simuleren de beide signalen Y1 en Y2 niet steeds gelijk aan elkaar zijn zodat gedurende een delta-delay het uitgangssignaal OUTP '1' is. Ook hier zal een synthese-tool het uitgangssignaal constant '0' maken.

5.1.8 Initiële waarden

Indien aan een variabele of een signaal geen initiële waarde is toegekend dan is in de VHDL standaard gedefinieerd dat de 'meest linkse waarde' van het bereik de initiële waarde is.

Voorbeelden:

integer: $(-2^{31})+1$, maar dit is tevens afhankelijk van de implementatie van VHDL

bit: '0'
 std_ulogic: '-'
 user defined enumeration: meest linkse element

Het is niet verstandig om in een beschrijving uit te gaan van deze impliciete initiële waarden. Beter is het om expliciet een initiële waarde aan een signaal of variabele toe te kennen indien hiervan gebruik wordt gemaakt. Dit maakt een beschrijving duidelijker en voorkomt tevens misverstanden. Bovendien hoeft de initiële waarde niet in elke omgeving gelijk te zijn! In de VHDL standaard is aangegeven dat het type integer *tenminste* het bereik moet hebben van $(-2^{31})+1$ tot $2^{31}-1$.

De gegeven initiële waarde van een integer is zelden vereist. Om dit te realiseren zijn 32 bits nodig. Synthese-tools gaan daarom vaak uit van de initiële waarde 0.

5.2 Wat is synthetiseerbaar

Welke VHDL constructies synthetiseerbaar zijn hangt sterk af van het synthese systeem. In dit hoofdstuk zal een indruk worden gegeven van wat synthetiseerbaar is aan de hand van de ViewSyn 5.0.1 van ViewLogic. Het is niet de intentie van dit hoofdstuk om alle facetten van deze synthese tool te behandelen.

De objecten (signalen, variabelen en constanten) mogen nagenoeg van elk type zijn, inclusief door de gebruiker gedefinieerde types. Het access type, record type, time en files worden niet ondersteund door de synthese tool, voor arrays gelden de nodige beperkingen.

5.2.1 Datatypes

Om een indruk te krijgen van de datatypes, en welke operaties op deze datatypes synthetiseerbaar zijn, volgt nu een overzicht van de verschillende datatypes uitgaande van de ViewSyn.

5.2.1.1 boolean.

Een boolean wordt afgebeeld op een logische '1' (true) of een logische '0' (false).

Synthetiseerbaar zijn de volgende operaties:

- De relationele operaties (=, /=, < etc.)
- De logische operaties (and, or etc.)
- De concatenatie

5.2.1.2 integer.

In de standaard is gedefinieerd dat een integer minimaal afgebeeld wordt op 32 bits. Indien geen range constraint wordt opgegeven bij de declaratie van een integer wordt een integer afgebeeld op een 32 bits (two's complement representatie).

Synthetiseerbaar zijn de volgende operaties:

- De relationele operaties (=, /=, < etc.)
- Een beperkte set van aritmetische operaties:
 - +, - en ABS
 - MOD, REM, *, / indien:
 - beide operanden constanten zijn, of
 - de tweede operand gelijk is aan 2^n ($n = 0,1,2,\dots$). Een macht van twee betekent voor vermenigvuldigen en delen niets anders dan over n

posities schuiven. Voor MOD en REM wordt hiermee bereikt dat 'automatisch' de gewenste functie verkregen wordt door n bits te gebruiken.

5.2.1.3 std_ulogic en std_logic.

Een std_(u)logic wordt afgebeeld op een logische '1' of een logische '0'. Een 'Z' geeft het gebruik van een tri-state uitgang aan (figuur 5.4). De '-' en meestal ook de 'X' worden beschouwd als don't care. Het gebruik van 'H', 'L' en 'W' wordt afgeraden.

Synthetiseerbaar zijn de volgende operaties:

- De relationele operaties (=, /=, < etc.)
- De logische operaties (and, or etc.)
- De concatenatie

5.2.1.4 character.

Een karakter wordt alleen ondersteund indien deze gebruikt wordt om een string te converteren naar een bit_vector, B"..", O"..", of X"..".

Synthetiseerbaar is de volgende operatie:

- De concatenatie

5.2.1.5 user defined.

De gebruiker kan ook zelf nieuwe types definiëren.

- Enumeration²⁷
Een enumeration type kan bijvoorbeeld worden gebruikt om een opsomming te geven van de mogelijke toestanden. Tijdens synthese wordt dit gecodeerd door de bovengrens te nemen van de twee logaritme van het aantal elementen, er wordt een binaire code gebruikt. Indien de gebruiker een andere codering wenst moet dit expliciet in VHDL zijn beschreven.
- Arrays
In het algemeen zijn alleen 1-dimensionale arrays te synthetiseren. Hoewel steeds meer tools ook 2-dimensionale arrays ondersteunen.
- Records
Records worden niet ondersteund. Dit is trouwens een merkwaardige beperking, een record is in feite niets anders dan een verzameling types en leveren voor synthese op zich geen problemen op. Het is dan ook merkwaardig dat dit niet ondersteund wordt.

5.2.2 Expliciete procesbeschrijving

Slechts een tweetal expliciete procesbeschrijvingen zijn toegestaan voor synthese:

- Een proces met een sensitivity list achter het keyword *process*.
- Een proces met één wait statement, en wel een wait until statement op de eerste regel van de procesbeschrijving.

5.2.3 Proces met een sensitivity list

Een proces met een sensitivity list wordt vaak gebruikt om combinatorisch gedrag te beschrijven. In de sensitivity list moet voor synthese alle signalen voorkomen welke in

²⁷ Voor de types 'std_ulogic' en 'std_logic' wordt een uitzondering gemaakt wat betreft de codering. Voor deze enumeratie types wordt exact één bit genomen.

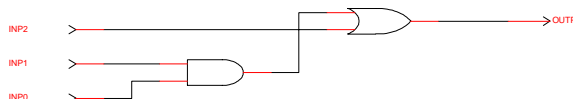
een expressie worden gebruikt. Figuur 5.6 geeft een voorbeeld van een combinatorische schakeling. De beschreven schakeling test of de integer waarde van een `vlbit_vector` groter is dan 2. Merk op dat dit niet uitgeschreven hoeft te worden door middel van booleaanse vergelijking. De gegeven gedragsbeschrijving kan direct gesynthetiseerd worden!

```

library ieee;
use ieee.std_logic_1164.all;
library pack1164;
use pack1164.pack1164.all;28
entity drempel is
  port(inp : in std_ulogic_vector(2 downto 0);
        outp : out std_ulogic);
  -- indien de integer waarde van inp groter
  -- dan 2 wordt outp '1'. Het signaal inp wordt
  -- geïnterpreteerd als unsigned magnitude
end drempel;

architecture gedrag of drempel is
begin
  process(inp)
    constant grens : integer := 2;
  begin
    outp <= boo2std( vld2int(inp) > grens ); -- 29
  end process;
end gedrag;

```



figuur 5.6 Beschrijving van een combinatorisch circuit.

5.2.4 Proces met een wait until statement

Een procesbeschrijving met een wait until statement wordt geïnterpreteerd als een synchroon circuit waarbij het kloksignaal in de conditie is opgenomen.

²⁸ Deze package bevat de conversie functies tussen integers, booleans en `std_(u)logic(_vector)`. Bovendien worden functies gedefinieerd om `std_(u)logic_vectoren` op te tellen en af te trekken.

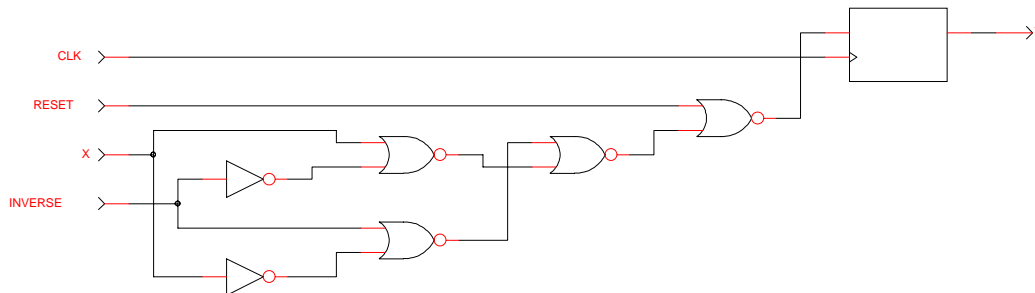
²⁹ "boo2std" is een funktie die een boolean omzet in een `std_ulogic`.
 "vld2int" zet een `vlbit_vector` welke geïnterpreteerd wordt als unsigned magnitude om in een integer.


```

library ieee;
use ieee.std_logic_1164.all;
entity synchroon is
  port (reset, clk, x, inverse : in std_ulogic;
        y : out std_ulogic);
end synchroon;

architecture test of synchroon is
begin
  process
  begin
    wait until clk='1'; --30
    if reset='1'
    then y <= '0';
    elsif inverse='1'
    then y <= not x;
    else y <= x;
    end if;
  end process;
end test;

```



figuur 5.7 Synchrone schakelingen met een synchrone reset.

In figuur 5.7 is een voorbeeld gegeven van een synchrone schakeling met een synchroon reset signaal RESET. Ook kan een asynchrone reset worden gemodelleerd door de reset voorwaarde in het wait statement op te nemen: *wait until clk='1' or reset='1'*.

In een proces mogen ook variabelen worden gedeclareerd. Indien een variabele eerst wordt gebruikt voordat het een waarde krijgt wordt voor zo'n variabele een register gebruikt (figuur 5.8).

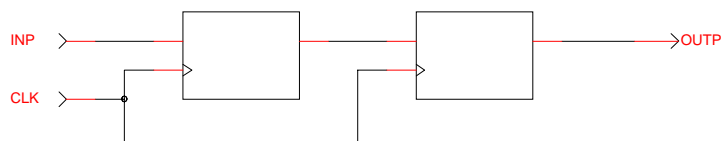
³⁰ In plaats van " wait until clk='1'; " wordt ook vaak het equivalente statement " wait until clk='1' and clk'event; " gebruikt. Ook is er vaak een functie 'rise' o.i.d. aanwezig welke een boolean oplevert die aangeeft of een signaal '1' is geworden.

```

library ieee;
use ieee.std_logic_1164.all;
entity reg_variabele is
  port (inp, clk : in std_logic; outp : out std_logic);
end reg_variabele;

architecture test of reg_variabele is
begin
  process
    variable var : std_logic;
  begin
    wait until clk='1';
    outp <= var;
    var := inp;
  end process;
end test;

```



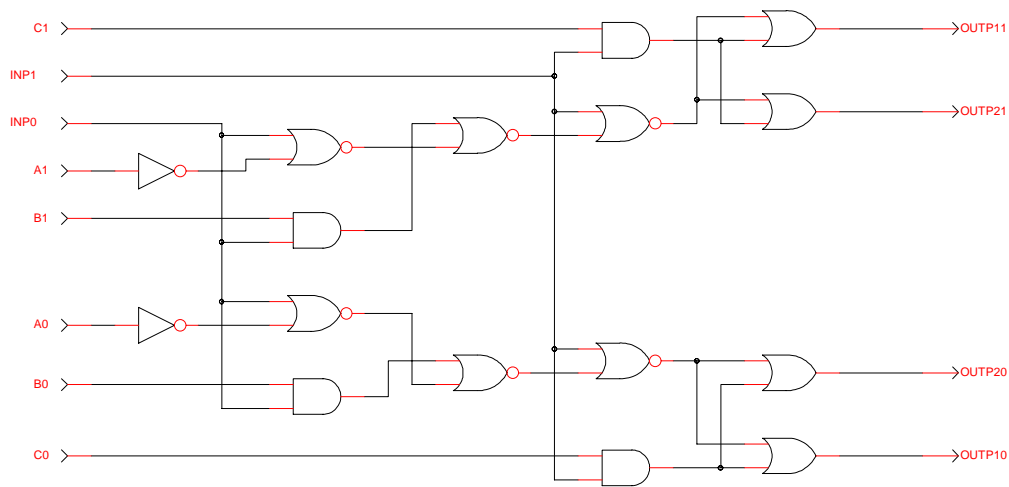
figuur 5.8 Voor het 'realiseren' van variabelen zijn soms registers nodig.

```

library ieee;
use ieee.std_logic_1164.all;
entity concurrent is
  port (a,b,c,inp : in std_ulogic_vector(1 downto 0);
        outp1, outp2 : out std_ulogic_vector(1 downto 0));
end concurrent;

architecture test of concurrent is
begin
  -- concurrent conditional assignment statement
  outp1 <= c when inp(1)='1' else
           b when inp(0)='1' else
           a;
  -- concurrent selected assignment statement;
  with vld2int(inp) select -- 31
    outp2 <= a when 0,
             b when 1,
             c when others;
end test;

```



figuur 5.9 Concurrent control structures.

5.2.5 Impliciete procesbeschrijving

De concurrent control structures, het concurrent conditional assignment statement en het concurrent selected assignment statement, worden beide ondersteund in het synthese proces. Figuur 5.9 geeft van beide beschrijvingen een voorbeeld. Merk op dat beide hetzelfde beschrijven. Dit is ook in het resultaat zichtbaar. Beide beschrijvingen gebruiken dezelfde logica op de vier OR poorten uiterst rechts in het schema na. Deze zijn toegevoegd omdat elke uitgang een eigen driver heeft.

5.2.6 'Standaard' omgeving

In de IEEE standaard 1076-1987 is ook een beschrijving gegeven van de package standard in de library STD. In deze package zijn een minimaal aantal types en functies e.d. opgenomen. Naast deze standaard package heeft elke CAE leverancier een package met een aantal extra functies. Deze functies zijn meestal in een aparte package beschreven zodat, indien de package wordt meegeleverd, er een beschrijving in standaard VHDL kan worden overgedragen aan anderen. Enkele uitbreidingen zijn al eerder genoemd in dit hoofdstuk.

³¹ "vld2int" zet een std_ulogic_vector welke geïnterpreteerd wordt als unsigned magnitude om in een integer.

In VIEWLogic's tool zijn o.a. de volgende uitbreidingen aanwezig:

- Conversie functies van:
 - std_ulogic naar boolean en vice versa
 - std_ulogic naar integer en vice versa
 - std_ulogic_vector naar integer en vice versa
(een integer wordt maximaal op 32 bits afgebeeld)
- Schuifoperaties:
Een vector wordt één positie naar links of rechts geschoven. Waarbij de vector geïnterpreteerd kan worden als een unsigned magnitude (dan wordt een '0' links ingeschoven) of een two's complement representatie (dan wordt het tekenbit links ingeschoven).
- Optellen van vectoren :
 - addum: optellen van bit vectoren waarbij de operanden als unsigned magnitude worden geïnterpreteerd.
 - add2c: optellen van bit vectoren waarbij de operanden als two's complement worden geïnterpreteerd.
- Synthese library van de technologie:
Een veel gehoorde vraag is “hoe beschrijf ik een RAM in VHDL die ook te synthetiseren is?”. Voor een synthese tool is dit geen geringe opgave en wordt dan ook meestal niet of slecht ondersteund. Vaak wordt voor een bepaalde technologie ook nog een bibliotheek met beschikbare componenten aangeleverd. Hierin kan bijvoorbeeld ook een RAM aanwezig zijn en dit is voor een synthese tool dan gunstiger. In figuur 5.10 geeft een voorbeeld gebruikt waarbij Xilinx als technologie is gebruikt met o.a. de componenten m2_1 (een multiplexer) en ram32x8 (een geheugen).

```

library ieee;
use ieee.std_logic_1164.all;
entity ram64x8 is
port (
    A0,A1,A2,A3,A4,A5 : in  std_logic;
    WE                 : in  std_logic;
    D                  : in  std_logic_vector (7 downto 0);
    Q                  : out std_logic_vector (7 downto 0));
end ram64x8;

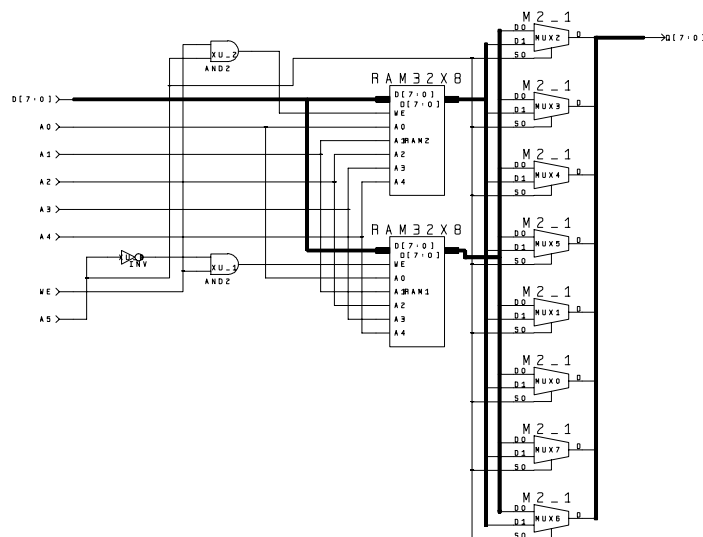
architecture structure of ram64x8 is
    signal WE1         : std_logic;
    signal WE2         : std_logic;
    signal O1          : std_logic_vector(7 downto 0);
    signal O2          : std_logic_vector(7 downto 0);
    component RAM32x8
    port (
        D      : in  std_logic_vector(7 downto 0);
        O      : out std_logic_vector(7 downto 0);
        WE     : in  std_logic;
        A0     : in  std_logic;
        A1     : in  std_logic;
        A2     : in  std_logic;
        A3     : in  std_logic;
        A4     : in  std_logic);
    end component;
    component M2_1
    port (
        S0     : in  std_logic;
        D0     : in  std_logic;
        D1     : in  std_logic;
        O      : out std_logic);
    end component;
begin
    ram1: RAM32x8
        port map(D, O1, WE1, A0, A1, A2, A3, A4);

    ram2: RAM32x8
        port map(D, O2, WE2, A0, A1, A2, A3, A4);

    multiplexers : for i in 0 to 7 generate
        muxi : m2_1 port map (A5, O1(i), O2(i), Q(i));
    end generate;

    WE1 <= WE and not(A5);
    WE2 <= WE and A5;
end structure;

```



figuur 5.10 Bcd code naar een 7-segment code met gebruikmaking van een PLA_table.

5.2.7 Toestandsmachines en de synthese er van.

In een digitaal systeem komt veelvuldig een toestandsmachine voor. Voor een toestandsmachine is er geen unieke beschrijvingswijze in VHDL, echter de wijze van beschrijven heeft wel degelijk invloed op de realisatie. Als toelichting wordt de

toestandsmachine in figuur 5.11 geeft gebruikt. Beide architectures beschrijven dezelfde toestandsmachine.

Het belangrijkste verschil tussen beide beschrijvingswijzen is dat in architecture *een_proces* de toestandsovergangsfunctie en de uitgangsfunctie in één proces is beschreven terwijl in architecture *meerdere_processen* de toestandsovergangsfunctie in een apart proces is opgenomen en de uitgangsfunctie in een tweetal concurrent signal assignment statements. Vaak wordt voor de uitgangsfunctie ook één proces beschreven.

Hoewel beide beschrijvingen exact hetzelfde gedrag hebben zijn er toch een aantal belangrijke verschillen met betrekking tot de synthese er van:

1. *een_proces*

Voor de realisatie van *state*, met zijn vier mogelijke waarden, wordt default uitgegaan van een tweetal flipflops met een binaire codering voor de toestanden. Verder wordt voor elke signaal toekenning in een ‘geklokt’ proces altijd een flipflop gebruikt. Daarmee is het minimaal aantal bits dat een synthese tool zal gebruiken gelijk aan vier. Deze wijze van beschrijven heeft een voordeel, elke uitgang is voorzien van een register. Dit kan belangrijk zijn als deze toestandsmachine onderdeel uitmaakt van een groter systeem welke door een synthese tool gerealiseerd gaat worden omdat het synthese proces moeilijker is indien de tool rekening moet houden met de verschillende aankomsttijden van binnenkomende signalen.

2. *meerdere_processen*

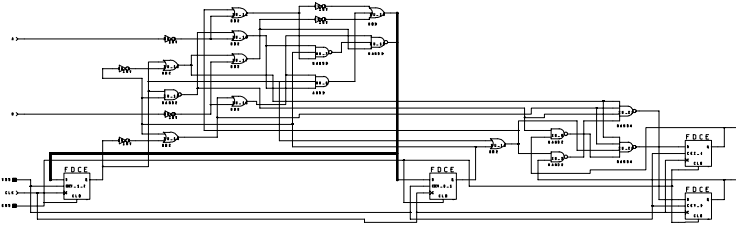
Eveneens is voor de realisatie van *state* weer minimaal twee flipflops nodig. Echter voor de uitgangssignalen zijn nu geen flipflops nodig. De uitgangswaarden worden combinatorisch afgeleid uit de toestandsregisters. Dit heeft als voordeel dat het aantal registers kleiner is maar in het algemeen zijn de uitgangen niet voorzien van een register.

```

entity fsm is
  port (a, b, clk : in bit;
        y, z : out bit);
end fsm;

architecture een_proces of fsm is
begin
  process
    type states is (idle,init,st0,st1);
    variable state : states;
  begin
    wait until clk='1';
    case state is
      when idle => if a='1' then state:=init; end if; y<='0'; z<='1';
      when init => if b='1' then state:=st0; end if; y<='1'; z<='1';
      when st0  => if b='1' then state:=st1; end if; y<='1'; z<='1';
      when st1  => if a='1' then state:=idle; end if; y<='1'; z<='0';
    end case;
  end process;
end een_proces;

```

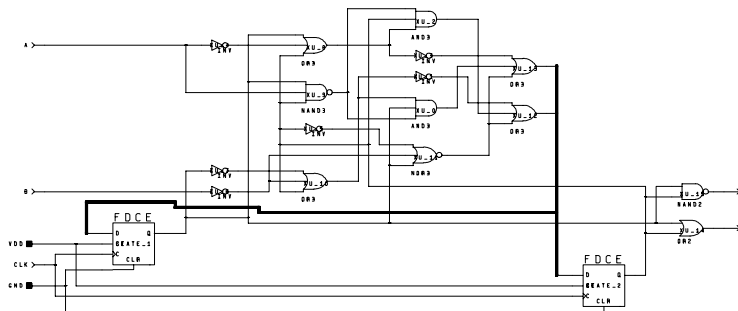


```

architecture meerdere_processen of fsm is
  type states is (idle,init,st0,st1);
  signal state : states;
begin
  process
    begin
      wait until clk='1';
      case state is
        when idle => if a='1' then state<=init; end if;
        when init => if b='1' then state<=st0; end if;
        when st0  => if b='1' then state<=st1; end if;
        when st1  => if a='1' then state<=idle; end if;
      end case;
    end process;

    y <= '1' when state > idle else '0';
    z <= '1' when state < st1  else '0';
  end meerdere_processen;

```



figuur 5.11 Alternatieve VHDL beschrijvingen voor een Moore toestandsmachine met daaronder de realisatie.

5.2.8 IEEE Synthesis Package

Bij het beschrijven van hardware is informatie vaak aanwezig in een vector van bits. Net als de VIEWLogic's tools hebben ook de andere tools vaak hun eigen bewerkingen gedefinieerd, met eigen namen, op vectoren. Om uniformiteit te bereiken wordt door de IEEE synthesis working group gewerkt aan een synthese package

waarin een tweetal getalsrepresentaties zijn gedefinieerd met een groot aantal functies (conversie-functies en ‘overloaded’ functies van operatoren).

De twee representaties zijn:

- unsigned magnitude representatie.
- two's complement representatie.

5.3 'Hoog niveau' synthese

In de vorige paragraaf is aan de hand van een beknopte beschrijving van een synthese systeem een indruk gegeven van wat wel en niet synthetiseerbaar is. Aan de hand hiervan kan bepaald worden tot hoever je een beschrijving in VHDL moet ontwerpen voordat het betrouwbaar gesynthetiseerd kan worden. Er wordt getracht steeds betere synthese systemen te ontwikkelen waardoor de ontwerper steeds minder detail in de VHDL beschrijving hoeft aan te geven, waardoor de beschrijving vaak ook beter te begrijpen blijft. In dit hoofdstuk zullen een drietal voorbeelden worden gegeven.

5.3.1 Meerdere 'wait until' statements in een beschrijving

In het begin beperkten veel synthese systemen zich tot één wait statement per proces. Indien er echter meerdere wait statements voorkomen in de procesbeschrijving zijn impliciet een aantal toestanden aanwezig. Het expliciet vinden van deze toestanden en het coderen ervan kan ook door een synthese systeem worden gedaan.

Een beschrijven zoals gegeven in figuur 5.12 kan door een 'logische synthese' niet worden verwerkt. Echter de meer krachtige 'hoog niveau synthese' tools zijn wel degelijk in staat zijn om dit te realiseren. Dit komt ook in het volgende hoofdstuk aan de orde.

```
process
begin
  wait until clk='1';
  ..
  wait until clk='1'
  ..
  if ..
    then ..
      wait until clk='1';
    ..
  else
    ..
  end if
end process;
```

figuur 5.12 Procesbeschrijving met meerdere wait until statements.

5.3.2 Samennemen van operaties in een beschrijving

Figuur 5.13a geeft een voorbeeld van een beschrijving waarin twee plus operatoren worden gebruikt. In figuur 5.13b wordt dezelfde bechrijving gegeven maar nu met slechts één plus operator. Vaak is een realisatie gegenereerd vanuit deze laatste beschrijving optimaler dan synthese vanuit de eerste beschrijving. Uiteraard is het wenselijk dat een synthese systeem het wel of niet samennemen van operaties zelf kan bepalen.


```

process
begin
  wait until clk='1';
  if up = '1'
    then count <= count + 1; -- count up ==> eerste "+" operator
    else count <= count - 1; -- count down ==> tweede "+" operator
    end if;
end process;

```

a) twee "+" operaties worden beschreven

```

process
  variable updn: integer;
begin
  wait until clk='1';
  if up='1' then updn:=1; else updn:=-1; end if;
  count <= count + updn; -- slechts e e n "+" operator
end process;

```

b) e e n "+" operatie wordt beschreven

figuur 5.13 Optel operatie komt tweemaal voor.

5.3.3 Resource sharing

Evenals het samennemen van dezelfde operaties, zoals beschreven is in de vorige paragraaf, kunnen ook registers voor meerdere doeleinden worden gebruikt. In figuur 5.14 is een beschrijving gegeven met twee variabelen REG1 en REG2. Voor de realisatie van beide variabelen zal een register nodig zijn echter indien variabele REG1 een nieuwe waarde krijgt wordt deze eerst weer gelezen, en daarna herhaalt dit zich voor register REG2. Beide registers bevatten dus nooit tegelijk data die later weer gebruikt gaat worden, dus kunnen beide registers gerealiseerd worden met slechts één register.

```

process
  variable reg1, reg2 : bit;
begin
  wait until clk='1';
  outp <= reg2;
  reg1 := .....
  wait until clk='1';
  outp <= reg1;
  reg2 := ..
end process;

```

figuur 5.14 Meerdere variabelen in de VHDL beschrijvingen die gebruik kunnen maken van hetzelfde register.

5.4 Samenvatting

Niet alle VHDL beschrijvingen zijn synthetiseerbaar. In dit hoofdstuk is een indruk gegeven van wat wel en wat niet te synthetiseren is.

In het algemeen geldt dat alle realisaties met simulatietijd niet te realiseren zijn. Dus expliciet opgegeven vertragingstijden worden genegeerd, ook attributen welke gerelateerd zijn aan de tijd worden niet ondersteund. Realisatie van dynamische structuren en recursieve functies is eveneens niet mogelijk. Verder wordt het event-driven simulatiemodel niet in alle opzichten gesynthetiseerd.

De meeste synthese systemen gaan uit van synchrone ontwerpen. In dat geval is het resultaat meestal 'correct by construction'. Verificatie, bijvoorbeeld door middel van simulatie, van het gerealiseerde tegen het gewenste gedrag blijft altijd wenselijk omdat de synthese software nooit foutloos is. Afhankelijk van het soort beschrijving

(data-pad of besturing) worden soms ook verschillende algoritmen gebruikt om een optimaal resultaat te krijgen. Of snelheid of oppervlak belangrijk is kan de synthese-tool niet weten, daarom kan dit in de tool worden aangegeven.

CAE systemen hebben naast de standaard package vaak een groot aantal extra functies en procedures waardoor beschrijvingen eenvoudiger worden. Deze functies en procedures worden meestal ook door een synthese-tool ondersteund.

6. Eenvoudige voorbeelden.

Na de inleidende hoofdstukken over VHDL en de synthese ervan zal in dit hoofdstuk aan de hand van een aantal problemen worden aangegeven hoe VHDL daadwerkelijk ingezet kan worden. Verder is geprobeerd een VHDL beschrijving zo op te zetten dat de ‘parameters’ gemakkelijk aangepast kunnen worden zodat een VHDL beschrijving eenvoudig hergebruikt kan worden voor iets afwijkend probleem.

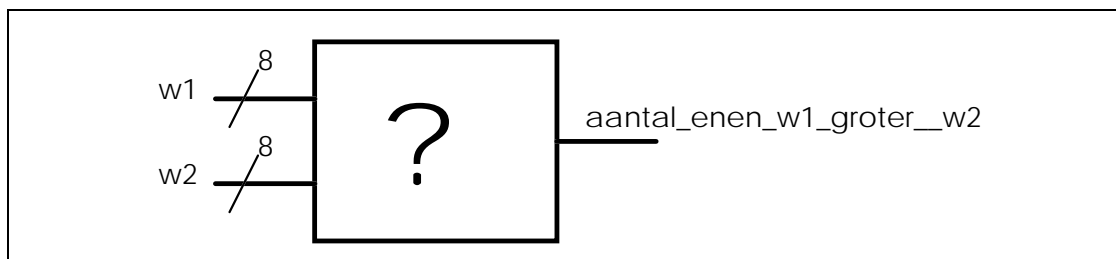
6.1 Welke input bevat meer enen?

6.1.1 Inleiding.

Deze kleine case zal worden gebruikt om een aantal aspecten van VHDL en synthese van VHDL nader toe te lichten.

Een systeem moet worden ontworpen welke twee ingaande bussen, respectievelijk $w1$ en $w2$, heeft met een breedte van n bits en een uitgangssignaal dat aangeeft of ingang $w1$ meer enen bevat dan ingang $w2$.

Er mag zowel een combinatorisch als een sequentieel circuit ontworpen worden, waarbij in het laatste geval nog een klokingang toegevoegd moet worden.



figuur 6.1 Input/output van het te ontwerpen systeem met $n=8$.

6.1.2 Specificatie in VHDL.

```
entity verschil_aantal_enen is
  generic (breedte : natural := 8)
  port (woord1 : in bit_vector(breedte-1 downto 0);
        woord2 : in bit_vector(breedte-1 downto 0);
        aantal_w1_groter_w2 : out bit);
end verschil_aantal_enen;

architecture gedrag of verschil_aantal_enen is
begin
  process(woord1,woord2)
    function aantal_enen (inp : in bit_vector) return integer is
      variable aantal : integer;
    begin
      aantal := 0;
      for i in inp'range loop
        if inp(i)='1' then aantal := aantal+1; end if;
      end loop;
      return aantal;
    end aantal_enen;
  begin
    if aantal_enen(woord1) > aantal_enen(woord2)
    then aantal_w1_groter_w2 <= '1';
    else aantal_w1_groter_w2 <= '0';
    end if;
  end process;
end gedrag;
```

figuur 6.2 Gedragsbeschrijving.

Figuur 6.2 beschrijft het gewenste gedrag in VHDL. Als specificatie is dit een prima beschrijving, immers zonder (veel) commentaar is duidelijk wat de functie is! Het

doel van een specificatie is niet het beschrijven van een mogelijke implementatie, hoewel de beschrijving misschien wel een implementatie suggereert.

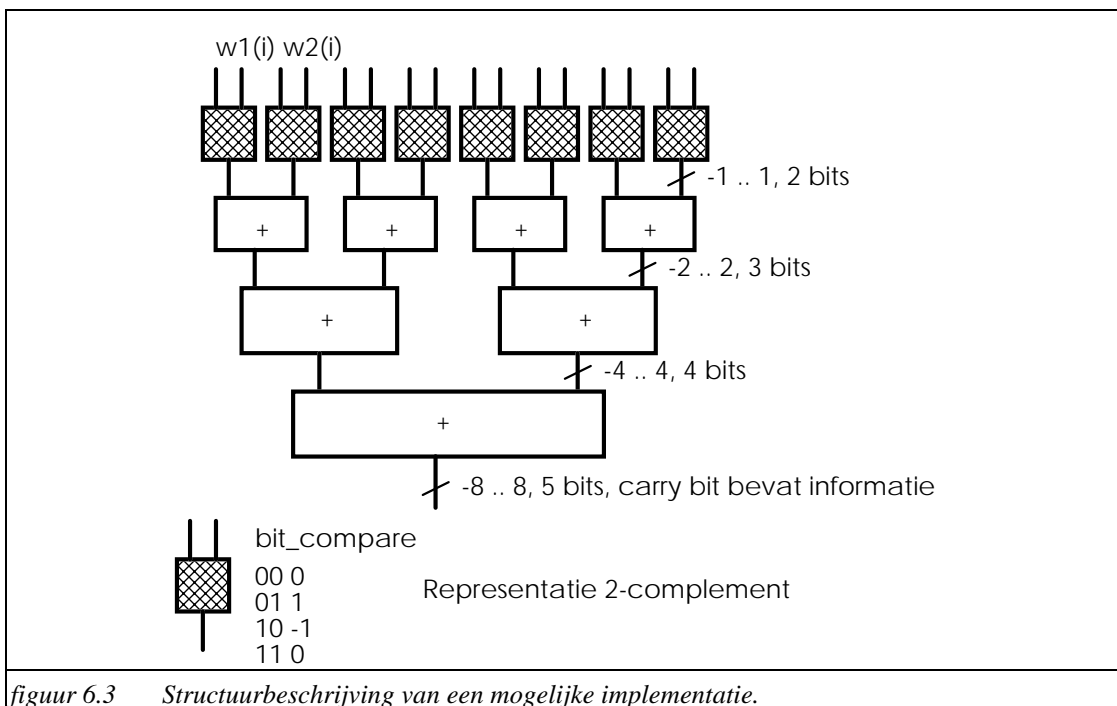
Nu wil het toeval dat veel specificaties ook voldoen aan de subset welke wordt ondersteund door een synthese tool. Maar het voldoen aan de subset betekent nog niet dat er een optimaal circuit gesynthetiseerd gaat worden. Concreet betekent dit dat meestal nog aan een aantal randvoorwaarden moet zijn voldaan met betrekking tot o.a. de grootte, de snelheid en de vermogensdissipatie.

Bovenstaande beschrijving zal door de meeste synthese tools geaccepteerd worden. Enkele synthese resultaten:

Tool	Technologie	oppervlak	max. vertraging
Viewsynthesis 2.4.1	Xilinx	97	38
Viewsynthesis 2.4.1	Actel	221	170
Synopsys Design Analyzer 3.2a	Actel	178	132

Indien dit circuit niet te groot of te traag is, is zeer snel een circuit ontworpen, dat achteraf ook nog eenvoudig aan te passen is voor andere breedtes van de bussen. Echter indien het circuit bijvoorbeeld te groot is zal de ontwerper zelf een implementatie moeten maken.

6.1.3 Implementatie.



figuur 6.3 Structuurbeschrijving van een mogelijke implementatie.

Hoe kan het gewenste gedrag worden gerealiseerd? Deze creatieve ontwerpstep kan bijvoorbeeld leiden tot het circuit zoals is gegeven in figuur 6.3. Het idee achter deze implementatie is dat overeenkomstige bitposities met elkaar worden vergeleken, door middel van component *bit_compare*. Daarna worden de verschillen opgeteld door de boom van optellers, waarvan het bereik van de op te tellen operanden steeds groter wordt. Uiteindelijk wordt het verschil in het aantal enen als resultaat opgeleverd. Indien het verschil negatief is moet het uitgangssignaal '1' zijn en anders '0'. Omdat gekozen is voor de 2-complement representatie geeft het meest linkse het gewenste resultaat. Merk op dat bij de implementatie is verondersteld dat de breedte van de

ingaaende bus een macht van twee is. Indien dit niet het geval is kan eenvoudig gekozen worden voor de eerstvolgende macht van twee en de extra ingangen verbinden met een '0'. Bij de optimalisatie door de synthese tool wordt hopelijk de overbodige combinatoriek opgeruimd.

In deze beschrijving komen twee soorten combinatorische schakelingen voor:

1. bitcompare
2. opteller

6.1.3.1 Bitcompare.

```
entity bit_compare is
  port (a,b : in bit;
        s : out bit_vector(1 downto 0));
end bit_compare;

architecture gedrag of bit_compare is
begin
  process(a,b)
    subtype bitv2 is bit_vector(1 downto 0);
    begin
      case bitv2'(a&b) is
        when "00" | "11" => s <= "00";
        when "01"       => s <= "01";
        when "10"       => s <= "11";
      end case;
    end process;
  end gedrag;
```

figuur 6.4 Entity bit_compare.

6.1.3.2 Opteller.

```
package arithm is
  function add2c (a,b : bit_vector) return bit_vector;
end arithm;

package body arithm is
  function add2c (a,b : bit_vector) return bit_vector is
    variable ai,bi,res : bit_vector(a'length downto 0);
    variable carry : bit;
  begin
    ai := a(a'left) & a;
    bi := b(b'left) & b;
    carry := '0';
    for i in ai'reverse_range loop
      res(i) := ai(i) xor bi(i) xor carry;
      carry := (ai(i) and bi(i)) or
               (ai(i) and carry) or
               (bi(i) and carry);
    end loop;
    return res;
  end add2c;
end arithm;

entity add_special is
  generic (breedte : natural := 2);
  port (a_b : in bit_vector(2*breedte-1 downto 0);
        s : out bit_vector(breedte downto 0));
end add_special;

use work.arithm.all;
architecture gedrag of add_special is
  alias a : bit_vector(breedte downto 1)
    is a_b(breedte-1 downto 0);
  alias b : bit_vector(breedte downto 1)
    is a_b(2*breedte-1 downto breedte);
begin
  s <= add2c(a,b);
end gedrag;
```

figuur 6.5 De opteller.

Aangezien de breedte van de operanden van de opteller varieert, is hiervoor gebruik gemaakt van een generic. Voor een correcte afhandeling van de optelling van 2-

complement representatie getallen worden de operanden intern eerst met het tekenbit verlengd.

6.1.3.3 Structuurbeschrijving.

```
entity verschil_enen is
  generic ( n : natural := 2); -- breedte = 2**n
  port ( woord1, woord2 : in bit_vector(2**n downto 1);
        aantal_wl_groter_w2 : out bit);
end verschil_enen;

architecture structuur of verschil_enen is
  component bit_compare
    port (a,b : in bit;
          s : out bit_vector(1 downto 0));
  end component;
  component add_special
    generic (breedte : natural := 2);
    port (a_b : in bit_vector(2*breedte-1 downto 0);
          s : out bit_vector(breedte downto 0));
  end component;
  type dim2 is array(0 to n) of bit_vector(2**(n+1) downto 1);
  signal intern : dim2;
  constant breedte : natural := 2**n;
begin
  rij_inp : for i in 1 to breedte generate
    bc : bit_compare
      port map (woord1(i), woord2(i), intern(n)(2*i downto 2*i-1) );
  end generate;

  rijen : for rij in 1 to n generate
    kolommen : for kolom in 1 to 2**(rij-1) generate
      adder:add_special
        generic map (n+2-rij)
        port map (intern(rij)((n+2-rij)*2*kolom downto
                           ((n+2-rij)*2*kolom)-((n+2-rij)*2-1)),
                  intern(rij-1)((n+3-rij)*kolom downto
                              ((n+3-rij)*kolom-(n+2-rij))));
    end generate;
  end generate;
  -- "intern(0)(n+2 downto 1)" bevat het verschil aantal enen
  aantal_wl_groter_w2 <= intern(0)(n+2);
end structuur;
```

figuur 6.6 De structuur beschrijving van de implementatie.

De regelmatige structuur van de implementatie is in figuur 6.6 beschreven door gebruik te maken van een geneste generate statement. Het tot stand komen van deze implementatie heeft aanzienlijk meer tijd geveerd dan de gedragsbeschrijving (waarvoor slechts 10 minuten nodig waren) en is bovendien ook foutgevoeliger. Er dient dan ook een VHDL beschrijving te worden gemaakt waarin de implementatie en het gedrag met elkaar worden vergeleken.

Het syntheseresultaat met Synopsys Design Analyzer, versie 3.2a and Actel als technologie is het oppervlak 58 en de maximale vertraging 64.

6.1.3.4 Synthese-resultaat van gedrag en structuur vergeleken.

Synopsys versie 3.2a, technologie Actel			
type	oppervlak	vertraging	'ontwerptijd'
gedrag	178	132	++
structuur	58	64	-

figuur 6.7 Vergelijking van de syntheseresultaten.

In figuur 6.7 zijn de syntheseresultaten van de gedragsbeschrijving en de implementatie vergeleken. De gedragsbeschrijving is zowel wat betreft oppervlak als vertraging slechter dan de implementatie. De tijd die nodig is om tot een correcte beschrijving te komen is voor de gedragsbeschrijving aanzienlijk gunstiger. Men moet een synthese tool dan ook leren kennen om er op een optimale wijze gebruik van te

kunnen maken: “tot welk detail en met welke taal constructies moet ik mijn beschrijving beschrijven?”. De ervaring heeft ook geleerd dat te veel detail in een VHDL beschrijving het synthese-resultaat weer nadelig kan beïnvloeden. Het is dus van belang om bij nieuwe updates van de tool altijd zorgvuldig de manual te lezen, immers een taal constructie die in het verleden werd afgeraden kan nu juist worden geprefereerd.

6.1.3.5 Sequentiële oplossing.

Vaak kan ook hardware worden bespaard door de bewerking in de tijd ‘uit te smeren’. Per klok-periode wordt een deelbewerking uitgevoerd. Een deelbewerking zou kunnen zijn het vergelijken van bitpositie i van de ingaande bussen met behulp van component *bit_compare* (figuur 6.4) en het verschil op te tellen bij de tot bitpositie i gevonden verschillen.

Dit betekent dat in het data-pad de volgende componenten voorkomen:

1. Twee multiplexers voor het selecteren van een bit positie.
 2. Een component *bit_compare*.
 3. Een opteller.
 4. Een register voor het onthouden van het verschil in het aantal enen tot nu toe.
- Daarnaast is er uiteraard nog een besturing nodig. Verwacht kan worden dat vooral bij brede bussen de sequentiële oplossing minder hardware nodig heeft, waarbij het circuit wel iets trager zal zijn.

```
entity verschil_aantal_enen is
  port (woord1 : in bit_vector(3 downto 0);
        woord2 : in bit_vector(3 downto 0);
        clk    : in bit;
        aantal_wl_groter_w2 : out bit);
end verschil_aantal_enen;

architecture sequentieel of verschil_aantal_enen is
begin
  process
    variable verschil_enen : integer;
    variable two_bit : bit_vector(1 downto 0);
  begin
    wait until clk='1'; -- ViewArchitect Synthesis Guideline
    verschil_enen := 0;
    for i in woord2'range loop
      wait until clk='1';
      two_bit := woord2(i) & woord1(i);
      case two_bit is
        when "01" => verschil_enen := verschil_enen + 1;
        when "10" => verschil_enen := verschil_enen - 1;
        when others => null;
      end case;
    end loop;
    if verschil_enen > 0
      then aantal_wl_groter_w2 <= '1';
    else aantal_wl_groter_w2 <= '0';
    end if;
  end process;
end sequentieel;
```

figuur 6.8 Een sequentiële implementatie.

De VHDL beschrijving (figuur 6.8) wordt door een ‘logische synthese tool’ niet geaccepteerd, maar wel door een hoog niveau synthese tool zoals VIEWArchitect. Deze genereert een VHDL beschrijving op RTL niveau waarbij nog geen keuze voor de technologie wordt opgegeven. Er wordt een schatting gegeven voor de onder- en de bovengrens van het aantal poorten.

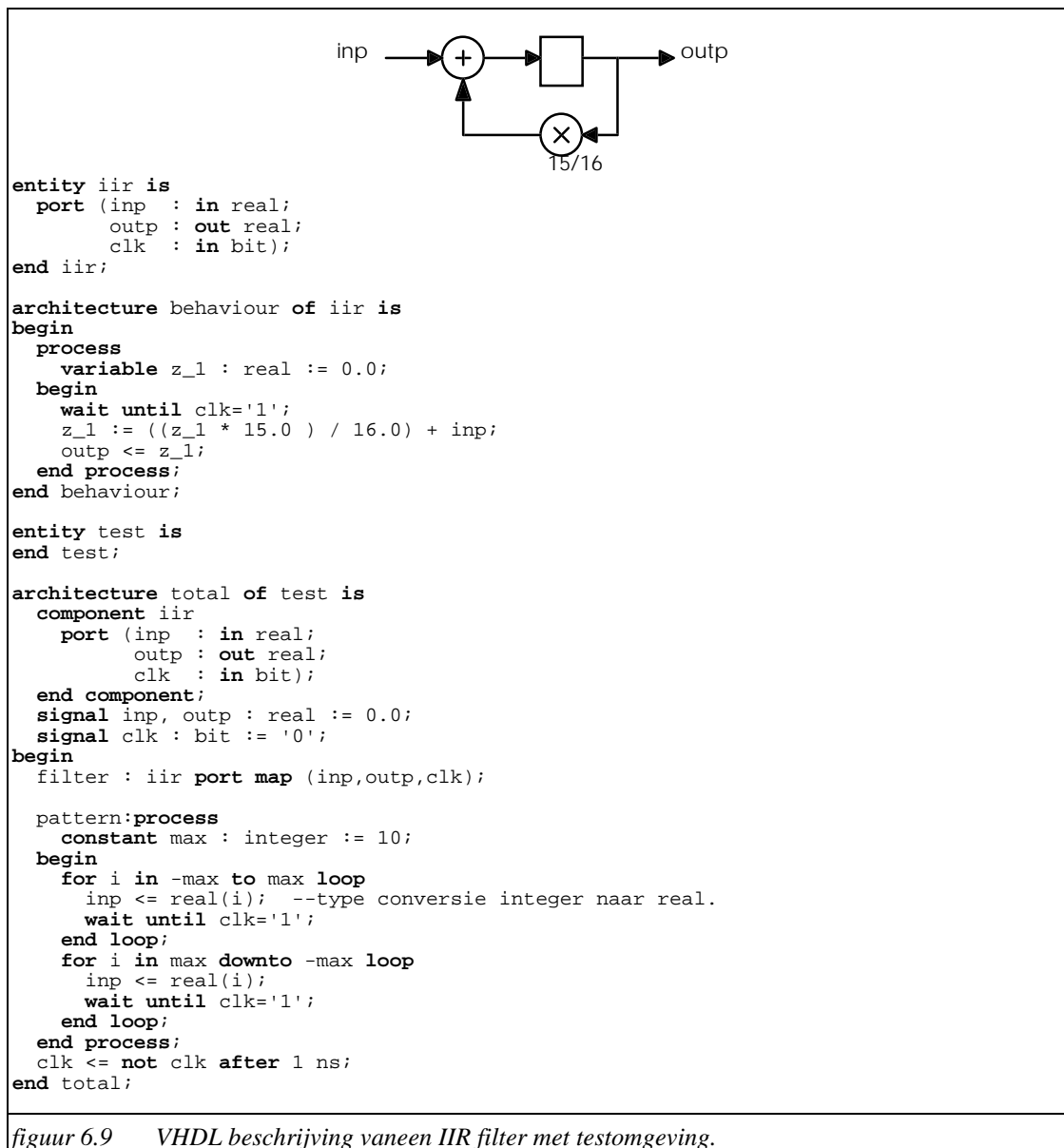
Zowel de gedragsbeschrijving (figuur 6.2) als deze sequentiële oplossing zijn gesynthetiseerd.

Het resultaat van VIEWArchitect is:

Welke input bevat meer enen?

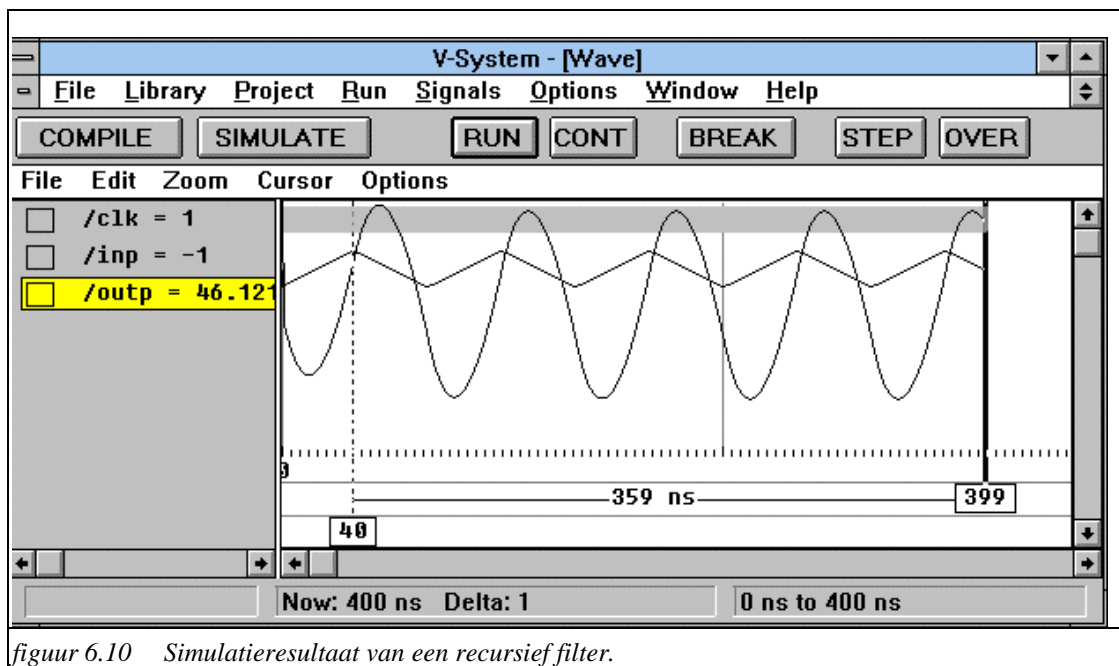
gedrag	2725 <= aantal gates <= 5003
sequentieel	1634 <= aantal gates <= 1808

6.2 Recursief filter.



figuur 6.9 VHDL beschrijving vaneen IIR filter met testomgeving.

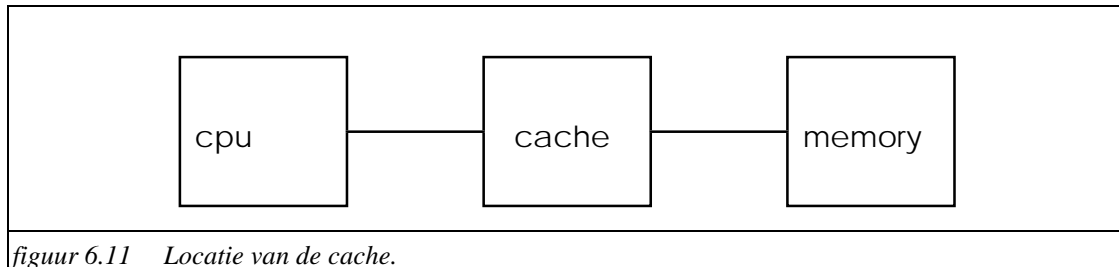
Van een recursief filter (Infite Impulse Response filter; IIR filter) is in figuur 6.9 een VHDL beschrijving gegeven. Voordat overgegaan kan worden tot een implementatie als digitaal systeem zal o.a. een passende representatie gevonden moeten worden voor de typering. Ook is in figuur 6.9 een testomgeving gegeven. Figuur 6.10 geeft grafisch het simulatieresultaat weer over de eerste 400 ns.



figuur 6.10 Simulatieresultaat van een recursief filter.

6.3 Cache.

6.3.1 Inleiding.



Als het geheugen traag is vergeleken met de cpu kan de performance van het totale systeem worden verbeterd door het toevoegen van een cache. Dit bestaat onder andere uit een relatief klein maar snel geheugen. Figuur 6.11 geeft een mogelijke locatie van de cache weer. De cache bevat een copie van de gegevens uit het primaire geheugen ('memory'). Als er data van een adres uit het geheugen nodig is maar een copie ervan is ook aanwezig in de cache (dit wordt een 'hit' genoemd), kan deze data snel worden geleverd. Is de data niet aanwezig in de cache dan wordt deze uit het relatief trage primaire geheugen gelezen en wordt hiervan ook een copie in het cache geheugen geplaatst. Niet alleen de data van dat ene adres komt in het cache geheugen, maar ook die uit de directe omgeving van het adres en juist deze gegevens zijn meestal ook op korte termijn nodig.

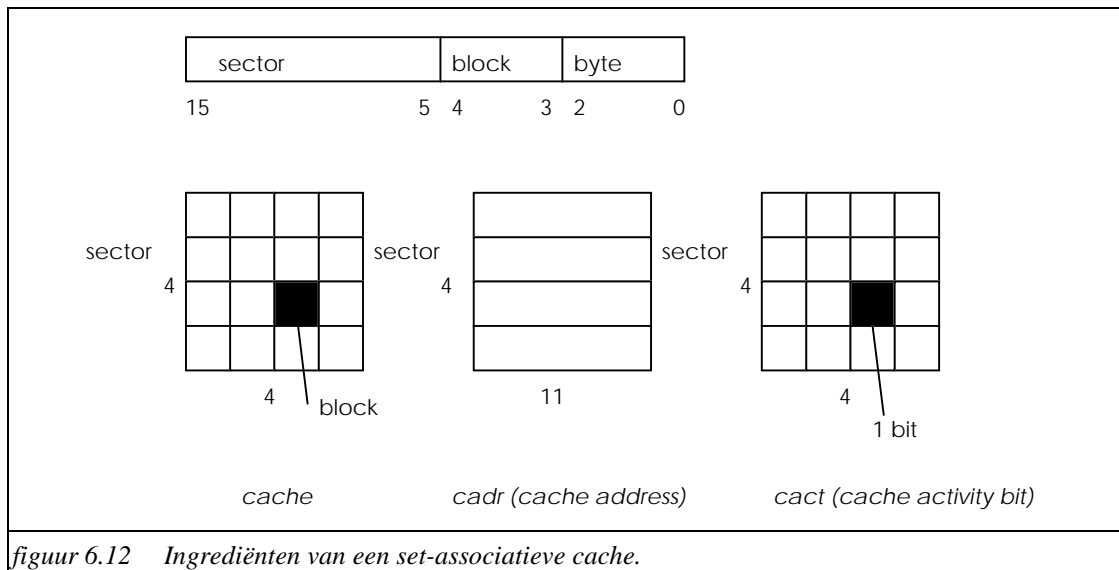
Zoals al eerder is aangegeven wordt begonnen met het opstellen van een specificatie in VHDL van de te ontwerpen cache. Maar wat is de specificatie van een cache? Functioneel gezien doet de cache niets; in feite kunnen de in- en uitgangen direct met elkaar worden doorverbonden¹. Je zou de cache kunnen specificeren door de 'hit-rate' vast te stellen maar deze hangt weer af van de applicatie die gebruikt wordt.

Er zal dan ook alleen een algoritmische beschrijving van een cache worden gegeven die gebaseerd is op de APL beschrijving van Blaauw².

¹ Citaat uit de consumentengids Mei 1995, pagina 338 "**Nep-chips**. Ook rommelen met cache-chips blijkt mogelijk. Deze chips zijn buffergeheugens die ervoor zorgen dat de computer sneller kan werken. In Duitsland zijn inmiddels computers ontdekt met nep-cache-chips. Ze zitten netjes op de printplaat, maar van binnen zijn ze leeg. ..."

² Blaauw, G.A., Digital System Implementation, Prentice Hall, Inc., 1976

6.3.2 Set-associatieve cache.



figuur 6.12 Ingrediënten van een set-associatieve cache.

In een set-associatieve cache is een adres opgedeeld in drie delen (in dit voorbeeld). Het adres is 16 bits breed. De drie minst significante bits geven het aantal bytes in een 'block' weer (2^3 bytes). Iedere keer als er data wordt gelezen in de cache wordt er data ter grootte van een block ingelezen. Bovendien wordt een 'cache activity bit' geset als teken dat de data geldig is. Verder bevat elke sector een viertal 'blocks'. Als data wordt weggeschreven naar het geheugen door de cpu kan de data in de cache ook worden aangepast. Maar ook kan dan volstaan worden met het resetten van het 'cache activity bit' waardoor de data in de cache ongeldig is.

Natuurlijk komt er een moment dat de cache vol is. Indien er dan data nodig is die nog niet voorkomt in de cache moet er ruimte worden vrijgemaakt in de cache, maar waar? Allerlei algoritmen zijn mogelijk. In dit voorbeeld wordt gemaakt van het LRU (least recently used) algoritme.

6.3.3 VHDL beschrijving van de set-associatieve cache.

Een tweetal packages zijn beschreven voor respectievelijk een resolutiefunctie (figuur 6.13) en een package die de veel gebruikte types, het LRU algoritme e.d. bevat (figuur 6.14).

Het algoritme van de cache is gegeven in figuur 6.15. Het volgende is beschreven:

- Lezen van het primair geheugen door cpu:
 1. Is de 'sector' aanwezig in de cache?
 - Ja: Naar punt 2.
 - Nee: Verwijder 'least recently used' sector en lees een block data in uit het primair geheugen.
 2. Is block geldig? ('cache activity bit')
 - Ja: Naar punt 3.
 - Nee: Lees 'block' data uit het primair geheugen.
 3. Data in de cache is geldig
- Schrijven van data naar het geheugen door de cpu:

Als de sector in de cache voorkomt en de data in het 'block' is geldig dan moet het 'cache activity bit' gereset worden.

Tot slot is nog een eenvoudige testomgeving opgenomen (figuur 6.16) waarmee de lees- en schrijfacties van de cpu worden gesimuleerd en het geheugen. Figuur 6.17 toont een deel van de simulatieresultaten.

```

package types is
  type bit4 is ('X','Z','0','1');
  type bit4_vector is array (natural range <>) of bit4;
  subtype byte is bit4_vector(7 downto 0);
  type byte_vector is array (natural range <>) of byte;
  function bus_res (inp : in byte_vector) return byte;
  subtype bbyte is bus_res byte;
end types;

package body types is

  function bit4_resolve (a,b : in bit4) return bit4 is
    type bit4_table is array (bit4'left to bit4'right,
                              bit4'left to bit4'right) of
                              bit4;
    constant tbl: bit4_table := -- 'X' 'Z' '0' '1'
                                ((('X','X','X','X'), -- 'X'
                                   ('X','Z','0','1'), -- 'Z'
                                   ('X','0','0','X'), -- '0'
                                   ('X','1','X','1')));-- '1'

    variable result: bit4 := 'Z';
  begin
    return tbl(a,b);
  end bit4_resolve;

  function bit4_vector_resolve (a,b : in bit4_vector) return bit4_vector is
    constant ai : bit4_vector(1 to a'length) := a;
    constant bi : bit4_vector(1 to b'length) := b;
    variable res : bit4_vector(1 to a'length);
  begin
    if ai'length /= bi'length
    then
      assert false report "not equal vector length in bit4_vector" severity error;
    else
      for i in ai'range loop
        res(i) := bit4_resolve( ai(i), bi(i) );
      end loop;
    end if;
    return res;
  end bit4_vector_resolve;

  function bus_res (inp : in byte_vector) return byte is
    variable res : byte := (others => 'Z');
  begin
    for i in inp'range loop
      res := bit4_vector_resolve( res, inp(i));
    end loop;
    return res;
  end bus_res;
end types;

```

figuur 6.13 'Resolved type' nodig voor de beschrijving van de cache.

```

use work.types.all;
package cache_help is
  subtype address is bit_vector(15 downto 0);
  type cadr_t is array(0 to 3) of bit_vector(10 downto 0);
  type cact_t is array(0 to 3, 0 to 3) of bit;
  type cach_t is array(0 to 3, 0 to 3, 0 to 7) of byte;
  type list is array(0 to 3) of integer range 0 to 3;
  procedure match (sector : bit_vector;
                  cache  : cadr_t;
                  pos    : out integer; -- position
                  avai   : out boolean);
    -- if sector is in cache then "pos", gives the position
    -- if not "avai" is false;

  function lru(usr_lst:list) return integer;
    -- the least recently used index in cadr_t is returned
    -- a local variable LIST is used to keep
    -- track of the used sectors
  procedure ru(usr_lst: inout list; who : integer);
    -- the recent used
    -- a local variable LIST is used to keep
    -- track of the used sectors
  function intval( inp : bit_vector) return integer;
    -- returns integer value of the bit_vector;

  function int2bitv(width,inp:integer) return bit_vector;
    -- encodes the integer value to a bitstrings of length width
end cache_help;

package body cache_help is
  function bit2int (inp : bit) return integer is
  begin
    if inp='1' then return 1; end if;
    return 0;
  end bit2int;

  function intval( inp : bit_vector) return integer is
    variable res : integer := 0;
  begin
    for i in inp'range loop
      res := res*2 + bit2int(inp(i));
    end loop;
    return res;
  end intval;

  procedure match (sector : bit_vector;
                  cache  : cadr_t;
                  pos    : out integer; -- position
                  avai   : out boolean) is
  begin
    for i in cache'range loop
      pos:=i; avai:=TRUE;
      exit when intval(sector)=intval(cache(i));
      avai := FALSE;
    end loop;
  end match;

  function card2bit (a : in integer) return bit is
  begin
    if a=1 then return '1'; else return '0'; end if;
  end card2bit;

  function int2bitv(width,inp:integer)
    return bit_vector is
    variable tmp : bit_vector(width downto 1);
    variable rm : integer;
  begin
    rm := inp;
    for i in 1 to width loop
      tmp(i) := card2bit(rm rem 2);
      rm := rm/2;
    end loop;
    return tmp;
  end int2bitv;

  function ffocc(usr_lst:list; nr:integer)
    return integer is
    -- find_first_occurencs of integer in list, returns
    -- that position. If no in list then return usr_lst'right
    variable i : integer := 0;
  begin
    loop
      exit when (usr_lst(i)=nr) or (i>=3);
      i:=i+1;
    end loop;
  end ffocc;

```

```
        end loop;  
        return i;  
    end ffocc;  
  
    function lru(usg_lst:list) return integer is  
    begin  
        return usg_lst(usg_lst'right);  
    end lru;  
  
    procedure ru(usg_lst:inout list; who : integer) is  
    variable pos:integer:=0;  
    begin  
        pos:=ffocc(usg_lst,who);  
        usg_lst:=who & usg_lst(0 to pos-1) & usg_lst(pos+1 to usg_lst'right);  
    end ru;  
end cache_help;
```

figuur 6.14 Gebruikte types en functies/procedures ten behoeve van de cache.


```

use work.types.all;
use work.cache_help.all;
entity cache is
port ( data_cpu   : inout bbyte;
      addr_cpu   : in address;
      RWr_cpu    : in bit;
      req_cpu    : in bit;
      ack_cpu    : out bit;
      data_mem   : inout bbyte;
      addr_mem   : out address;
      RWr_mem    : out bit;
      req_mem    : out bit;
      ack_mem    : in bit);
end cache;

architecture algorithm of cache is
begin
  process
    alias sctr : bit_vector(15 downto 5) is addr_cpu(15 downto 5);
    alias blk  : bit_vector(4  downto 3) is addr_cpu(4  downto 3);
    alias wrd  : bit_vector(2  downto 0) is addr_cpu(2  downto 0);
    variable who : integer; -- points to row in cadr
    variable avl_sctr : boolean; -- is sector present in cache?
    variable useage_list : list := (0,1,2,3); -- useage list
    variable cach : cach_t;
    variable cadr : cadr_t;
    variable cact : cact_t;
  begin
    data_mem<= (others => 'Z');
    data_cpu<= (others => 'Z');
    wait until req_cpu='1';
    if RWr_cpu='1'
    then
      -- test for sector present
      match(sctr,cadr,who,avl_sctr);
      if not avl_sctr then
        who:=lru(useage_list);
        cadr(who):=sctr;
        for i in 0 to 3 loop
          cact(who,i):='0';
        end loop;
      end if;
      -- test for block present
      if cact(who,intval(blk))='0' then
        -- fetch block
        RWr_mem <= '1';
        for i in 0 to 7 loop
          addr_mem <= sctr & blk & int2bitv(3,i);
          req_mem <= '1';
          wait until ack_mem='1';
          cach(who,intval(blk),i):=data_mem;
          req_mem <= '0';
          wait until ack_mem='0';
        end loop;
        cact(who,intval(blk)):= '1';
      end if;
      -- read from cache
      data_cpu<=cach(who,intval(blk),intval(wrd));
      -- update useage list
      ru(useage_list,who);
      ack_cpu <= '1';
      wait until req_cpu='0';
      data_cpu <= (others => 'Z');
      ack_cpu <= '0';
    else -- write to primary memory
      -- clear cache activity bit, if in cache
      match(sctr,cadr,who,avl_sctr);
      if avl_sctr then
        cact(who,intval(blk)):= '0';
      end if;
      RWr_mem <= '0';
      addr_mem <= addr_cpu;
      data_mem <= data_cpu;
      req_mem <= '1';
      wait until ack_mem = '1';
      req_mem <= '0';
      data_mem <= (others => 'Z');
      wait until ack_mem = '0';
      ack_cpu <= '1';
      wait until req_cpu='0';
      ack_cpu <= '0';
    end if;
  end process;
end algorithm;

```

figuur 6.15 Algoritmische beschrijving van de cache.

```

use work.types.all;
use work.cache_help.all;
-- "cpu" read from/write to memory
entity cpu is
    port (data : inout bbyte;
          addr : out address;
          Rwn : out bit;
          req : out bit;
          ack : in bit);
end cpu;

architecture read_write_actions of cpu is
    type addr is array(0 to 10) of address;
    constant used_addr : addr :=
        (B"000000000011_01_000", -- r      r=read. w=write
         B"000000000011_01_001", -- r
         B"000000000011_01_010", -- w      cache invalid
         B"000000000011_01_000", -- r
         B"000000000000_11_000", -- r
         B"000000000001_11_010", -- r
         B"000000000010_01_010", -- r
         B"000000000100_11_100", -- r      last used entry (01) removed
         B"000000000011_01_000", -- r      is new data read from memory ?
         B"000000000011_01_010", -- r
         B"000000000011_00_000"); -- r
    type rwn_vector is array (0 to 10) of bit;
    constant rwn_vec : rwn_vector := B"110111111111";
begin
    process
        variable dummy : bbyte;
    begin
        data <= (others => 'Z');
        for i in rwn_vec'range loop
            Rwn<=rwn_vec(i); addr<= used_addr(i);
            if rwn_vec(i)='1'
            then -- read data from memory
                req <='1' after 10 ns;
                wait until ack='1';
                dummy := data;
                req <='0' after 10 ns;
                wait until ack='0';
            else -- write data to memory
                data <= "01010011"; -- random number
                req <= '1' after 10 ns;
                wait until ack='1';
                data <= (others => 'Z');
                req <= '0' after 10 ns;
                wait until ack='0';
            end if;
        end loop;
    end process;
end read_write_actions;

-----

use work.types.all;
use work.cache_help.all;
entity memory is
    port (data : inout bbyte;
          addr : in address;
          Rwn : in bit;
          req : in bit;
          ack : out bit);
end memory;

architecture behavior of memory is
begin
    process
        type mem is array (0 to 255) of bbyte;
        variable memory : mem :=
            (104 => "00000100",
             105 => "00000101",
             106 => "00000110",
             107 => "00000111",
             108 => "00001000",
             109 => "00001001",
             110 => "00001010",
             111 => "00001011",
             112 => "00001100",
             113 => "00001101",
             others=> "00000000");
    begin
        wait until req='1';
        if Rwn='1'

```

```

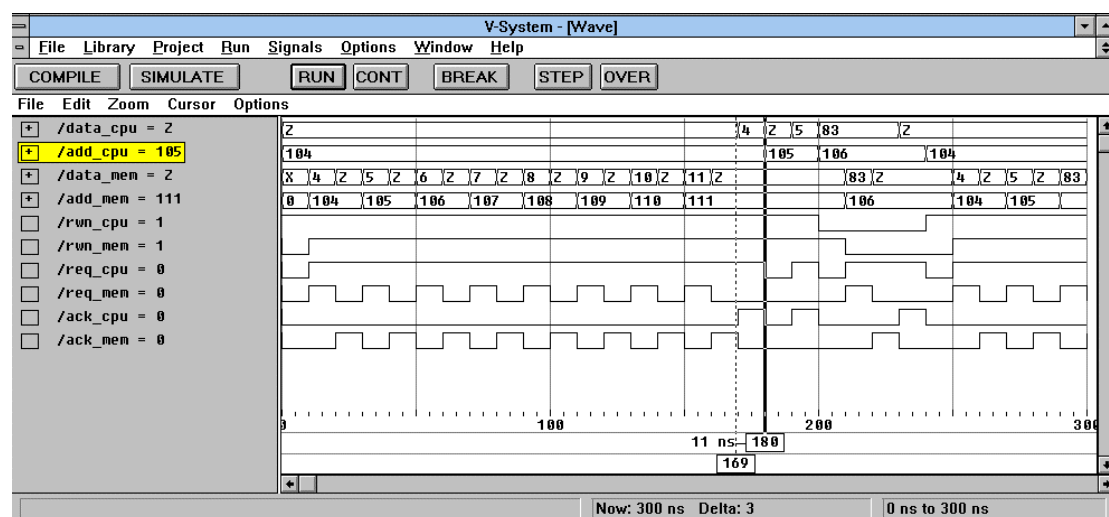
    then -- read from memory
      data <= memory(intval(addr));
      ack <='1' after 10 ns;
      wait until req='0';
      data <= (others => 'Z');
      ack <='0' after 10 ns;
    else -- write to memory
      memory(intval(addr)):=data;
      ack <='1' after 10 ns;
      wait until req='0';
      ack <='0' after 10 ns;
    end if;
  end process;
end behavior; entity test_environment is
end test_environment;

use work.types.all;
use work.cache_help.all;
architecture structure of test_environment is
  component cache
    port ( data_cpu : inout bbyte;
          addr_cpu : in address;
          RWr_cpu  : in bit;
          req_cpu  : in bit;
          ack_cpu  : out bit;

          data_mem : inout bbyte;
          addr_mem : out address;
          RWr_mem  : out bit;
          req_mem  : out bit;
          ack_mem  : in bit);
  end component;
  component cpu
    port (data : inout bbyte;
          addr : out address;
          RWr  : out bit;
          req  : out bit;
          ack  : in bit);
  end component;
  component memory
    port (data : inout bbyte;
          addr : in address;
          RWr  : in bit;
          req  : in bit;
          ack  : out bit);
  end component;
  signal data_cpu,data_mem : bbyte;
  signal add_cpu,add_mem   : address;
  signal rwr_cpu,rwr_mem   : bit;
  signal req_cpu,req_mem   : bit;
  signal ack_cpu,ack_mem   : bit;
begin
  ch : cache port map (data_cpu,add_cpu,rwr_cpu,req_cpu,ack_cpu,
                      data_mem,add_mem,rwr_mem,req_mem,ack_mem);
  cp : cpu port map (data_cpu,add_cpu,rwr_cpu,req_cpu,ack_cpu);
  m  : memory port map (data_mem,add_mem,rwr_mem,req_mem,ack_mem);
end structure;

```

figuur 6.16 Eenvoudige testopstelling overeenkomend met figuur 6.11.

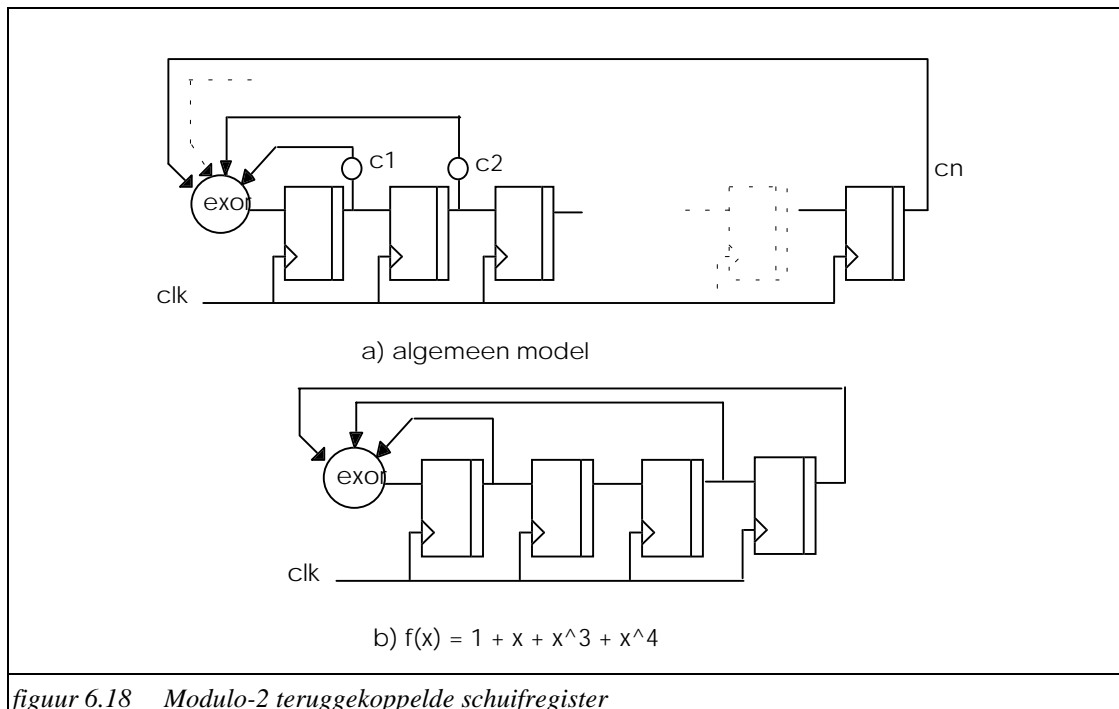


figuur 6.17 Een deel van de simulatieresultaten

6.4 Random generator.

6.4.1 Inleiding.

Vaak zijn er toepassingen waarin random patronen nodig zijn, bijvoorbeeld het on-chip genereren van random testpatronen. Er wordt dan vaak gebruik gemaakt van een pseudo-random generator, waarvan de gegenereerde random patronen reproduceerbaar zijn. Een veel gebruikte techniek voor een dergelijke generator is het gebruik maken van een modulo-2 teruggekoppeld schuifregister (ook wel “linear feedback shift register” (LFSR) genoemd).



figuur 6.18 Modulo-2 teruggekoppelde schuifregister

Het kenmerk van een modulo-2 teruggekoppeld schuifregister is dat het bestaat uit een schuifregister waarvan een aantal uitgangen modulo-2 wordt opgeteld en het resultaat hiervan is teruggekoppeld naar de ingang. De modulo-2 optelling vindt plaats door middel van exclusive or's. In figuur 6.18b is een LFSR gegeven met polynoom $f(x)=1+x^1+x^3+x^4$, dat wil zeggen dat de eerste, derde en vierde register-uitgang teruggekoppeld is. Uitgaande van een begintoestand komt na een aantal keer schuiven dezelfde begintoestand weer terug. Met bovenstaand voorbeeld zijn de onderstaande zes reeksen mogelijk.

0000	1111	0101	0110	0010	0001
		1010	1011	1001	1000
			1101	0100	1100
					1110
					0111
					0011

Het is eenvoudig in te zien elke LFSR een reeks van lengte één heeft, namelijk als de register-inhouden '0' zijn. De maximale haalbare lengte van een reeks is dan ook $2^n - 1$. Dit is bijvoorbeeld het geval indien de polynoom gelijk is aan $f(x)=1+x^1+x^4$. Dan is er

sprake van een ‘maximum lengte reeks generator’. De maximum lengte reeks generatoren worden vaak gebruikt voor pseudo-random generatoren. Het aantal registers bepaalt de maximale lengte van een reeks. Enkele voorbeelden van polynomen voor maximale reeks generatoren zijn (voor hetzelfde aantal registers zijn soms verschillende polynomen mogelijk):

aantal registers	polynoom voor maximum lengte reeks generator
8	$f(x)=1+x^1+x^5+6^1+x^8$
16	$f(x)=1+x^2+x^3+6^5+x^{16}$
32	$f(x)=1+x^{27}+x^{28}+6^{32}$

Interessant is het of het mogelijk is een zodanige beschrijving in VHDL te geven dat eenvoudig de lengte en de polynoom aangepast kunnen worden zonder de beschrijving aan te passen.

6.4.2 Beschrijving van een pseudo-random generator.

```
entity pseude_random is
  generic (fb_position : bit_vector := "1001");
  -- generic for polynoom 1+x+x^4 = "1001"
  --          1+x+x^5+x^6+x^8 = "10001101"
  port ( clk : in bit;
        outp : out bit_vector (1 to fb_position'length):= (others=>'1'));
end pseude_random;

architecture lfsr of pseude_random is
begin
  process
    function xor_n_wide (inp : bit_vector) return bit is
      variable res : bit := '0';
    begin
      for i in inp'range loop
        res := res xor inp(i);
      end loop;
      return res;
    end xor_n_wide;
    variable feedback_reg : bit_vector (1 to fb_position'length) := (others=>'1');
    -- for initialization purposes one position is '1'.
  begin
    wait until clk='1';
    feedback_reg := xor_n_wide(fb_position and feedback_reg) &
                    feedback_reg(1 to fb_position'length-1);
    outp <= feedback_reg;
  end process;
end lfsr;
```

figuur 6.19 VHDL beschrijving van een pseudo-random generator.

Alleen door het aanpassen van de *generic map* in de component instantiatie van de pseudo-random generator kan eenvoudig voor een ander polynoom en het aantal registers worden gekozen.

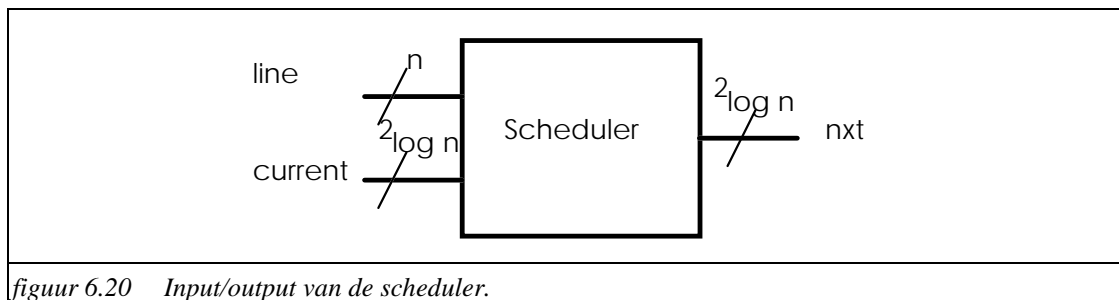
Voor het polynoom $f(x)=1+x+x^4$ ("1001") wordt de volgende reeks gegenereerd:

outp:

1111 → 0111 → 1011 → 0101 → 1010 → 1101 → ... → 1110 → 1111

6.5 Scheduler.

6.5.1 Probleemstelling.



Een digitaal systeem heeft n communicatie-kanalen. De te ontwerpen scheduler heeft als taak het eerstvolgende vrije kanaal aan te wijzen. De scheduler is een combinatorische schakeling met de volgende gegevens:

Ingangen:

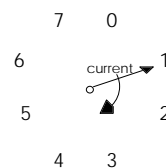
- *line*. Een n bits brede bus. Elk bit correspondeert met een communicatie-kanaal en geeft aan dat deze vrij is door middel van een '1' en bezet is door middel van een '0'. Het aantal kanalen is een macht van 2 (waarschijnlijk 4, 8 of 16).
- *current*. Geeft het huidige kanaal aan dat is geselecteerd. Het kan dus de gehele waarden 0 t/m $n-1$ aannemen. Voor de realisatie is dus een $2^{\log(n)}$ bits brede bus nodig.

Uitgang:

- *nxt*. Geeft het eerstvolgende vrije kanaal aan. Het kan dus de gehele waarden 0 t/m $n-1$ aannemen. Voor de realisatie is dus een $2^{\log(n)}$ bits brede bus nodig.

Functie:

- Uitgaande van de vrije communicatie-kanalen (*line*) en het huidige geselecteerde kanaal (*current*) wordt het eerstvolgende vrije kanaal aangegeven (*nxt*). Met 'eerstvolgende' wordt bedoeld dat de communicatie-kanalen denkbeeldig in een cirkel worden gedacht, welke met de klokrichting mee wordt doorlopen vanaf het huidige geselecteerde kanaal. Zie onderstaand figuur voor 8 kanalen. Indien geen enkel kanaal vrij is, is *nxt* gelijk aan *current*.



Aangezien nog niet precies bekend is uit hoeveel kanalen het systeem uiteindelijk zal bestaan (n), maar het wel een macht van 2 is, is het prettig als eenvoudig voor een ander aantal kanalen kan worden gekozen.

6.5.2 Specificatie in VHDL.

```

entity scheduler is
  generic (line_width : natural := 8);
  port (current : natural range 0 to line_width-1;
        line : bit_vector(0 to line_width-1);
        nxt : out natural range 0 to line_width-1);
end scheduler;

architecture behaviour of scheduler is
begin
  process(current,line)
    variable sel : natural;
    constant nul : bit_vector(0 to line_width-1) := (others => '0');
  begin
    assert nul/=line report "no free channels" severity note;
    for i in line'range loop
      sel := (current + 1 + i) mod line_width;
      exit when line(sel)='1';
    end loop;
    nxt <= sel;
  end process;
end behaviour;

```

figuur 6.21 Specificatie van de scheduler.

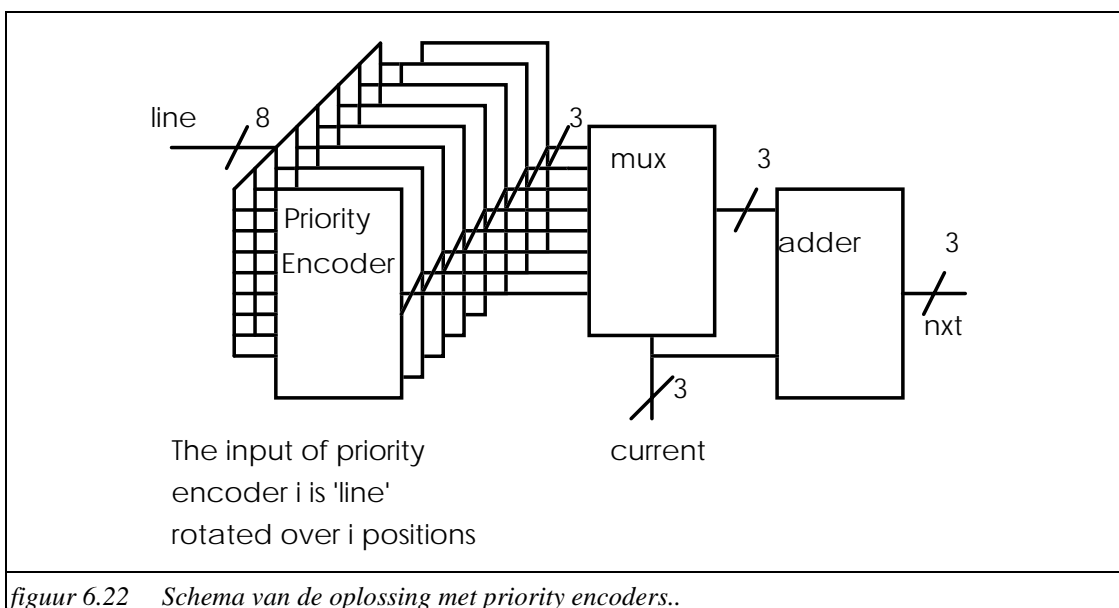
De specificatie van het probleem is verrassend eenvoudig. Maar nog verrassender is het misschien dat het ook nog te synthetiseren is met bijvoorbeeld Synopsys Design Analyzer 3.2a en als Actel als technologie is het resultaat: oppervlak 94 en de maximale vertraging 82.

6.5.3 Implementatie in VHDL.

Hoewel de specificatie al te synthetiseren is, kan er behoefte bestaan zelf te gaan implementeren omdat bijvoorbeeld niet aan het oppervlakte-criterium wordt voldaan. Een drietal mogelijke implementaties komen aan de orde:

1. Een oplossing waarbij gebruik is gemaakt van priority encoders.
2. Een asynchrone oplossing bestaande uit een 1-dimensioneel array.
3. Een oplossing waarbij gebruik is gemaakt van een synthese-tool specifiek concurrent procedure, een PLA.

6.5.3.1 Het gebruik maken van priority encoders.



figuur 6.22 Schema van de oplossing met priority encoders..

Figuur 6.22 geeft schematisch de oplossing weer waarbij gebruik is gemaakt van priority encoders. Het uitgangssignaal van de priority encoder geeft het nummer van eerste ingang aan waar een '1' aanwezig is. De ingang van priority encoder i krijgt een geroteerde versie over i posities aan van input *line*. Vervolgens wordt met de multiplexer de priority encoder uitgang geselecteerd met over *current* geroteerde input *line*. Hiermee is de volgende vrije lijn gevonden, echter bij dit resultaat moet nog de verschuiving over de *current* posities worden opgeteld.

```
architecture priority_encoders of scheduler is
  subtype integer_linewidth is integer range
    0 to line_width-1;
  type int_vector is array (natural range <>)
    of integer_linewidth;
  signal output_enc : int_vector(0 to line_width-1);
  signal y : integer_linewidth;
begin
  priority_encoders:
    for i in 0 to line_width-1 generate
      enc:process(line)
        variable rotate: bit_vector(0 to line_width-1);
      begin
        rotate:=line(i+1 to line_width-1) & line(0 to i);
        for j in rotate'range loop
          output_enc(i) <= (j+1) mod line_width;
          exit when rotate(j)='1';
        end loop;
      end process;
    end generate;

  multiplexer: y <= output_enc(current);

  nxt <= (y + current) mod line_width;
end priority_encoders;
```

figuur 6.23 Een VHDL beschrijving van de implementatie met priority encoders.

De beschrijving in figuur 6.23 komt overeen met het schema gegeven in figuur 6.22. Merk op dat er niet eerst design entities worden beschreven voor de priority encoders, de multiplexer en de opteller, maar dat alles in één architecture is beschreven. Met behulp van het generate statement in figuur 6.23 worden de priority encoders geïnstantieerd. Indien index i gelijk is aan *line_width-1* wordt aan variabele *rotate* het volgende toegekend: *line* (*line_width to line_width-1*) & *line*(0 to *line_width-1*). Synthese met behulp van de Synopsys Design Analyzer versie 3.2a gaf een foutmelding. Merk op dat index *line_width* in variabele *rotate* buiten de grenzen ligt van deze variabele. Echter de slice *line_width to line_width-1* is een null array en voor een dergelijk array geldt dat de grenzen niet binnen het bereik hoeven te vallen. Citaat IEEE Std. 1076-1993, page 88 line 198:

"For the evaluation of a name that is a slice, the prefix and the discrete range are evaluated. It is an error if either of the bounds of the discrete range does not belong to the index range of the prefixing array, unless the slice is a null slice. (The bounds of a null slice need not belong to the subtype of the index.) "

Om toch een synthetiseerbare beschrijving voor Synopsys te krijgen werd het process statement behorend bij deze index afzonderlijk beschreven (figuur 6.24). Het synthese resultaat met Actel als technologie is oppervlak 101 en de maximale vertraging 62.

```

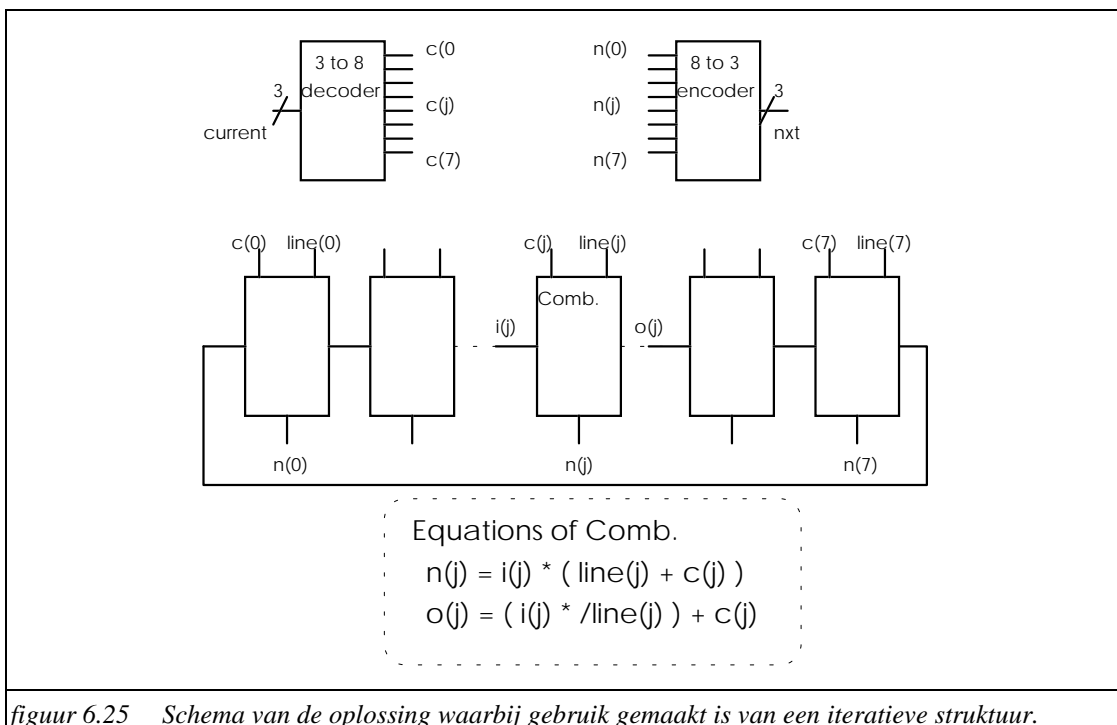
priority_encoders:for i in 0 to line_width-2 generate
  enc:process(line)
    variable shift : bit_vector(0 to line_width-1);
  begin
    shift:=line(i+1 to line_width-1) & line(0 to i);
    for j in shift'range loop
      output_enc(i) <= (j+1) mod line_width;
      exit when shift(j)='1';
    end loop;
  end process;
end generate;

shift_line_width:process(line)
begin
  for j in line'range loop
    output_enc(line_width-1) <= (j+1) mod line_width;
    exit when line(j)='1';
  end loop;
end process;

```

figuur 6.24 Deel van de VHDL beschrijving van de implementatie met priority encoders aangepast voor Synopsys Design Analyzer versie 3.2a..

6.5.3.2 Een iteratieve structuur.



figuur 6.25 Schema van de oplossing waarbij gebruik gemaakt is van een iteratieve structuur.

Een fraaie implementatie van de scheduler is gegeven in figuur 6.25. Er is gebruik gemaakt van een itartieve structuur bestaande uit identieke combinatorische cellen. Elke cel heeft drie ingangen en twee uitgangen. Indien ingang $c(j)$ de waarde '1' heeft moet vanaf de volgende cel worden gezocht naar een vrij kanaal. Daarvoor wordt het uitgangssignaal $o(j)$ gebruikt. Indien dit signaal '1' is moet het volgende blok, dat dit als ingangssignaal $i(j+1)$ binnenkrijgt, nagaan of $line(j+1)$ de waarde '1' heeft. Indien $line(j+1)$ niet '1' is moet uiteraard het uitganssignaal $o(j+1)$ '1' worden, etc.. Indien $line(j+1)$ wel '1' is hoeft niet verder te worden gezocht en dus krijgt uitgang $o(j+1)$ de waarde '0'. Voor cel $j+1$ geldt dan dat $n(j+1)$ de waarde '1' moet krijgen en de overige uitgangen n de waarde '0'.

Samengevat:

- Uitgang $o(j)$. Deze uitgang wordt '1' indien:
 1. $i(j) = '1'$ en $line(j) = '0'$, of
 2. $c(j) = '1'$.
 De laatste term is nodig om het zoeken op de juiste plaats te starten. Dit geeft de formule: $o(j) \leq (i(j) \text{ AND NOT } line(j)) \text{ OR } c$;
- Uitgang $n(j)$. Deze uitgang wordt '1' indien:
 1. $i(j) = '1'$ en $line(j) = '1'$, of
 2. $i(j) = '1'$ en $c(j) = '1'$.
 De laatste term is nodig voor het geval dat geen enkel ander kanaal vrij is. Dit geeft de formule: $n(j) \leq i(j) \text{ AND } (line(j) \text{ OR } c(j))$;

Voor de regelmatige structuur van de 1-dimensionale array kan een generate statement worden gebruikt met de combinatorische logica van een cel als component of als proces. In figuur 6.26 is een alternatief gegeven waarbij het 1-dimensionaal array met drie concurrent signal assignment statements is beschreven. Verder zijn nog twee processen toegevoegd voor de decoder en de encoder. De decoder beschrijving lijkt veel gemakkelijker beschreven te kunnen worden door het volgende concurrent signal assignment statement:

$c \leq (\text{current} \Rightarrow '1', \text{others} \Rightarrow '0');$

Echter dit is niet legaal VHDL omdat een aggregate met meerdere keuzes alleen is toegestaan voor een statische keuzes en dat geldt niet voor *current*.

Het synthese resultaat met Synopsys Design Analyzer 3.2a en Actel als technologie is: oppervlak 30 en de maximale vertraging 78.

```
architecture one_dim_array of scheduler is
  signal i, c, n, o : bit_vector(0 to line_width-1);
begin
chain:  n <= i and ( line or c );
        o <= (i and not line) or c;
        i <= o(line_width-1) & o(0 to line_width-2);

  inp:process(current)
  begin
    c <= (others=>'0');
    c(current) <= '1';
  end process;

  outp:process(n)
  begin
    for i in n'range loop
      nxt <= i;
      exit when n(i)='1';
    end loop;
  end process;
end one_dim_array;
```

figuur 6.26 Implementatie van de scheduler met een iteratieve structuur.

6.5.3.3 Het gebruik maken van een pla.

Een mogelijkheid is het gebruik maken van een PLA waarin voor een vast aantal kanalen alle mogelijkheden worden opgesomd. Voor n is 8 zijn er dus totaal 11 ingangssignalen en dus $2^{11} = 2048$ mogelijke ingangspatronen, voorwaar geen aantrekkelijk vooruitzicht. Soms heeft een tool een aantal procedures die een pla beschrijving aanzienlijk kan vereenvoudigen. In figuur 6.27 is gebruik gemaakt van de procedure *pla_table*. In het gebruikte type *vlbit* heeft 'x' de betekenis van don't care. Hiermee is het mogelijk het probleem te beschrijven met slechts 64 ingangspatronen. Elke combinatie van in- en uitgang is gegeven in de constante *tbl*. Voorbeeld:

B"XX001XXX_001_100",

Elke rij van de tabel is opgebouwd uit drie delen:

1. De meest linkse 8 bits geven aan of een kanaal vrij is: XX001XXX
2. De middelste drie bits geven de huidige kanaal: 001
3. Meest rechts zijn de drie bits die het uitgangssignaal aangeven. Het huidige kanaal is 1 (001). Indien de kanalen 2 en 3 bezet ('0') zijn en kanaal 4 vrij ('1') is, is het vrij of bezet zijn van de overige kanalen niet van belang ('x') en is het uitgangssignaal dus kanaal 4 (100).

Het synthesesresultaat is: oppervlak 40 en de maximale vertraging 24.

```

library synth;
use synth.stdsynth.all;
entity scheduler1 is
  port (current : in vlbit_vector(2 downto 0);
        line : in vlbit_vector(0 downto 7);
        nxt : out vlbit_vector(2 downto 0));
end scheduler1;

architecture pla of scheduler1 is
  constant tbl : vlbit_2d(0 to 63,13 downto 0) := (
    B"X1XXXXXX_000_001",
    B"X01XXXXX_000_010",
    B"X001XXXX_000_011",
    B"X0001XXX_000_100",
    B"X00001XX_000_101",
    B"X000001X_000_110",
    B"X0000001_000_111",
    B"X0000000_000_000",

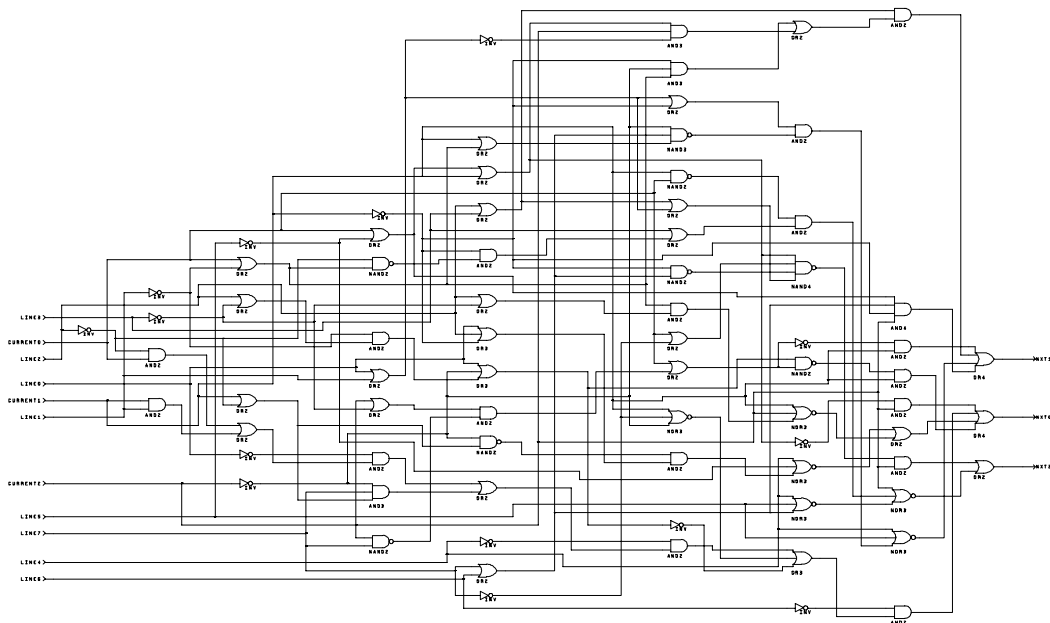
    B"XX1XXXXX_001_010",
    B"XX01XXXX_001_011",
    B"XX001XXX_001_100",
    B"XX0001XX_001_101",
    B"XX00001X_001_110",
    B"XX000001_001_111",
    B"1X000000_001_000",
    B"0X000000_001_001",

    B"XXX1XXXX_010_011",
    B"XXX01XXX_010_100",
    B"XXX001XX_010_101",

    ....

    B"0000000X_111_111");
  signal input : vlbit_vector(10 downto 0);
begin
  input <= line & current;
  pla_table(input,nxt,tbl);
end pla;

```



figuur 6.27 Implementatie met behulp van de procedure pla_table.

6.5.4 Samenvatting van de resultaten van de scheduler.

- slecht + goed ++ zeer goed	ontwerp- tijd	documen- tatie	fout- gevoeligheid	syntheseresultaat	
				oppervlak	vertraging
specification	++	++	++	94	82
priority encoders	+	+	+	101	62
1-dimensional array	++	+	+	30	78
PLA tabel	-	-	-	40	24

figuur 6.28 De resultaten van de scheduler vergeleken.

In figuur 6.28 zijn de resultaten van de specificatie en de implementatie alternatieven voor de scheduler vergeleken. Hierbij moet wel worden opgemerkt dat de instellingen van de tools niet zijn aangepast.

6.6 Simulatiemodel van een flipflop.

6.6.1 Inleiding.

Soms kom je VHDL-beschrijvingen tegen die niet echt leesbaar geschreven zijn. In het ontwerptraject van een systeem is dit uiteraard minder wenselijk, maar als het gaat om een simulatiemodel van een bestaande component speelt een ander criterium een belangrijke rol: de simulatiesnelheid. De essentie van deze paragraaf is niet een simulatiemodel te beschrijven met de daarbij behorende set-up tijd, hold tijd en delay tijd, maar te concentreren op de simulatiesnelheid.

```
entity flipflop is
  port (d, clk, enable : in bit;
        q               : out bit);
end flipflop;

architecture classic of flipflop is
begin
  process
  begin
    wait until clk='1';
    if enable='1'
    then q <= d;
    end if;
  end process;
end classic;

architecture fast_simulation of flipflop is
begin
  process
  begin
    busy_loop:loop
      wait until enable='1';
      exit when clk='1' and clk'event;
      wait until clk='1';
      exit when enable='1';
    end loop busy_loop;
    q <= d;
  end process;
end fast_simulation;
```

figuur 6.29 Twee beschrijving van een flipflop met hetzelfde gedrag.

In deze paragraaf zal aan de hand van een flipflop met een enable ingang de consequentie voor de simulatiesnelheid worden gegeven. Entity flipflop(classic), figuur 6.29, geeft een veel gebruikte methode om een dergelijk flipflop te beschrijven. Opvallend is het dat indien het enable signaal meestal laag is onnodig vaak het proces wordt opgestart waardoor kostbare simulatietijd verloren gaat. Entity flipflop(fast_simulation), gebaseerd op een beschrijving van Joseph Pick³, heeft exact hetzelfde gedrag als flipflop(classic). Maar deze beschrijving zal, indien het enable signaal meestal '0' is, de simulatiesnelheid aanzienlijk verbeteren.

6.6.2 Vergelijking van de simulatiesnelheid.

Om ook de overhead te kunnen meewegen in de vergelijking tussen de entiteiten flipflop(classic) en flipflop(fast_simulation) is nog de volgende entiteit flipflop(empty) aanwezig:

```
architecture empty of flipflop is
begin
end empty;
```

³ Joseph Pick, "VHDL Coding Tricks and Techniques", Tutorial at the Fall VIUF'95 conference 1995, Boston.


```

entity test_env is
end test_env;
architecture structure of test_env is
  component flipflop
    port (d, clk, enable : in bit;
          q               : out bit);
  end component;
  constant lgt : positive := 20;
  signal clk, enable, d : bit;
  signal q : bit_vector(1 to lgt);
begin
  clk <= not clk after 1 ns;
  -- enable <= not enable after 1 ms;
  -- d <= not d after 10 ms;
  gen:for i in 1 to lgt generate
    fl : flipflop port map (d,clk,enable,q(i));
  end generate;
end structure;

configuration fast of test_env is
  for structure
    for gen
      for fl:flipflop use entity work.flipflop(fast_simulation);
    end for;
  end for;
end fast;

configuration normal of test_env is
  for structure
    for gen
      for fl:flipflop use entity work.flipflop(classic);
    end for;
  end for;
end normal;

configuration no_flipflops of test_env is
  for structure
    for gen
      for fl:flipflop use entity work.flipflop(empty);
    end for;
  end for;
end no_flipflops;

```

figuur 6.30 De testopstelling voor het vergelijken van de simulatiesnelheid.

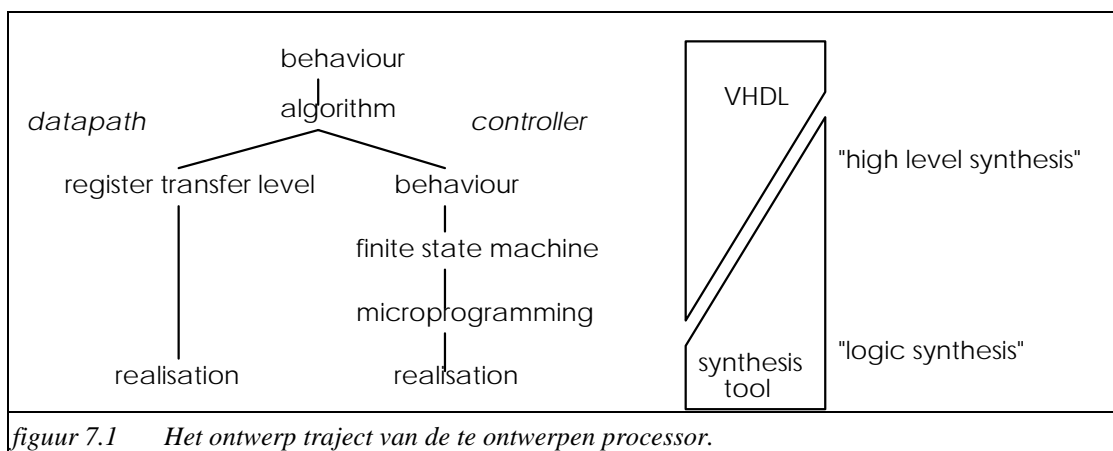
In de testopstelling is door middel van het generate statement een twintigtal flipflops geïnstantieerd. Vervolgens is een simulatie run van dezelfde duur uitgevoerd, in de Vsystem/windows versie 4.2e omgeving. De resultaten zijn:

configuratie	simulatietijd	relatief t.o.v. no_flipflops
no_flipflops	25 sec.	1
fast	28 sec.	1.12
normal	205 sec.	8.2

De minder leesbare beschrijving bevordert in deze extreme situatie (het enable signaal is voortdurend '0') de simulatiesnelheid aanzienlijk.

7. Het ontwerpen van een eenvoudige processor.

7.1 Inleiding.

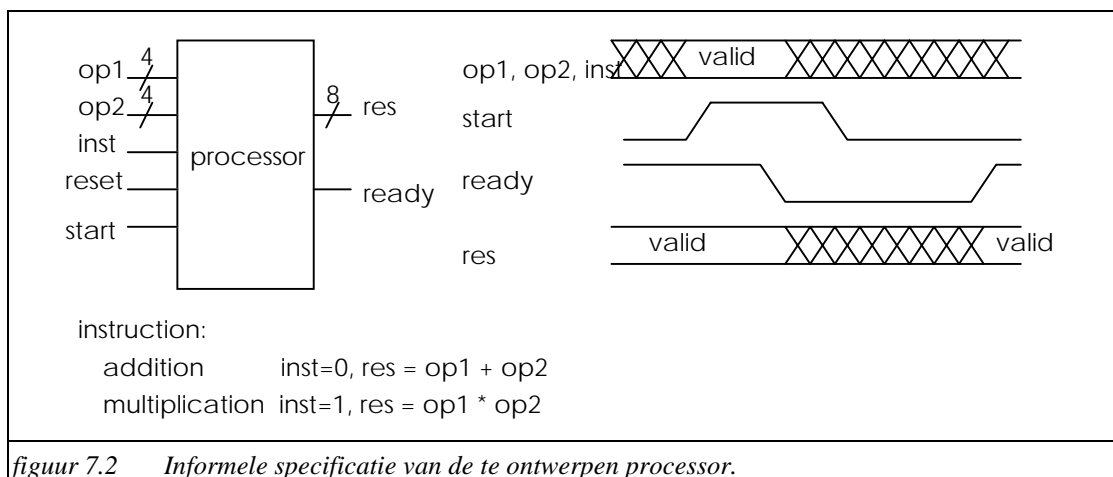


figuur 7.1 Het ontwerp traject van de te ontwerpen processor.

Hoe kan een complex digitaal systeem worden ontworpen? Het is de bedoeling van dit hoofdstuk om aan de hand van een zeer eenvoudige processor een compleet ontwerptraject te tonen. Een eenvoudige processor heeft als nadeel dat voor het probleem op zich (veel) te veel detail zal worden beschreven in VHDL. Bovendien bepaalt ook de kracht van de synthese-tool tot welk niveau van beschrijven in VHDL afgedaald moet worden. Het ontwerptraject is gegeven in figuur 7.1.

Het is een voordeel dat de ontwerpstappen in dezelfde taal zijn beschreven omdat elke ontwerpstep dan tegen het gewenste gedrag gesimuleerd kan worden. De daarvoor ontwikkelde testomgeving, voor deze processor, is eveneens gegeven.

7.2 Specificatie van de processor.



figuur 7.2 Informele specificatie van de te ontwerpen processor.

Figuur 7.2 geeft een informele specificatie van de te ontwerpen processor. Voor veel informele specificaties geldt echter dat deze incompleet of dubbelzinnig zijn bevatten. In de gegeven informele specificatie ontbreekt bijvoorbeeld de representatie van de operanden, maar ook of de reset actief is bij een '0' of een '1'.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity processor is
    port (op1,op2      : unsigned(3 downto 0);
          inst,start   : std_logic;
          reset        : std_logic;
          ready        : out std_logic;
          res          : out unsigned(7 downto 0));
end processor;

architecture behaviour of processor is
    signal rdy_int : std_logic := '1';
begin
    alu:process
        constant delay0,delay1 : time := 20 ns;
        constant add           : std_logic := '0';
        constant mul           : std_logic := '1';
        constant zero3         : unsigned(2 downto 0) := "000";
        constant dontcare8     : unsigned(7 downto 0) := (others => '-');
    begin
        wait until rising_edge(start) or reset='1';

        case inst is
            when add => res <= zero3 & (op1 + ('0' & op2));
            when mul => res <= op1 * op2;
            when others => res <= dontcare8;
        end case;

        if reset='1'
            then res <= dontcare8; rdy_int <='1' after delay1;
        else rdy_int <= '0' after delay0;
            wait until falling_edge(start) or reset='1';
            if reset='1' then res <= dontcare8; end if;
            rdy_int <= '1' after delay1;
        end if;
    end process;

    ready <= rdy_int;

    stable_inputs:process(op1,op2,inst)
    begin
        assert start='0' or rdy_int='0'
            report "op1,op2, and should be stable" severity warning;
    end process;

    handshake_violation:process
    begin
        wait until rising_edge(start);
        assert rdy_int='1' report "handshake violation" severity warning;
        wait until falling_edge(start);
        assert rdy_int='0' report "handshake violation" severity warning;
    end process;
end behaviour;

```

figuur 7.3 Formele specificatie van de processor.

Het gedrag van de processor (figuur 7.3) is beschreven met een drietal processen, met de labels:

- Handshake_violation
- Stable_inputs
- Alu

De processen *handshake_violation* en *stable_inputs* signaleren een afwijken van het afgesproken communicatieprotocol. Het ingangssignaal op de data-ingang moet stabiel zijn gedurende de tijd dat het signaal *start* '1' wordt tot en met het '0' worden van het uitgangssignaal *ready*.

De functie van de processor is beschreven met het proces *alu*. Het gewenste gedrag van het te ontwerpen systeem kan meestal het beste met zo weinig mogelijk processen worden beschreven. Bij een toename van het aantal processen die elkaar beïnvloeden is het moeilijker het overzicht over het totaal te behouden. Het is gebleken dat vooral een beginnende VHDL gebruiker te veel in afzonderlijke processen een probleem specificeert. Binnen een procesbeschrijving kan het probleem weer worden opgedeeld

in subproblemen waarvoor gebruik gemaakt kan worden van procedures en functies. Veel voorkomende functies en dergelijke kunnen, mits goed gedocumenteerd, centraal beschikbaar worden gesteld.

Voor deze eenvoudige processor zijn slechts de operaties *optellen* en *vermenigvuldigen* van unsigned vectoren nodig. Er is een IEEE standaard, std. 1076.1-1997, met de package *numeric_std*. Deze package bevat de types unsigned en signed (2-complementrepresentatie), met veelvoorkomende operaties. Hiervan is ook in de gedragsbeschrijving gebruik gemaakt.

De essentie van de functie van de processor is beschreven met de vijf regels beginnend met de regel "*case inst is ..*". Door de reset conditie wordt de beschrijving minder leesbaar. De reset onderbreekt de aan de gang zijnde actie op een abrupte wijze. Dit geeft ook in VHDL meestal een ondoorzichtige beschrijving. In figuur 7.4 is een alternatieve beschrijving gegeven waarbij gebruik gemaakt is van een geneste loop statement. De binnenste loop, voorzien van het label *functional*, beschrijft het normale gedrag. Alleen ten gevolge van een reset wordt deze loop beëindigd.

Hoewel het de intentie is van een specificatie om geen implementatie te beschrijven zou de specificatie kunnen suggereren dat gebruik gemaakt moet worden van de twee componenten *optellen* en *vermenigvuldigen*. Het is dan ook belangrijk dat de naam van architecture goed gekozen wordt om dit soort misverstanden te voorkomen.

```
architecture behaviour_alternative of processor is
    signal rdy_int : std_logic := '1';
begin
    alu:process
        constant delay0,delay1 : time := 20 ns;
        constant add           : std_logic := '0';
        constant mul           : std_logic := '1';
        constant zero3         : unsigned(2 downto 0) := "000";
        constant dontcare8     : unsigned(7 downto 0) := (others => '-');
    begin
        resetting:loop
            -- resetting
            res <= dontcare8; rdy_int <= '1' after delay1;
            wait until falling_edge(reset);

            functional:loop
                wait until rising_edge(start) or reset='1';
                exit resetting when reset='1';
                case inst is
                    when add => res <= zero3 & (op1 + ('0' & op2));
                    when mul => res <= op1 * op2;
                    when others => res <= dontcare8;
                end case;
                rdy_int <= '0' after delay0;
                wait until falling_edge(start) or reset='1';
                exit resetting when reset='1';
                rdy_int <= '1' after delay1;
            end loop functional;
        end loop resetting;
    end process;

    ready <= rdy_int;

    stable_inputs:process(op1,op2,inst)
    begin
        assert start='0' or rdy_int='0'
            report "op1,op2, and should be stable" severity warning;
    end process;

    handshake_violation:process
    begin
        wait until rising_edge(start);
        assert rdy_int='1' report "handshake violation" severity warning;
        wait until falling_edge(start);
        assert rdy_int='0' report "handshake violation" severity warning;
    end process;
end behaviour_alternative;
```

figuur 7.4 Alternatieve beschrijving voor de specificatie van de processor.

7.3 Hoe moet het gewenste gedrag geïmplementeerd worden?

Een groot aantal implementaties zijn mogelijk om het gewenste gedrag te verkrijgen. Echter de keuze wordt beperkt door gestelde randvoorwaarden aan de snelheid, het oppervlak en de vermogensdissipatie. Eerst wordt een algoritme bepaald waarmee het gewenste gedrag geïmplementeerd kan worden. Dit wordt beschreven in VHDL en vergeleken met de specificatie door middel van simulatie. In de volgende paragraaf wordt hieraan aandacht besteed. Het ontwerptraject suggereert alleen een *top-down* aanpak. Echter men moet bekend zijn met de technologie en de gebruikte synthese-tools.

De meeste synthese-tools hebben geen enkel probleem met een *opteller*. Er is verondersteld dat de tools dit dan ook naar tevredenheid doen; rekening houdend met de randvoorwaarden. Voor het vermenigvuldigen wordt gekozen voor het *shift/add* algoritme. De VHDL beschrijving van de algoritme is gegeven in figuur 7.5. In de procesbeschrijving is een functie *multiply* gebruikt.

```

architecture algorithm of processor is
  signal rdy_int : std_logic := '1';
begin
  alu:process
    function multiply (a,b:unsigned(3 downto 0))
      return unsigned is
    -- algorithm is based on a repeating addition and shift operation
    function and4(x:std_logic; y:unsigned(3 downto 0))
      return unsigned is
    -- the logical and of 'x' with all bits of 'y'
    variable x4 : unsigned(3 downto 0) := (others => x);
    begin
      return x4 and y;
    end and4;
    variable b_int, res : unsigned(3 downto 0);
    variable high      : unsigned(3 downto 0) := "0000";
    variable sum        : unsigned(4 downto 0);
    begin
      b_int:=b;
      for i in 0 to 3 loop
        res:=and4(b_int(0),a);
        sum:=high + ('0' & res);
        b_int:=sum(0) & b_int(3 downto 1);
        high:=sum(4 downto 1);
      end loop;
      return high & b_int;
    end multiply;
    constant delay0,delay1 : time := 20 ns;
    constant add           : std_logic := '0';
    constant mul           : std_logic := '1';
    constant zero3         : unsigned(2 downto 0) := "000";
    constant dontcare8     : unsigned(7 downto 0) := (others => '-');
  begin
    resetting:loop
      -- resetting
      res <= dontcare8; rdy_int <= '1' after delay1;
      wait until falling_edge(reset);

      functional:loop
        wait until rising_edge(start) or reset='1';
        exit resetting when reset='1';
        case inst is
          when add => res <= zero3 & (op1 + ('0' & op2));
          when mul => res <= multiply(op1,op2);
          when others => res <= dontcare8;
        end case;
        rdy_int <= '0' after delay0;
        wait until falling_edge(start) or reset='1';
        exit resetting when reset='1';
        rdy_int <= '1' after delay1;
      end loop functional;
    end loop resetting;
  end process;

  ready <= rdy_int;
end algorithm;

```

figuur 7.5 Algoritme voor de te ontwerpen processor.

7.3.1 De testomgeving.

Om een ontwerpstep te verifiëren zou gebruik gemaakt kunnen worden gemaakt van een formele methode die de twee beschrijvingen vergelijkt. Dergelijke methoden zijn in ontwikkeling echter nog ongeschikt voor complexe beschrijvingen. Daarom wordt voor simulatie gekozen om een ontwerpstep te testen. Hierbij wordt een testomgeving opgezet die de ontwerpstep vergelijkt tegen de specificatie. De vergelijking wordt dus niet door de ontwerper zelf uitgevoerd omdat dit het ontwerpproces foutgevoeliger zal maken. In het algemeen hoeft een ontwerpstep niet op dezelfde tijdstippen hetzelfde gedrag te vertonen. Op het moment dat van de beide beschrijvingen het signaal *ready* '1' is moeten de uitgangssignalen gelijk zijn. In figuur 7.6 is een testomgeving beschreven die op dit moment nagaat of het gedrag gelijk is. Door gebruik te maken

van een configuratie (figuur 7.7) kan dezelfde testomgeving worden gebruikt voor de verschillende ontwerpstappen.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity test_environment is
end test_environment;

architecture structure of test_environment is
  component processor
    port (op1,op2      : unsigned(3 downto 0);
          inst,start   : std_logic;
          reset        : std_logic;
          ready        : out std_logic;
          res          : out unsigned(7 downto 0));
  end component;

  type operands is array(0 to 3) of unsigned(3 downto 0);
  constant operand1 : operands :=
    ("0111","0101","1001","1111");
  constant operand2 : operands :=
    ("1000","0001","1101","1011");
  type instructions is array (0 to 3) of std_logic;
  constant instruction : instructions := "0101";
  signal reset,inst,start,rdy1,rdy2 : std_logic;
  signal op1,op2 :unsigned(3 downto 0);
  signal res1,res2 : unsigned(7 downto 0);
begin
  process
  begin
    start<='0'; reset<='1';
    wait for 100 ns;
    for i in 0 to 3 loop
      op1<=operand1(i); op2<=operand2(i);inst<=instruction(i);
      reset<='0'; start<='1' after 10 ns;
      -- synchronise both processors
      wait until rdy1='0' and rdy2='0';
      start<='0' after 10 ns;
      -- synchronise both processors
      wait until rdy1='1' and rdy2='1';
      -- are the results the same?
      assert res1=res2 report "resultaten zijn niet gelijk" severity error;
    end loop;
  end process;

  gdrg:processor port map(op1,op2,inst,start,reset,rdy1,res1);
  ontw:processor port map(op1,op2,inst,start,reset,rdy2,res2);
end structure;

```

figuur 7.6 De testomgeving.

```

configuration cnf_algorithm of test_environment is
  for structure
    for gdrg:processor use entity work.processor(behaviour_alternative); end for;
    for ontw:processor use entity work.processor(algorithm); end for;
  end for;
end cnf_algorithm;

```

figuur 7.7 Configuratie voor het testen van een ontwerpstep.

Het opzetten van een goede testomgeving is niet eenvoudig. Welke stimuli moeten worden gekozen? Op welk moment moeten de uitgangssignalen gelijk zijn? De testomgeving zoals hierboven is beschreven lijkt dit goed op te lossen. Deze beschrijving is toch onvoldoende. Indien een ontwerpstep ten gevolge van een fout het uitgangssignaal *ready* nooit '1' maakt zal de testomgeving continu wachten, maar dit niet melden aan de ontwerper. Dit probleem kan eenvoudig worden opgelost door het volgende proces toe te voegen:

```

PROCESS
BEGIN
    WAIT ON rdy2 FOR 300 ns;
    ASSERT rdy2'EVENT REPORT "implementation is not responding"
        SEVERITY warning;
END PROCESS;

```

De tijdsduur, 300 ns in dit voorbeeld, moet zodanig worden gekozen dat zo min mogelijk ten onrechte een waarschuwing wordt gegeven. De tijdsduur wordt bepaald door de langst durende opeenvolging van operaties zonder communicatie met de omgeving.

7.4 De opdeling in een data-pad en een besturing.

Zoals al eerder is opgemerkt is verondersteld dat voor de opteller al een geschikte implementatie aanwezig is. Het algoritme voor de vermenigvuldiger veronderstelt het herhaald gebruiken van een opteller. Dit kan dezelfde opteller zijn als die nodig is voor de instructie optellen. Ook nu zijn er nog vrijheidsgraden, bijvoorbeeld wat is een goede keuze voor de interne breedte van de data-bus: 1, 2 of 4 bits breed. Daarbij moet wel worden opgemerkt dat een kleinere breedte van de bus het data-pad misschien wel kleiner zal maken maar dat de besturing complexer en dus waarschijnlijk groter wordt. In dit ontwerpvoorbeeld wordt de interne breedte gelijk gehouden aan die van de externe bus; 4 bits breed. Figuur 7.8 geeft de opdeling in een data-pad en een besturing.

Vaak is zal het aantal controle-signalen van de besturing naar het data-pad toe groot zijn. Ook is het wenselijk om de codering van de controle-signalen niet op dit moment al vast te leggen. Het aantal controle-signalen kan immers gedurende het ontwerpproces nog altijd variëren. Verder is het voor de leesbaarheid en onderhoudbaarheid van de VHDL beschrijving wenselijk dat logische namen voor de controle-signalen worden gebruikt. Dit is bereikt door in een package *control_names* (figuur 7.21) een opsomming te geven van deze logische namen. Indien op een later tijdstip een controle-signaal moet worden toegevoegd is het voldoende alleen dit opsommingstype aan te passen.

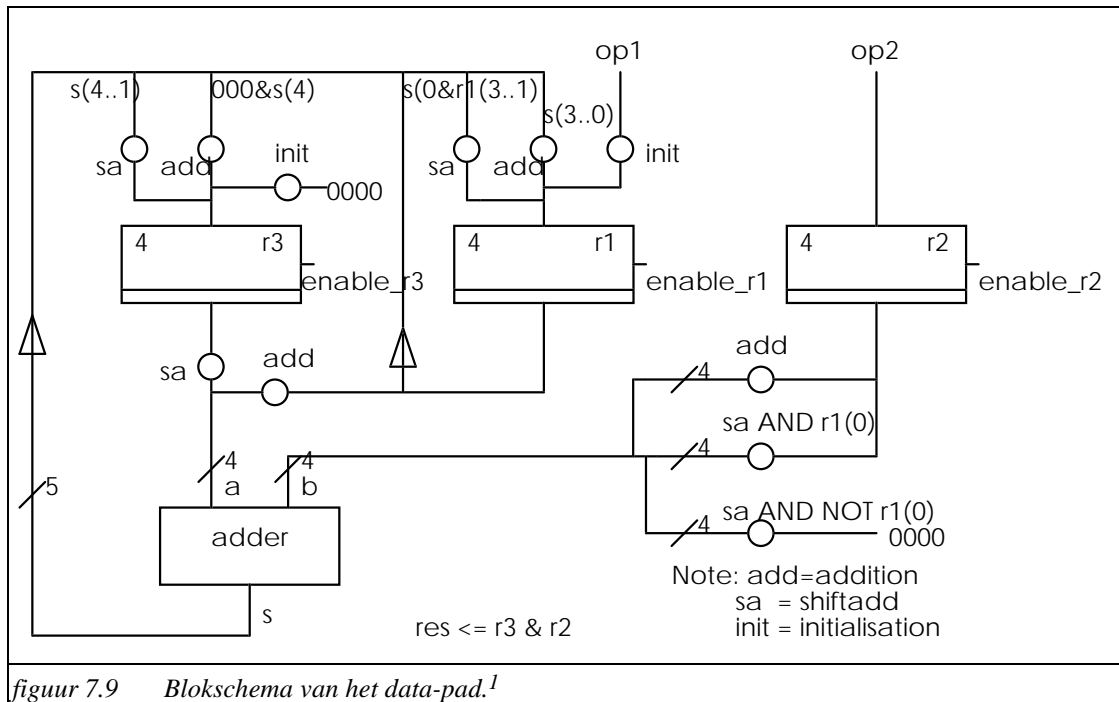
```

use work.control_names.all;
architecture datapath_controller of processor is
    signal control : control_bus;
    signal clk      : std_logic := '0';
    component datapath
        port (op1,op2 : unsigned(3 downto 0);
              control : control_bus;
              clk      : std_logic;
              res      : out unsigned(7 downto 0));
    end component;
    component controller
        port (inst : std_logic;
              start : std_logic;
              clk   : std_logic;
              reset : std_logic;
              control : out control_bus;
              ready  : out std_logic);
    end component;
begin
    dp:datapath
        port map (op1,op2,control,clk,res);
    ct:controller
        port map (inst,start,clk,reset,control,ready);
    clk <= not clk after 5 ns;
end datapath_controller;

```

figuur 7.8 Opdeling in een data-pad en een besturing.

7.4.1 Het data-pad.



figuur 7.9 Blokschema van het data-pad.¹

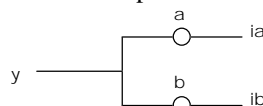
De beschrijving van het data-pad (figuur 7.9) is een *Register Transfer Level* (RTL) beschrijving. De status van een dergelijk blokschema heeft een informeel karakter. Indien er een verschil aanwezig is tussen het blokschema en de VHDL beschrijving (figuur 7.10) is de laatst genoemde correct. Verder is ook volledigheid van een blokschema niet nagestreeft, zo ontbreekt bijvoorbeeld de globale kloklijn. De essentie van een blokschema is gelegen in een stuk documentatie ter verduidelijking van de VHDL beschrijving van het data-pad.

Als de controle-signalen *init*, *enable_r1* en *enable_r2* '1' zijn, worden de operanden ingelezen op een opgaande flank van het kloksignaal. Indien vervolgens de controle-signalen *add(ition)*, *enable_r2*, en *enable_r3* '1' zijn worden de ingelezen operanden bij elkaar opgeteld en het resultaat wordt ingelezen in de registers *r3* en *r1*.

¹ Betekenis van de gebruikte symbolen:

- Dubbel onderstreept.
Dit zijn flank gevoelige registers. De globale kloklijnen worden niet getoond.
- Overige blokken.
Combinatoriek.
- Cirkels

Cirkels beschrijven poorten. Het gedrag van een poort kan niet op zichzelf worden beschouwd. De 'uitgangslijnen' van de poorten komen samen in een knooppunt. Indien exact één poortconditie '1' is, wordt het signaal aan de ingang van deze poort doorgegeven aan het knooppunt. Zijn er twee of meer condities '1' of geen enkele conditie is '1' dan is het signaal op het knooppunt *don't care*. In feite wordt geabstraheerd van de realisatie. Uiteindelijk kan dit bijvoorbeeld worden gerealiseerd met een multiplexer.



In dit schema is de waarde van knooppunt *y* gelijk aan de waarde van ingang *ia* indien alleen het controle-signaal *a* gelijk is aan '1'.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.control_names.all;
entity datapath is
  port (op1,op2 : unsigned(3 downto 0);
        control : control_bus;
        clk      : std_logic;
        res      : out unsigned(7 downto 0));
end datapath;

architecture rtl of datapath is
  constant zero3 : unsigned(2 downto 0) := "000";
  constant zero4 : unsigned(3 downto 0) := "0000";
  constant dontcare4 : unsigned(3 downto 0) := "----";
  signal s : unsigned(4 downto 0);
  signal a,b,ir1,ir2,ir3 : unsigned(3 downto 0);
  signal r1,r2,r3 : unsigned(3 downto 0);
begin
  ir1 <= s(0) & r1(3 downto 1) when control(shift_add)='1' else
        s(3 downto 0) when control(addition)='1' else
        op1 when control(init)='1' else
        dontcare4;
  ir2 <= op2;
  ir3 <= s(4 downto 1) when control(shift_add)='1' else
        zero3 & s(4) when control(addition)='1' else
        zero4 when control(init)='1' else
        dontcare4;
  a <= r3 when control(shift_add)='1' else
        r1 when control(addition)='1' else
        dontcare4;
  b <= r2 when control(addition)='1' else
        r2 when control(shift_add)='1' and r1(0)='1' else
        zero4 when control(shift_add)='1' and r1(0)='0' else
        dontcare4;
  s <= ('0' & a) + b;
  res <= r3 & r1;

  registers:process
  begin
    wait until rising_edge(clk);
    if control(enable_r1)='1' then r1<=ir1; end if;
    if control(enable_r2)='1' then r2<=ir2; end if;
    if control(enable_r3)='1' then r3<=ir3; end if;
  end process;
end rtl;

```

figuur 7.10 VHDL beschrijving van het data-pad.

In de toelichting bij het blokschema van het data-pad (figuur 7.9) is aangegeven dat bij het samenkomen van poorten slechts één van de condities van de poorten '1' is. Indien er meerdere condities '1' zijn mag het resultaat willekeurig zijn. Echter in de beschrijving van figuur 7.10 is al deze ontwerpvrijheid al gedeeltelijk gebruikt. Ingang *a* van de opteller kan van register *r3* of register *r1* data ontvangen. Echter indien de beide controle signalen '1' zijn wordt aan ingang *a* de data van register *r3* doorgegeven en geen willekeurige waarde. Deze ontwerpvrijheid is nog wel beschreven in de VHDL beschrijving van figuur 7.11. Of een synthese-tool deze ontwerpvrijheid optimaal zal gebruiken is nog maar de vraag.

```

architecture rtl_alternative of datapath is
  constant zero3      : unsigned(2 downto 0) := "000";
  constant zero4      : unsigned(3 downto 0) := "0000";
  constant dontcare4   : unsigned(3 downto 0) := "----";
  signal s             : unsigned(4 downto 0);
  signal a,b,ir1,ir2,ir3 : unsigned(3 downto 0);
  signal r1,r2,r3      : unsigned(3 downto 0);
  subtype std_logic3 is std_logic_vector(2 downto 0);
  subtype std_logic2 is std_logic_vector(1 downto 0);
begin

  with std_logic3'(control(shift_add) & control(addition) & control(init)) select
    ir1 <= s(0) & r1(3 downto 1) when "100",
          s(3 downto 0)          when "010",
          op1                     when "001",
          dontcare4               when others;

  ir2 <= op2;

  with std_logic3'(control(shift_add) & control(addition) & control(init)) select
    ir3 <= s(4 downto 1) when "100",
          zero3 & s(4)   when "010",
          zero4          when "001",
          dontcare4      when others;

  with std_logic2'(control(shift_add) & control(addition)) select
    a <= r3      when "10",
        r1       when "01",
        dontcare4 when others;

  with std_logic3'(control(shift_add) & control(addition) & r1(0)) select
    b <= r2      when "010" | "011",
        r2       when "101",
        zero4     when "100" | "110",
        dontcare4 when others;

  s <= ('0' & a) + b;

  res <= r3 & r1;

  registers:process
  begin
    wait until rising_edge(clk);
    if control(enable_r1)='1' then r1<=ir1; end if;
    if control(enable_r2)='1' then r2<=ir2; end if;
    if control(enable_r3)='1' then r3<=ir3; end if;
  end process;
end rtl_alternative;

```

figuur 7.11 Datapad beschrijving met meer ontwerpvrijheid.

7.4.2 De besturing.

Een VHDL beschrijving van het data-pad is waarschijnlijk niet in één keer foutloos. Toch kan een simulatie tegen de specificatie pas worden uitgevoerd nadat ook de besturing in VHDL is beschreven. Als deze combinatie dan gesimuleerd wordt tegen de specificatie van de processor en er worden verschillen waargenomen kan de fout in het data-pad en/of besturing zitten. Het is dan ook zaak om voor de besturing eerst weer een specificatie te gebruiken omdat hierin minder snel fouten worden gemaakt.

7.4.2.1 De specificatie van de besturing.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.control_types.all;
use work.control_names.all;
entity controller is
  port (inst      : std_logic;
        start     : std_logic;
        clk       : std_logic;
        reset     : std_logic;
        control   : out control_bus;
        ready     : out std_logic);
end controller;

architecture behaviour of controller is
  type mode is (resetting,operation);
  signal ctrl_mode : mode;
  signal rdy_int : std_logic;
begin
  synchroniser:process
  begin
    wait until rising_edge(clk);
    if reset='1'
      then ctrl_mode <= resetting;
    elsif reset='0' and rdy_int='1'
      then ctrl_mode <= operation;
    end if;
  end process;

  cntrl:process
    variable count : natural range 0 to 3;
    constant add : std_ulogic := '0';
    constant mul : std_ulogic := '1';
  begin
    idle:loop
      -- detect start condition
      exit when start='1' and ctrl_mode=operation;
      control<=(others => '0'); rdy_int<='1';
      wait until rising_edge(clk);
    end loop;

    -- initialisation
    control<=(enable_r1|enable_r2|enable_r3|init =>'1',others =>'0'); rdy_int<='1';
    wait until rising_edge(clk);
    case inst is
      when add => control<=(enable_r1|enable_r3|addition=>'1', others =>'0');
        rdy_int<='0';
        wait until rising_edge(clk);
      when mul => count:=3;
        loop
          control<=(enable_r1|enable_r3|shift_add =>'1', others =>'0');
          rdy_int<='0';
          wait until rising_edge(clk);
          exit when count=0;
          count:=count-1;
        end loop;
      when others => assert false report "illegal instruction" severity warning;
    end case;

    loop -- is start signal again '0' (handshake condition)
      exit when start='0' or ctrl_mode=resetting;
      control<=(others=>'0'); rdy_int<='0';
      wait until rising_edge(clk);
    end loop;
  end process;

  ready <= rdy_int;
end behaviour;

```

figuur 7.12 *Specificatie van de besturing.*

```

configuration cnf_behaviour_controller of test_environment is
  for structure
    for gdr:processor use entity work.processor(behaviour_alternative); end for;
    for ontw:processor use entity work.processor(datapath_controller);
      for datapath_controller
        for dp:datapath use entity work.datapath(rtl); end for;
        for ct:controller use entity work.controller(behaviour); end for;
      end for;
    end for;
  end for;
end cnf_behaviour_controller;

```

figgur 7.13 De configuratie voor het testen van de ontwerpstep.

De specificatie van de besturing is eenvoudig (figuur 7.12) . Opgemerkt moet worden dat veel ontwerpers vaak te snel een toestandsmachine willen beschrijven. In de volgende paragraaf zal de besturing in de vorm van een toestandsmachine worden gegeven. Deze beschrijving is gevoeliger voor ontwerpfouten.

Ook nu is al een ontwerpbeslissing genomen, als het signaal *reset* '1' wordt tijdens de vermenigvuldig operatie wordt deze operatie eerst afgerond in plaats van deze abrupt te onderbreken.

7.4.2.2 Een toestandsmachine beschrijving voor de besturing.

```

architecture fsm of controller is
  type mode is (resetting,operation);
  signal ctrl_mode : mode;
  signal rdy_int : std_ulogic;
begin
  synchroniser:process
  begin
    wait until rising_edge(clk);
    if reset='1'
      then ctrl_mode <= resetting;
    elsif reset='0' and rdy_int='1'
      then ctrl_mode <= operation;
    end if;
  end process;

  cntrl:process
  type states is (idle,init,add,mul,wait_start_0);
  variable state : states;
  variable count : natural range 0 to 3;
  begin
    wait until rising_edge(clk);
    case state is
      when idle      => if start='1' and ctrl_mode=operation
                        then state:=init; end if;
      when init      => if inst='1'
                        then state:=mul; count:=3;
                        else state:=add;
                        end if;
      when add       => if start='0'
                        then state:=idle;
                        else state:=wait_start_0;
                        end if;
      when mul       => if count>0
                        then count:=count-1;
                        elsif start='0'
                        then state:=idle;
                        else state:=wait_start_0;
                        end if;
      when wait_start_0 => if start='0' or ctrl_mode=resetting
                        then state:=idle; end if;
    end case;

    case state is
      when idle      => control<=(others=>'0'); rdy_int<='1';
      when init      => control<=(enable_r1|enable_r2|enable_r3|init => '1',
                                others=>'0'); rdy_int<='1';
      when add       => control<=(enable_r1|enable_r3|addition=>'1', others=>'0');
                        rdy_int<='0';
      when mul       => control<=(enable_r1|enable_r3|shift_add=>'1', others=>'0');
                        rdy_int<='0';
      when wait_start_0 => control<=(others=>'0'); rdy_int<='0';
    end case;
  end process;

  rdy_int <= rdy_int;
end fsm;

configuration cnf_fsm_controller of test_environment is
  for structure
  for gdr:processor use entity work.processor(behaviour_alternative); end for;
  for ontw:processor use entity work.processor(datapath_controller);
  for datapath_controller
  for dp:datapath use entity work.datapath(rtl); end for;
  for ct:controller use entity work.controller(fsm); end for;
  end for;
  end for;
end cnf_fsm_controller;

```

figuur 7.14 FSM (finite state machine) beschrijving voor de besturing en de configuratie nodig voor het testen tegen de specificatie van de processor.

```

architecture fsm1 of controller is
    type mode is (resetting,operation);
    signal ctrl_mode : mode;
    signal rdy_int : std_ulogic;
    type states is (idle,init,add,mul1, mul2,mul3,mul4,wait_start_0);
    signal n_state, c_state : states;
begin
    synchroniser:process
    begin
        wait until rising_edge(clk);
        if reset='1'
            then ctrl_mode <= resetting;
            elsif reset='0' and rdy_int='1'
                then ctrl_mode <= operation;
            end if;
        end process;

    process
    begin
        wait until rising_edge(clk);
        c_state <= n_state;
    end process;

    state_transition:process(c_state,ctrl_mode,start)
    begin
        case c_state is
            when idle          => if start='1' and ctrl_mode=operation
                                   then n_state<=init;
                                   end if;
            when init          => if inst='1'
                                   then n_state<=mul1;
                                   else n_state<=add;
                                   end if;
            when add           => if start='0'
                                   then n_state<=idle;
                                   else n_state<=wait_start_0;
                                   end if;
            when mul1          => n_state <= mul2;
            when mul2          => n_state <= mul3;
            when mul3          => n_state <= mul4;
            when mul4          => if start='0'
                                   then n_state<=idle;
                                   else n_state<=wait_start_0;
                                   end if;
            when wait_start_0 => if start='0' or ctrl_mode=resetting
                                   then n_state<=idle;
                                   end if;
        end case;
    end process;

    output_function:process(c_state)
    begin
        case c_state is
            when idle          => control<=(others=>'0'); rdy_int<='1';
            when init          => control<=(enable_r1|enable_r2|enable_r3|init => '1',
                                   others=>'0'); rdy_int<='1';
            when add           => control<=(enable_r1|enable_r3|addition=>'1', others=>'0');
                                   rdy_int<='0';
            when mul1|mul2|mul3|mul4 => control<=(enable_r1|enable_r3|shift_add=>'1',
                                   others=>'0'); rdy_int<='0';
            when wait_start_0 => control<=(others=>'0'); rdy_int<='0';
        end case;
    end process;

    ready <= rdy_int;
end fsm1;

```

figuur 7.15 Alternatieve beschrijving van een toestandsmachine.

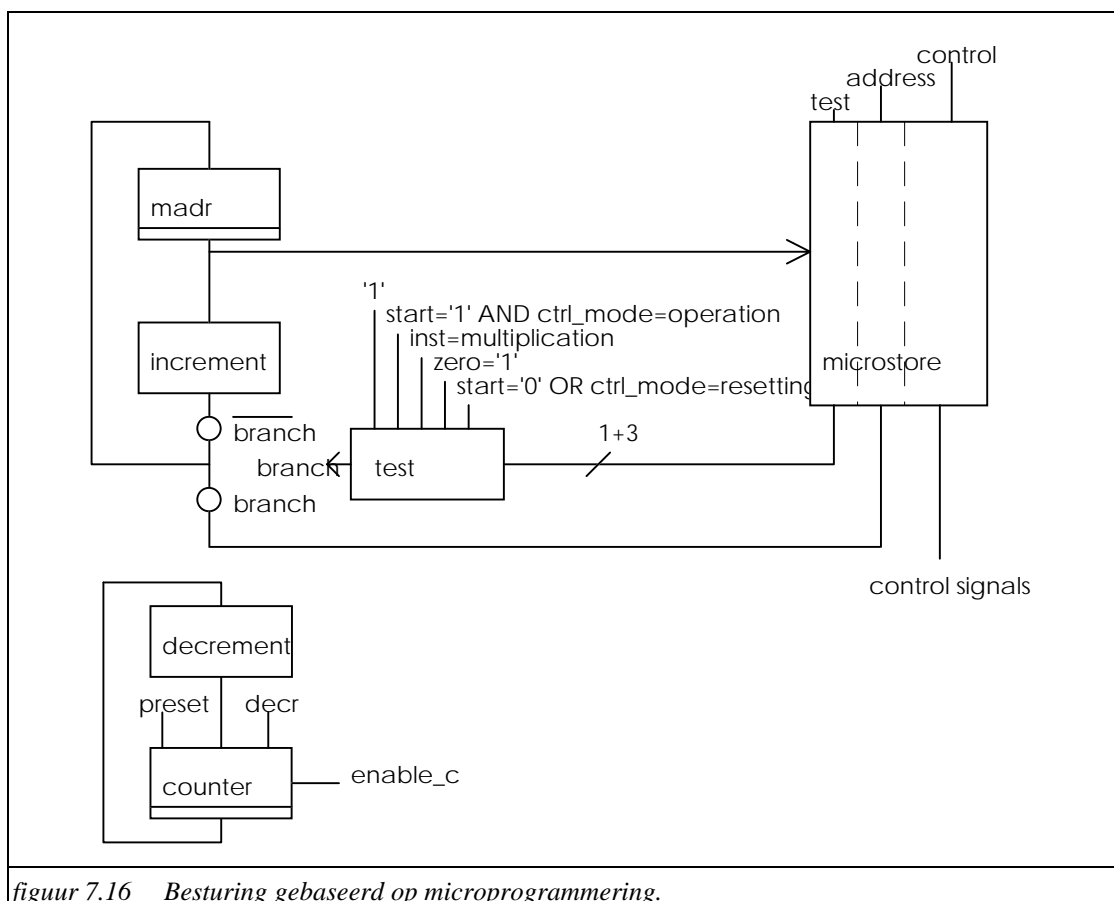
In de specificatie van de besturing (figuur 7.12) is de toestand niet expliciet aanwezig. In de *finite state machine* (FSM) beschrijving (figuur 7.14) zijn de volgende toestanden expliciet aanwezig:

- Een *idle* toestand voor het wachten op de volgende operatie.
- Een *initialisation* toestand voor het inlezen van de operanden.
- Een *addition* toestand voor het optellen van de twee operanden.

- Een *multiplication* toestand voor het herhaald optellen en schuiven ten behoeve van de instructie vermenigvuldigen. Een teller *count* houdt dan het aantal optel- en schuifoperaties bij.
- Een *wait for start is '0'* toestand. Deze toestand is toegevoegd om een nette afhandeling van het handshake protocol te verkrijgen. Er wordt in de beschrijving (figuur 7.14) altijd naar deze toestand gegaan, ook al is hetingangssignaal *start* al '0'. Hier kan dus nog enige snelheidswinst worden bereikt door een extra conditie op te nemen in de toestand *add(ition)* en de toestand *mul(tiplication)*. Dit gaat waarschijnlijk wel weer ten koste van een grotere besturing.

In de de toestandsmachine beschrijving gegeven in figuur 7.14 is nog de variable *count* aanwezig. Deze variable is gebruikt voor het vermenigvuldigen algoritme. In figuur 7.15 is een alternatief gegeven waarbij voor het vermenigvuldigen een viertal states zijn gebruikt. In deze beschrijving zijn ook een tweetal processen aanwezig die de volgende toestand bepalen en de uitgangssignalen. In principe hebben synthese-tools geen enkel probleem met beide beschrijvingswijzen.

7.4.2.3 Microprogrammering toegepast voor de besturing.



figuur 7.16 Besturing gebaseerd op microprogrammering.

Figuur 7.16 geeft het blokschema van de besturing gebaseerd op microprogrammering. Voor deze eenvoudige processor wel een zeer vreemde keuze. Om toch aan te geven hoe een dergelijke besturing in VHDL beschreven kan worden is het opgenomen.

Elk microwoord uit het microgeheugen bestaat uit de volgende drie velden:

- Test-veld

- Adres-veld
- Controle-signalen

Het test-veld selecteert een van de mogelijke ingangssignalen van de besturing. Het geselecteerde signaal bepaalt de volgende toestand. Dit kan zijn het volgende adres indien de testconditie niet waar is, of de besturing vervolgt de executie vanaf het adres aangegeven door het adres-veld indien de testconditie waar is.

Figuur 7.17 geeft de VHDL beschrijving. Ook hier is weer geprobeerd een leesbare vorm te hanteren. Merk op dat het microgeheugen wordt ‘gevuld’ aan het begin van de simulatierun door middel van procedure *init_memory*.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package microprogramming_components is
    function test(yes:std_logic; condition:std_logic_vector; test_inp:std_logic_vector)
        return std_logic;
    -- The 'condition' selects one of the test inputs (test_inp). If the yes bit
    -- is '1' the function returns that input, otherwise it is inverted.
    -- The selection is based on the integer value of the condition, and the
    -- numbering of the test inputs goes from left to right.
    -- example:
    --     test('1',"001", '1' & neg1 & neg2 & '0' & '0' & pos1 & pos2 & '0');
    --     "001" selects input 'neg1'.
end microprogramming_components;

package body microprogramming_components is
    function test(yes:std_logic; condition:std_logic_vector;
        test_inp:std_logic_vector)
        return std_logic is
        variable t_inp : std_logic_vector(0 to test_inp'length-1):=test_inp;
    begin
        return t_inp(to_integer(unsigned(condition))) xor not yes;
    end test;
end microprogramming_components;

-----
use work.microprogramming_components.all;
architecture microprogramming of controller is
    function boo2std_logic (inp : boolean) return std_logic is
        type bool2std_logic_table is array (boolean) of std_logic;
        constant bool2std_logic : bool2std_logic_table := (false => '0', true => '1');
    begin
        return bool2std_logic(inp);
    end boo2std_logic;

    type mode is (resetting,operation);

    type word is record
        yes          : std_logic;
        tst_field    : std_logic_vector(2 downto 0);
        address_field : natural range 0 to 7;
        preset       : std_logic;
        decr         : std_logic;
        enable_c     : std_logic;
        rdy_int      : std_logic;
        ext          : control_bus;
    end record;

    type memory is array (0 to 7) of word;

    procedure init_memory(signal micro : out memory) is
    begin
        micro<= (
            ('0',"001",0,'0','0','0','1',(others=>'0')),
            ('1',"010",3,'1','0','1','1',(enable_r1|enable_r2|enable_r3|init=>'1',others=>'0')),
            ('1',"000",7,'0','0','0','0','0',(enable_r1|enable_r3|addition=>'1',others=>'0')),
            ('0',"011",3,'0','1','1','0','0',(enable_r1|enable_r3|shift_add=>'1',others=>'0')),
            ('1',"000",7,'0','0','0','0','0',(others=>'0')),
            ('1',"000",7,'0','0','0','0','0',(others=>'0')),
            ('1',"000",7,'0','0','0','0','0',(others=>'0')),
            ('0',"100",7,'0','0','0','0','0',(others=>'0')));
        -- tst_fld 000 : always true
        -- tst_fld 001 : start='1' and ctrl_mode=operation?
        -- tst_fld 010 : is instruction multiplication?
        -- tst_fld 011 : is zero '1'?
        -- tst_fld 100 : start='0' or ctrl_mode=resetting?
        wait;
    end init_memory;

    signal ctrl_mode : mode;
    signal microstore : memory;
    signal microword  : std_logic_vector(7 downto 0);
    signal zero       : std_logic;
    signal branch     : std_logic;
    signal imadr,madr  : natural range 0 to 7;
    constant bra       : std_logic := '1';
    signal icount,count : natural range 0 to 3;
begin
    synchroniser:process
    begin
        wait until rising_edge(clk);
        if reset='1'
            then ctrl_mode <= resetting;
        elsif reset='0' and microstore(madr).rdy_int='1'
            then ctrl_mode <= operation;
        end if;
    end process;
end architecture;

```

```

    end if;
end process;

init_memory(microstore);
control    <= microstore(madr).ext;
ready      <= microstore(madr).rdy_int;
branch     <= test(microstore(madr).yes,microstore(madr).tst_field,
                bra &
                (start and boo2std_logic(ctrl_mode=operation)) &
                inst &
                zero &
                ((not start) or boo2std_logic(ctrl_mode=resetting)) &
                '0' & '0' & '0');

imadr      <= 0
            microstore(madr).address_field when branch='1' else
            (madr + 1) mod 8;
icount     <= 3
            when (microstore(madr).preset='1')
            or (microstore(madr).decr='1' and count=0) else
            (count-1) mod 4;
zero       <= '1'
            when count=0 else
            '0';

process
begin
    wait until rising_edge(clk);
    madr    <= imadr;
    if microstore(madr).enable_c='1' then count<=icount; end if;
end process;
end microprogramming;

```

figuur 7.17 VHDL beschrijving voor de besturing gebaseerd op microprogrammering.

7.5 De realisatie.

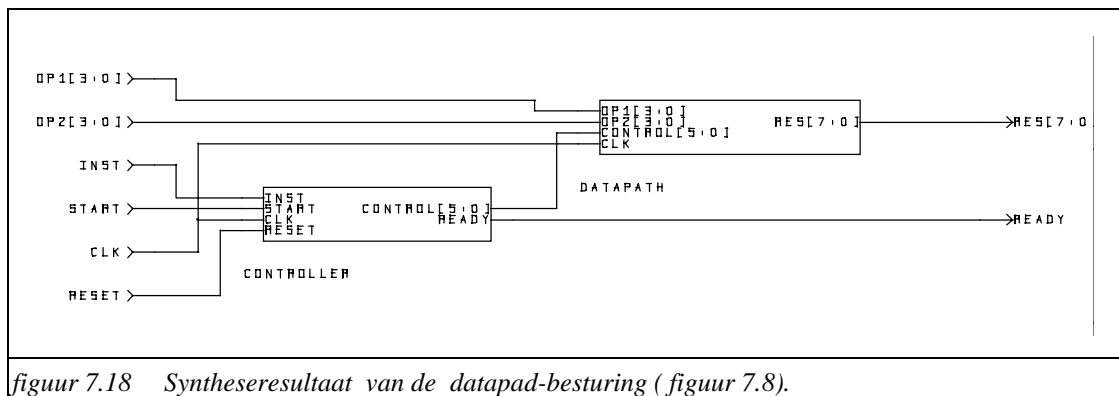
Wanneer kan gebruik gemaakt worden van een synthese-tool? Dit hangt in hoge mate af van de kracht van de synthese-tool. De beschrijving voor het data-pad (figuur 7.10) wordt meestal geaccepteerd, mogelijk met enkele kleine syntactische aanpassingen². Figuur 7.18 geeft de resultaten van synthese met behulp van VIEWSyn 7.20 en Actel Act3 als technologie.

De specificatie van de besturing (figuur 7.12) zal voor veel synthese-tools problemen geven. Veel synthese-tools ondersteunen slechts een procesbeschrijving met één *wait until* statement. Sommige synthese-tools ondersteunen in principe wel de beschrijving zoals gegeven in figuur 7.12, maar moeten de wait-statements op speciale plaatsen staan, bijvoorbeeld in een *loop* statement direct na het gereserveerde woord *loop*.

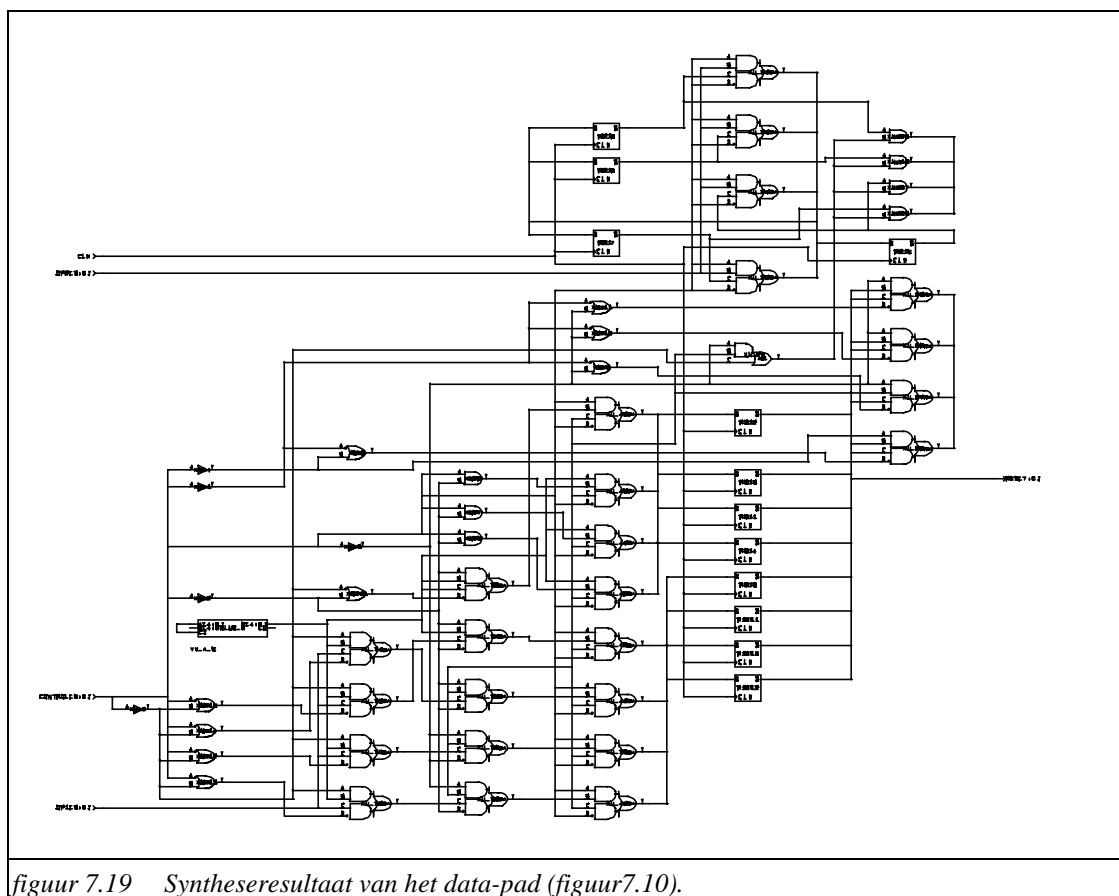
De VHDL beschrijving op basis van de microprogrammering (figuur 7.16) zal, op deze wijze beschreven, waarschijnlijk niet gesynthetiseerd worden. Het herkennen van het (micro)geheugen is het probleem. Dit kan worden opgelost door een model van een geheugen als component te instantiëren. Dit model is vaak technologie afhankelijk. In de figuren figuur 7.18, 7.19 en 7.20 zijn als illustratie de schema's gegeven van de synthetiseresultaten.

² De wijzingen aangebracht voor synthese:

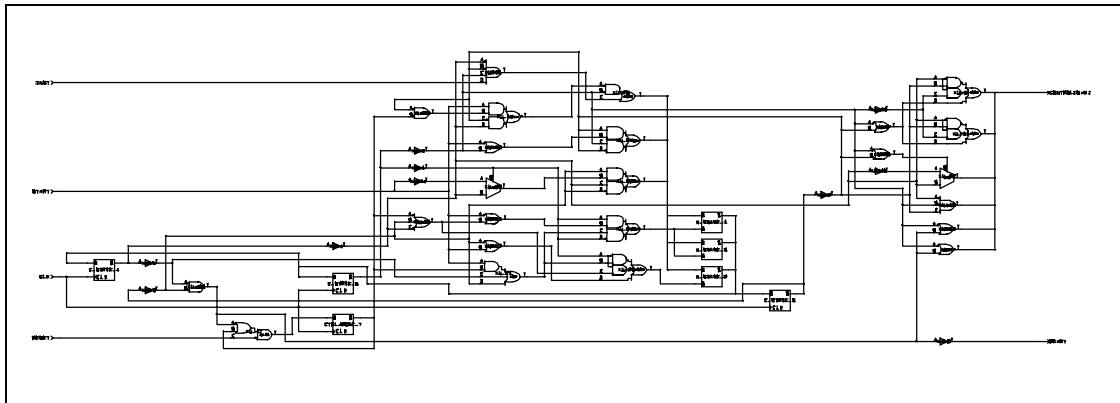
- In plaats van de package `numeric_std` is de package `std_logic_arith` (Synopsys) gebruikt. De meeste synthesesetools ondersteunen de packages van Synopsys.
- Voor het type `control_bus`, met de symbolische namen, is een `std_logic_vector` genomen, wat enigszins ten koste is gegaan van de leesbaarheid.



figuur 7.18 Syntheseresultaat van de datapad-besturing (figuur 7.8).



figuur 7.19 Syntheseresultaat van het data-pad (figuur 7.10).



figuur 7.20 Syntheseresultaat van de besturing (figuur 7.15).

7.6 De gebruikte packages.

In dit hoofdstuk zijn de gebruikte packages opgenomen behalve de IEEE packages *std_logic_1164* en *numeric_std*.

7.6.1 Package control_names.

```
library ieee;
use ieee.std_logic_1164.all;
package control_names is
    -- Only fill in the names of the control signals.
    type control_signals is (enable_r1,enable_r2,enable_r3,init,shift_add,addition);
    type control_bus is array (control_signals) of std_logic;
end control_names;
```

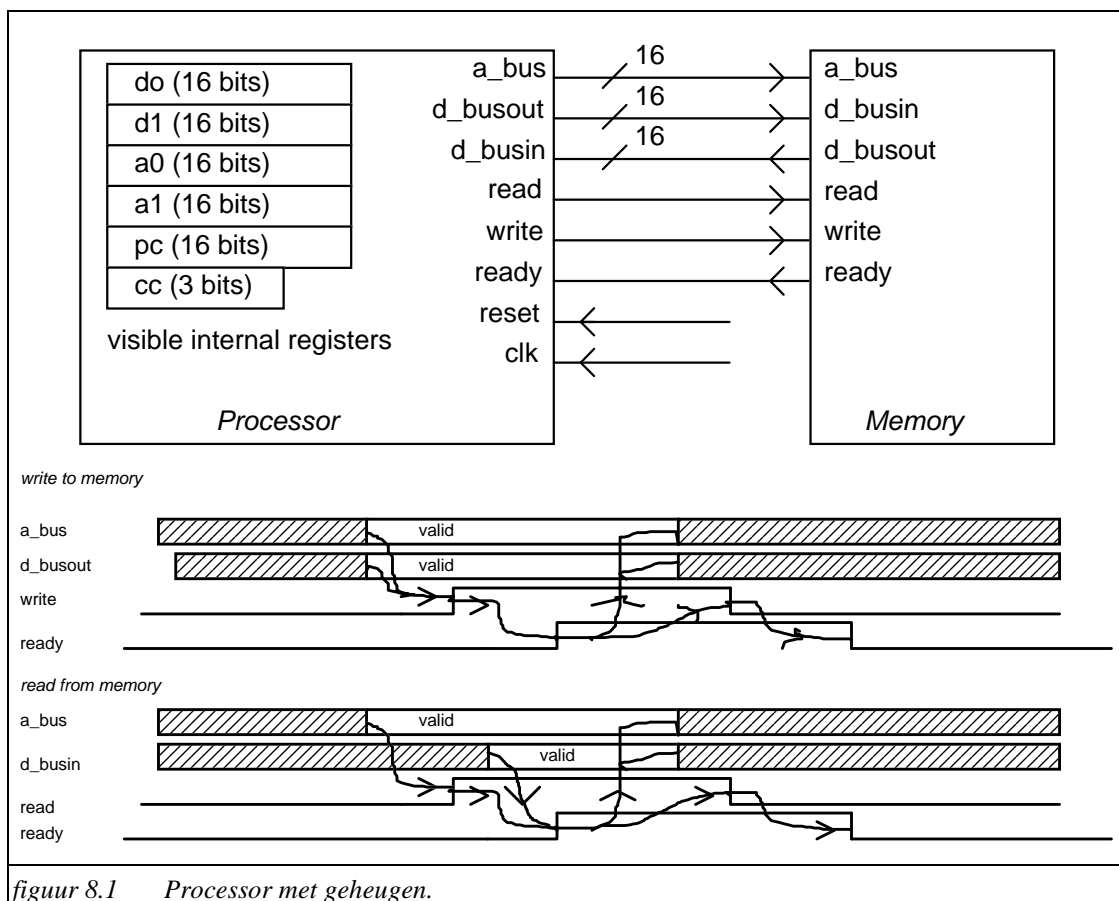
;figuur 7.21 Package control_names.

8. Specificatie van een processor.

Hoe kan een complexe processor van gespecificeerd in VHDL? In dit hoofdstuk wordt een functioneel model beschreven, dus exclusief de tijdsaspecten. De processor is gebaseerd op de processor van Peter Ashenden's "VHDL Cookbook" first edition 1990. Het aantal adresseringsmodi is uitgebreid, evenals de instructieset. Wat de instructieset betreft moet worden opgemerkt dat het voor de lezer wel als een zeer merkwaardige instructieset moet overkomen. Echter om didactische redenen wordt in het college "Ontwerpen van Digitale Systemen" deze processor als uitgangspunt gebruikt. Eveneens is om didactische redenen afgeweken van de oorspronkelijke bi-directionele data-bus.

Eerst zal een informele beschrijving van de processor worden gegeven, gevolgd door de VHDL beschrijving.

8.1 Informele beschrijving van de processor.



De processor heeft de volgende voor de programmeur zichtbare registers:

- Twee data-registers, d0 en d1 beiden 16 bits breed.
- Twee adres-registers, a0 en a1 beiden 16 bits breed.
- Een *program counter* (pc) van 16 bits breed.
- Een conditie code register (cc) van 3 bits breed, met:
 - N bit: Is waar indien het resultaat negatief is (2-complement representatie)
 - Z bit: Is waar indien het resultaat '0' is.
 - V bit: Dit bit heeft twee betekenissen:
 1. is waar indien er een verflow optreedt bij rekenkundige operaties, of

2. bevat het resultaat van een vergelijkingsoperatie.

De externe data-bus is 16 bits breed en de communicatie is asynchroon en gebaseerd op een handshake-protocol.

Het instructieformaat bestaat uit één of twee woorden. Het formaat van het eerste woord is:

```

< opcode >
<inst_type> <instr> <source> <destination>
      3       5       4       4       bits

```

met:

```

'inst_type':
    000  data movement
    001  arithmetic and logical operations
    010  shift operations
    100  program control
    111  miscellaneous
'source' en 'destination':
    0000  none
    0001  immediate(#)
    0010  D0
    0011  D1
    0100  A0
    0101  A1
    0110  (A0) Register a0 bevat het adres van de data
    0111  (A1) Register a1 bevat het adres van de data

```

'instr'

zie figuur 8.2

Voorbeelden:

```

sub D0 D1      001_00000_0010_0011
beq (A0)       100_00001_0000_0100
beq #4355      100_00001_0000_0001
               0100_0011_0101_0101
vset           111_00100_0000_0000
inca A0        111_10000_0000_0100
ror D1         010_00101_0000_0011
mov #4231 , (A0) 000_00000_0001_0110
               0100_0010_0011_0001
add D0 , D1     001_00100_0010_0011
add #3124 , D0  001_00100_0001_0010
               0011_0001_0010_0100

```

Arithmetic and logical operations

restriction: d = D0 of D1, s = D0, D1, (A0), (A1) or immediate

In case of an overflow of an arithmetic operation the result is don't care and the V-bit is set.

instruct	mnemonic		
00000	subt s,d	$d := d - s$	subtract
00001	abssub s,d	$d := (d-s) $	subtract, absolute result
00010	absmsub s,d	$d := - (d-s) $	subtract, negative result
00100	add s,d	$d := d + s$	add
00101	absadd s,d	$d := (d+s) $	add, absolute result
00110	absmadd s,d	$d := - (d+s) $	add, negative result
01000	maxi s,d	$d := \text{Maxi}(d,s)$	maximum in d
01001	maxa s,d	$d := \text{Maxi}(d , s)$	maximum absolute values in d
01010	mini s,d	$d := \text{Mini}(d,s)$	minimum in d
01011	mina s,d	$d := \text{Mini}(d , s)$	minimum absolute values in d
01100	abs s,d	$d := s $	absolute value of s in d
01101	absmin s,d	$d := - s $	negative abs. value of s in d
01110	mul s,d	$d := d(7:0)*s(7:0)$	two bytes are multiplied
01111	absmul s,d	$d := (d(7:0)*s(7:0)) $	absolute result of product
10000	kl s,d	$d < s$	V bit of condition code reg. is affected
10001	klg s,d	$d \leq s$	V bit of condition code reg. is affected
10010	kla s,d	$(d) < (s)$	V bit of condition code reg. is affected
10011	klga s,d	$(d) \leq (s)$	V bit of condition code reg. is affected
10100	comp s,d	$d = s$	V bit of condition code reg. is affected

shift operations

restriction: destination = D0, D1, source = none

instruct	mnemonic		
00000	asl d	$d \leftarrow d(14:0) \& 0$	arithmetic shift left
00001	asr d	$d \leftarrow d(15) \& d(15:1)$	arithmetic shift right
00010	lsl d	$d \leftarrow d(14:0) \& 0$	logical shift left
00011	lsr d	$d \leftarrow 0 \& d(15:1)$	logical shift right
00100	rol d	$d \leftarrow d(14:0) \& d(15)$	rotate left
00101	ror d	$d \leftarrow d(0) \& d(15:1)$	rotate right

data movement

restriction: d = D0, D1, A0, A1, (A0), (A1) or Immediate s = D0, D1, A0, A1, (A0), (A1) or Immediate, AND s and d not both immediate.

instruct	mnemonic	
00000	mov s,d	move data from s to d
condition code register is not affected		

program control

restriction: destination = immediate, (A0), (A1), source = none

If the condition is true the new program counter is:

pc \leftarrow pc + displacement,
condition code register is not affected

instruct	mnemonic	
00000	bra d	branch always

00001	beq d	branch if Z=1
00010	bne d	branch if Z=0
00011	bvs d	branch if V=1
00100	bvc d	branch if V=0
00101	bpl d	branch if N=0
00110	bmi d	branch if N=1
condition code register is not affected		

miscellaneous

setting and resetting of condition code register bits:

restriction: destination = none, source = none

the not used condition code bits are not affected

instruct mnemonic

00000	nset	set N bit
00001	nclr	clear N bit
00010	zset	set Z bit
00011	zclr	clear Z bit
00100	vset	set V bit
00101	vclr	clear V bit

increment/decrement the registers A0 en A1.

restriction: destination = A0, A1, source = none

instruct mnemonic

10000	inca d	d:=d+1	increment d
10001	deca d	d:=d-1	decrement d

figuur 8.2 Instructieset van de processor.

8.2 Het geheugen.

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.processor_types.all;
entity memory is
  generic (tpd : time := 1 ns);
  port(d_busout : out std_logic_vector(15 downto 0);
       d_busin  : in  std_logic_vector(15 downto 0);
       a_bus    : in  unsigned(15 downto 0);
       write    : in  std_logic;
       read     : in  std_logic;
       ready    : out std_logic);
end memory;
-----
use work.utilities.all;
architecture behaviour of memory is
begin
  process
    constant low_address:natural:=0;
    constant high_address:natural:=300; -- upper limit of the memory
                                         -- INCREASE this number if the program
                                         -- needs more memory. Don't FORget
                                         -- that the addresses used to write
                                         -- to and read from should be available.

    type memory_array is
      array (natural range low_address to high_address) of integer;
    variable mem:memory_array:=
      (18, --      mov #6,d0      0000 0000 0001 0010
       6,  --      0000 0000 0000 0110
       20, --      mov #62,a0     0000 0000 0001 0100
       62, --      0000 0000 0011 1110
       21, --      mov #63,a1     0000 0000 0001 0101
       63, --      0000 0000 0011 1111
       19, --      mov #1,d1      0000 0000 0001 0011
       1,  --      0000 0000 0000 0001
       54, --      mov d1,(a0)     0000 0000 0011 0110
       55, --      mov d1,(a1)     0000 0000 0011 0111
       13347, -- lbl: comp d0,d1  0011 0100 0010 0011
       -31999, --      bvs einde:  1000 0011 0000 0001
       9,  --      0000 0000 0000 1001
       9235, --      add #1,d1     0010 0100 0001 0011
       1,  --      0000 0000 0000 0001
       55, --      mov d1,(a1)     0000 0000 0011 0111
       11875, --      mul (a0),d1  0010 1110 0110 0011
       -3836, --      deca a0      1111 0001 0000 0100
       54,  --      mov d1,(a0)     0000 0000 0011 0110
       115, --      mov (a1),d1     0000 0000 0111 0011
       -32767, --      bra lbl:    1000 0000 0000 0001
       -12,  --      1111 1111 1111 0100
       -32767, --      einde: bra einde: 1000 0000 0000 0001
       -2,  --      1111 1111 1111 1110
       others => 0
      );
    variable address:natural;
    constant unknown : std_logic_vector(15 downto 0) := (others=>'-');
  begin
    ready <= '0' after tpd;
    --
    -- wait for a command
    --
    wait until (read='1') or (write='1');
    address:=to_integer(a_bus);
    assert (address>=low_address) and (address<=high_address)
      report "out of memory range" severity warning;
    if write='1'
    then
      mem(address):=to_integer(signed(d_busin));
      ready<='1' after tpd;
      wait until write='0'; -- wait until end of write cycle
    else -- read = '1';
      d_busout <= std_logic_vector(to_signed(mem(address),16));
      ready<='1' after tpd;
      wait until read='0';
      d_busout <= unknown;
    end if;
  end process;
end behaviour;

```

figuur 8.3 Gedragsbeschrijving van het geheugen.

Figuur 8.3 geeft een gedragsbeschrijving voor het geheugen. Het programma opgeslagen in het geheugen bepaalt de faculteiten van de getallen 1 tot en met 6. De resultaten worden in de adressen 62 (voor 1!) tot en met 57 (voor 6!) opgeslagen. Voor deze processor is ook een assembler geschreven die niet behandeld zal worden.

8.3 De formele beschrijving van de processor.

De processor kent een aantal rekenkundige operaties. Voor de leesbaarheid en onderhoudbaarheid zijn ook nu weer een tweetal packages ontwikkeld:

- Package utilities (figuur 8.4). Hierin zijn een aantal hulpfuncties voor de schuifoperaties beschreven.
- Package processor_types (figuur 8.6). Deze package bevat de codering van de instructieset.

8.3.1 Package utilities.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package utilities is
  type bool2std_logic_table is array (boolean) of std_logic;
  constant bool2bit:bool2std_logic_table:=(false=>'0', true=>'1');
  type direction is (left,right);
  type domain is (logical,arithmetic);
  function shift(x : signed; dir:direction; mode:domain)
    return signed;
  function rotate(x: signed; dir:direction)
    return signed;
  function member(x:std_logic_vector;list:std_logic_vector)
    return boolean;
    -- is x member of the list, where x is a std_logic_vector
    -- and list is a concatenation of these std_logic_vectors
    -- exa. x=001 and list=000_100_011, hence x is not in the list
end utilities;
```

figuur 8.4 Package utilities.

```

package body utilities is
  function shift(x : signed; dir:direction; mode:domain)
    return signed is
    variable tmp : signed(x'length downto 1):=x;
  begin
    case dir is
      when left => return tmp(tmp'length-1 downto 1) & '0';
      when right =>
        case mode is
          when logical => return '0' & tmp(tmp'length downto 2);
          when arithmetic => return tmp(tmp'length) & tmp(tmp'length downto 2);
        end case;
      end case;
    end shift;

    function rotate(x: signed; dir:direction)
      return signed is
      variable tmp : signed(x'length downto 1):=x;
    begin
      case dir is
        when left => return tmp(tmp'length-1 downto 1) & tmp(tmp'length);
        when right => return tmp(1) & tmp(tmp'length downto 2);
      end case;
    end rotate;

    function member(x:std_logic_vector;list:std_logic_vector) return boolean is
      variable lgt_x : natural := x'length;
      variable lgt_list : natural := list'length;
      variable llist : std_logic_vector(0 to list'length-1):=list;
      variable i : natural := 0;
    begin
      while i<lgt_list loop
        if x=llist(i to i+lgt_x-1) then return true; end if;
        i:=i+lgt_x;
      end loop;
      return false;
    end member;
  end utilities;

```

figuur 8.5 Package body utilities.

8.3.2 Package *processor_types*.

figuur 8.6 Package en package body van *processor_types*.

8.3.3 Beschrijving van het gedrag van de processor.

```

library ieee;
use ieee.std_logic_1164.all;
use work.processor_types.all;
use work.utilities.all;
entity processor is
  port (d_busout: out bit16;
        d_busin : in  bit16;
        a_bus   : out bit16;
        write   : out std_ulogic;
        read    : out std_ulogic;
        ready   : in  std_ulogic;
        reset   : in  std_ulogic;
        clk     : in  std_ulogic);
end processor;

architecture behaviour of PROCESSor is
begin
  process
    variable pc : natural;
    variable a0 : bit16;
    variable a1 : bit16;
    variable d0 : bit16;
    variable d1 : bit16;
    variable cc : bit3;
    alias cc_n : std_ulogic is cc(2);
    alias cc_z : std_ulogic is cc(1);
    alias cc_v : std_ulogic is cc(0);
    variable data : bit16;
    variable current_instr:bit16;
    alias op : bit8 is current_instr(15 downto 8);
    alias src : bit4 is current_instr( 7 downto 4);
    alias dst : bit4 is current_instr( 3 downto 0);
    variable error_src_dst : boolean; -- error in src or dst in instruction
    variable rs,rd         : bit16;   -- temporary VARIABLES
    variable rs_int, rd_int : integer;  -- integer representation of rs, rd.
    variable rs_low, rd_low : integer;  -- " " positions 7..0 of rs and rd.

    variable rc             : std_ulogic; -- "
    variable displacement   : bit16;
    variable jump           : boolean;    -- used in branch instructions
    variable tmp            : bit16;
    constant one            : bit16 := (0 => '1', others => '0');
    constant dontcare       : bit16 := (others => '-');

    procedure memory_read (addr : in natural;
                           result : out bit16) is
      -- Used 'global' signals are:
      -- clk, reset, ready, read, a_bus, d_busin
      -- read data from addr in memory
      variable inp : bit16;
      variable add : bit16;
    begin
      nat2bitv(addr,add);
      -- put address on output
      a_bus <= add;
      wait until clk='1';
      if reset='1' then
        return;
      end if;

      loop -- ready must be low (handshake)
        if reset='1' then
          return;
        end if;
        exit when ready='0';
        wait until clk='1';
      end loop;

      read <= '1';
      wait until clk='1';
      if reset='1' then
        return;
      end if;

      loop
        wait until clk='1';
        if reset='1' then
          return;
        end if;

        if ready='1' then
          result:=d_busin;

```

```

        exit;
    end if;
end loop;
wait until clk='1';
if reset='1' then
    return;
end if;

read <= '0';
a_bus <= dontcare;
end memory_read;

procedure memory_write(addr : in natural;
                       data : in bit16) is
-- Used 'global' signals are:
--   clk, reset, ready, write, a_bus, d_busout
-- write data to addr in memory
    variable add : bit16;
begin
    nat2bitv(addr,add);
    -- put address on output
    a_bus <= add;
    wait until clk='1';
    if reset='1' then
        return;
    end if;

    loop -- ready must be low (handshake)
        if reset='1' then
            return;
        end if;
        exit when ready='0';
        wait until clk='1';
    end loop;

    d_busout <= data;
    wait until clk='1';
    if reset='1' then
        return;
    end if;
    write <= '1';

    loop
        wait until clk='1';
        if reset='1' then
            return;
        end if;
        exit when ready='1';
    end loop;
    wait until clk='1';
    if reset='1' then
        return;
    end if;
    --
    write <= '0';
    d_busout <= dontcare;
    a_bus <= dontcare;
end memory_write;

procedure read_data(s_d      : in bit4;
                   d0, d1 : in bit16;
                   a0, a1 : in bit16;
                   pc      : inout natural;
                   data     : out bit16) is
-- read data from d0,d1,a0,a1,(a0),(a1),imm
    variable tmp : bit16;
begin
    case s_d is
        when rd0 => data := d0;
        when rd1 => data := d1;
        when ra0 => data := a0;
        when ra1 => data := a1;
        when a0_ind => memory_read(bitv2nat(a0),data);
        when a1_ind => memory_read(bitv2nat(a1),data);
        when imm   => memory_read(pc,data);
                    pc := pc + 1;
        when others => assert false report "illegal src/dst while reading data"
                        severity warning;
    end case;
end read_data;

procedure write_data(s_d      : in bit4;
                    d0, d1 : inout bit16;
                    a0, a1 : inout bit16;
                    pc      : inout natural;
                    data     : in bit16) is
-- write data to d0,d1,a0,a1,(a0),(a1),imm

```

```

variable tmp:bit16;
variable addr: bit16;
begin
  case s_d is
    when rd0 => d0:=data;
    when rd1 => d1:=data;
    when ra0 => a0 := data;
    when ral => a1 := data;
    when a0_ind => memory_write(bitv2nat(a0),data);
    when a1_ind => memory_write(bitv2nat(a1),data);
    when imm => memory_read(pc,addr);
                  pc := pc + 1;
                  memory_write(bitv2nat(addr),data);
    when others => assert false report "illegal src or dst while writing data"
                  severity warning;
  end case;
end write_data;

begin
  --
  -- check for reset active
  --
  if reset='1' then
    read <= '0';
    write <= '0';
    pc := 0;
    cc := "000"; -- clear condition code register
    loop -- synchrone reset
      wait until clk='1';
      exit when reset='0';
    end loop;
  end if;
  --
  -- fetch next instruction
  --
  memory_read(pc,current_instr);
  if reset /= '1' then
    pc:=pc+1;
    --
    -- decode & execute
    --
    case op is
      when mov =>
        error_src_dst:= not member(src,rd0&rd1&ra0&ral&a0_ind&a1_ind&imm) or
                        not member(dst,rd0&rd1&ra0&ral&a0_ind&a1_ind&imm) or
                        ((src=imm) and (dst=imm));
        assert not error_src_dst report "illegal inst. mov"
        severity warning;
        read_data(src,d0,d1,a0,a1,pc,rs);
        write_data(dst,d0,d1,a0,a1,pc,rs);
        cc := "---";

      when sub|abssub|absmsub|add|absadd|absmadd|maxi|maxa|mini|mina|
            abs1|absmin|mul|absmul =>
        error_src_dst:= not member(src,rd0&rd1&a0_ind&a1_ind&imm) or
                        not member(dst,rd0&rd1);
        assert not error_src_dst report "illegal inst. ARITHMETIC" severity warning;
        read_data(src,d0,d1,a0,a1,pc,rs);
        rs_int:=bitv2int(rs); rs_low:=bitv2int(rs(7 downto 0));
        read_data(dst,d0,d1,a0,a1,pc,rd);
        rd_int:=bitv2int(rd); rd_low:=bitv2int(rd(7 downto 0));
        case op is
          when sub => rd_int := rd_int - rs_int;
          when abssub => rd_int := abs( rd_int - rs_int );
          when absmsub => rd_int := -abs( rd_int - rs_int );
          when add => rd_int := rd_int + rs_int;
          when absadd => rd_int := abs( rd_int + rs_int );
          when absmadd => rd_int := -abs( rd_int + rs_int );

          when maxi => if rs_int > rd_int then rd_int:=rs_int; end if;
          when maxa => if abs(rs_int) > abs(rd_int)
                        then rd_int:=abs(rs_int); end if;
          when mini => if rs_int < rd_int
                        then rd_int:=rs_int; end if;
          when mina => if abs(rs_int) < abs(rd_int)
                        then rd_int:=abs(rs_int); end if;

          when abs1 => rd_int := abs(rs_int);
          when absmin => rd_int := -abs(rs_int);
          when mul => rd_int := rd_low * rs_low;
          when absmul => rd_int := abs( rd_low * rs_low );
          when others => null;
        end case;
        set_cc_rd(rd_int,cc,rd);
        write_data(dst,d0,d1,a0,a1,pc,rd);

      when kl|klg|kla|klga|comp =>

```

```

error_src_dst:= not member(src,rd0&rd1&a0_ind&a1_ind&imm) or
               not member(dst,rd0&rd1);
assert not error_src_dst report "illegal inst. COMPARE" severity warning;
read_data(src,d0,d1,a0,a1,pc,rs); rs_int:=bitv2int(rs);
read_data(dst,d0,d1,a0,a1,pc,rd); rd_int:=bitv2int(rd);
case op is
  when kl      => cc_v := bool2bit( rd_int < rs_int);
  when klg     => cc_v := bool2bit( rd_int <= rs_int );
  when kla     => cc_v := bool2bit(abs(rd_int) < abs(rs_int));
  when klga    => cc_v := bool2bit(abs(rd_int) <= abs(rs_int));
  when comp    => cc_v := bool2bit( rd = rs);
  when others  => null;
end case;
cc_n := '-'; cc_z := '-';

when asl|asr|lsl|lsr|rol_87|ror_87 =>
  error_src_dst:= not member(src,none) or
                 not member(dst,rd0&rd1);
  assert not error_src_dst report "illegal inst. SHIFT" severity warning;
  read_data(dst,d0,d1,a0,a1,pc,rd);
  case op is
    when asl => rd:=shift(rd,left,arithmetic);
    when asr => rd:=shift(rd,right,arithmetic);
    when lsl => rd:=shift(rd,left,logical);
    when lsr => rd:=shift(rd,right,logical);
    when rol_87 => rd:=rotate(rd,left);
    when ror_87 => rd:=rotate(rd,right);
    when others => null;
  end case;
  cc := "----";
  write_data(dst,d0,d1,a0,a1,pc,rd);

when bra|beq|bne|bvs|bvc|bpl|bmi =>
  error_src_dst:= not member(src,none) or
                 not member(dst,a0_ind&a1_ind&imm);
  assert not error_src_dst report "illegal inst. BRANCH" severity warning;
  case op is
    when bra => jump := TRUE;
    when beq => jump := cc_z='1';
    when bne => jump := cc_z='0';
    when bvs => jump := cc_v='1';
    when bvc => jump := cc_v='0';
    when bpl => jump := cc_n='0';
    when bmi => jump := cc_n='1';
    when others => null;
  end case;
  -- condition code register has not changed
  if jump
  then
    case dst is
      when imm    => memory_read(pc,displacement);
                    pc := pc + 1;
      when a0_ind => memory_read(bitv2nat(a0),displacement);
      when a1_ind => memory_read(bitv2nat(a1),displacement);
      when others => assert false
                    report "illegal destination in BRANCH instruction"
                    severity warning;
    end case;
    pc := pc + bitv2int(displacement);
  else if dst=imm then pc := pc + 1; end if; -- skip contents next address
  end if;

when nset|nclr|zset|zclr|vset|vclr =>
  error_src_dst:= not member(src,none) or
                 not member(dst,none);
  assert not error_src_dst report "illegal instruction SET or CLR of CC"
                    severity warning;
  case op is
    when nset    => cc_n:='1';
    when nclr    => cc_n:='0';
    when zset    => cc_z:='1';
    when zclr    => cc_z:='0';
    when vset    => cc_v:='1';
    when vclr    => cc_v:='0';
    when others  => null;
  end case;
  -- other condition code bits will be not changed

when inca|deca =>
  error_src_dst:= not member(src,none) or
                 not member(dst,ra0&ra1);
  assert not error_src_dst report "illegal inst. INCA, DECA" severity warning;
  case op is
    when inca =>
      case dst is
        when ra0 => nat2bitv(bitv2nat(a0)+1,a0);
        when ra1 => nat2bitv(bitv2nat(a1)+1,a1);

```



```

        when others => null;
    end case;
when deca =>
    case dst is
        when ra0 => nat2bitv(bitv2nat(a0)-1,a0);
        when ral => nat2bitv(bitv2nat(al)-1,al);
        when others => null;
    end case;
    when others => null;
end case;
cc := "---";

when others => assert false report "illegal instruction" severity warning;

end case;
end if;
end process;
end behaviour;

```

figuur 8.7 Formele beschrijving van de specificatie van de processor.

8.4 Interactie tussen de processor en het geheugen.

Figuur 8.1 geeft schematisch weer hoe de processor en het geheugen met elkaar zijn verbonden. De VHDL beschrijving is gegeven in figuur 8.8. Een deel van het simulatieresultaat is gegeven in figuur 8.9. De integerrepresentatie van de 16 bits brede data-bus en adres-bus is gegeven. Dit kan vergeleken worden met de inhoud van het geheugen (figuur 8.3).

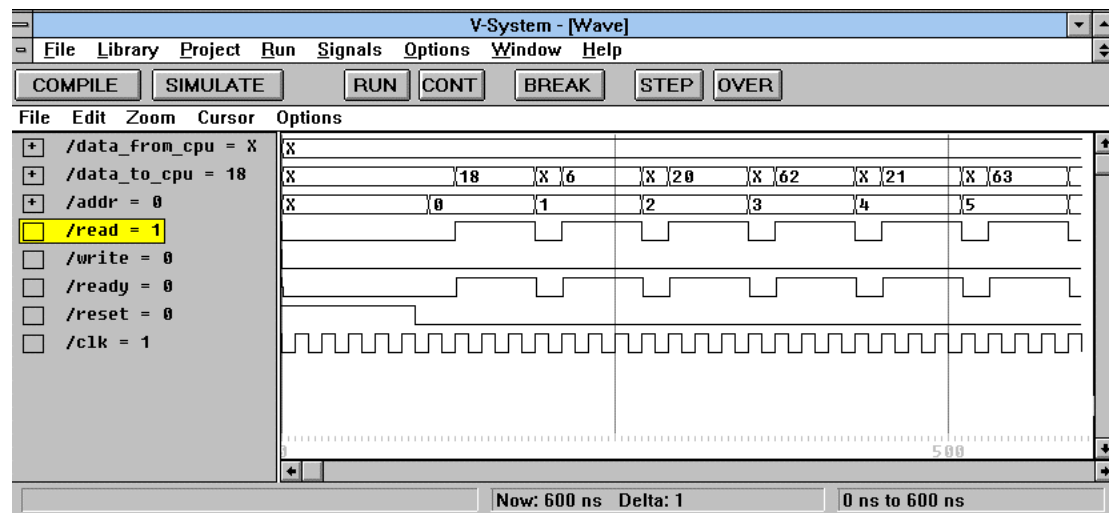
```

entity dut is
end dut;

library ieee;
use ieee.std_logic_1164.all;
use work.processor_types.all;
architecture memory_processor of dut is
    component memory
        generic (tpd : time := 1 ns);
        port(d_busout : out bit16;
             d_busin  : in  bit16;
             a_bus    : in  bit16;
             write     : in  std_ulogic;
             read      : in  std_ulogic;
             ready     : out std_ulogic);
    end component;
    component processor
        port (d_busout: out bit16;
             d_busin : in  bit16;
             a_bus   : out bit16;
             write   : out std_ulogic;
             read    : out std_ulogic;
             ready   : in  std_ulogic;
             reset   : in  std_ulogic;
             clk     : in  std_ulogic);
    end component;
    signal data_from_cpu,data_to_cpu,addr : bit16;
    signal read,write,ready               : std_ulogic;
    signal reset                          : std_ulogic := '1';
    signal clk                            : std_ulogic := '0';
begin
    cpu:processor
        port map(data_from_cpu,data_to_cpu,addr,write,read,ready,reset,clk);
    mem:memory
        generic map (1 ns)
        port map (data_to_cpu,data_from_cpu,addr,write,read,ready);
    reset <= '1', '0' after 100 ns;
    clk   <= not clk after 10 ns;
end memory_processor;
-----
configuration test_of_mem_proc of dut is
    for memory_processor
        for cpu:processor use entity work.processor (behaviour); end for;
        for mem:memory use entity work.memory (behaviour); end for;
    end for;
end test_of_mem_proc;

```

figuur 8.8 Structuurbeschrijving van de interactie tussen de processor en het geheugen.



figuur 8.9 Eerste 600 ns van het simulatieresultaat.

A. Documenten over VHDL.

A.1 Selectie van de uitgaven van de IEEE.

- "1076-1987 Standard VHDL Language Reference Manual", IEEE Standards, The IEEE computer society press 1991 catalog: catalog number: 983; Standard No. 1076-1987; Order Code SH11957;
- "IEEE Standard VHDL Language Reference Manual," IEEE Std 1076-1993, IEEE Standards, Order Code SH 16840, \$56, 30% IEEE Member Discount, bulk discounts, ISBN 1-55937-376-8, 1994
- The Sense of VASG Standard No. 1076-CONC-1990; Order Code SH13326; \$22.50; "This publication is the companion document to IEEE Std 1076-1987. It clarifies ambiguities and editorial errors in IEEE Std 1076-1987 and contains accepted requests for new language features".
- "IEEE Standards Interpretations: IEEE Standard VHDL Language Reference Manual," IEEE Std 1076/INT-1991, Order Code SH148941991
- "IEEE Standard Multivalued Logic System for VHDL Model Interoperability (std_logic_1164)," IEEE Std 1164-1993
- "IEEE Standard VHDL Language Math Packages", IEEE Std. 1076.2-1996
- "IEEE Standard VHDL Synthesis Packages", IEEE Std. 1076.3-1997
- "IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification", IEEE Std. 1076.4-1995

A.2 Boeken.

- Doug Perry, **VHDL**, 390 pages, ?? illustrations, 2nd edition, ISBN 0-07-049434-7, MacGraw-Hill, ATTN: Charles Decker, Professional Publishing Group 11 West 19th Street, New York, NY 10011, available
- David Coelho, **The VHDL Handbook**, ISBN 0-7923-90310-8, Kluwer Academic Publishers, 1989 examples and std package on (PC) disc for \$100 handling cost from: Coelho Publications, 43000 Christy Street, Fremont, CA 94538 voice: (415) 770-0875, fax: (415) 770-0728, email: !uunet!coelho!drc
- Peter J. Ashenden, **The VHDL Cookbook**, per ftp from ftp.cs.adelaide.edu.au (129.127.8.8) pub/VHDL-Cookbook (Mac,PC,PS) or ftp://bears.ece.ucsb.edu/pub/VHDL/comp.lang.vhdl
- J. Armstrong, **Chip Level Modelling in VHDL**, Prentice Hall, 1988, pp. 148, 119,- german marks
- Lipsett, Schaeffer, Ussery, **An Introduction to VHDL: Hardware Description and Design**, Kluwer Academic Publishers, 1989, 320 pp, \$59.95, ISBN 0-7923-9030-x
- Randolph Harr and Alec Stanculescu, **Applications of VHDL to Circuit Design**, Kluwer Academic Publishers, 1991, 256 pp, \$75.00, ISBN 0-7923-9153-5 101 Philip Drive, Norwell, MA 02061
- S. Leung, M.A. Shanblatt, **ASIC System Design with VHDL: A Paradigm**, Kluwer Academic Publishers, 1989, 240 pp, \$56.50, ISBN 0-7923-9032-6
- Steve Carlson, **Introduction to HDL-Based Design Using VHDL**, Synopsys, Inc., \$49.95, 700 East Middlefield Road, Mountain View, CA 94043 (415)962-5000
- Larry M. Augustin, David C. Luckham, Benoit A. Gennart, Yo Huh and Alec G. Stanculescu, **Hardware Design and Simulation in VAL/VHDL**, Kluwer Academic Publishers, 1991, 352 pp, \$69.95, ISBN 0-7923-9087-3

- Joel M. Schoen, **Performance and Fault Modeling with VHDL**, Prentice Hall ISBN 0-13-658816-6 \$44.00 406 pages clothbound
- J. Bhasker, **A VHDL Primer**, Prentice Hall, ISBN 0-13-952987-X, revised edition (includes features from VHDL-93) ISBN 0-13-181447-8 phone 1-515-284-6751
- Jean Michel Berge, Alain Fonkua, Serge Maginot, Jacques Roulliard, **VHDL Designer's Reference**, Kluwer Academic Publishers, ISBN 0-7923-1756-4
- Stanley Mazor, Patricia Langstraat, **A Guide to VHDL**, Kluwer Academic Publishers, ISBN 0-7923-9255-8
- Jean Mermet, **VHDL for Simulation, Synthesis and Formal Proofs of Hardware**, Kluwer academic publishers, ISBN 0-7923-9253-1
- Zainalabedin Navabi, **VHDL: Analysis and Modelling of Digital Systems**, 1/E, ISBN 0-07-046472-3, Mc Graw Hill, US\$38.50
- Louis Baker, **VHDL Programming with Advanced Topics**, John Wiley & Sons, New York, 1993
- J.R. Armstrong and F. Gail Gray, **Structured Logic Design With VHDL**, Prentice Hall, Englewood Cliffs, N.J., USA, ISBN 0-13-855206-1, May of 1993
- Chin-Hwa Lee, **Digital System Design using VHDL**, CorralTek P.O. 2616, Salinas, CA 93902 (408) 484-1726, Price \$29 (answer book \$10)
- A. Dewey, **Analysis and Design of Digital Systems with VHDL**, Addison-Wesley, 1992
- Circuit Synthesis with VHDL, R Airiau, JM Berge, V Olive, Kluwer Academic Publishers, 1994, ISBN 0-7923-9429-1
- Berge, Fonkoua, Maginot and Rouillard, **VHDL '92; The New Features of the VHDL Hardware Description Language**, Kluwer Academic Publishers ISBN:0-7923-9356-2, Price: \$87.50 Dfl180.00; table of contents, preface and chapter 1 can be retrieved by anonymous ftp from ftp.std.com. The file is Kluwer/books/vhdl92
- Peter J. Ashenden, **The Designer's Guide to VHDL**, Morgan Kaufmann Publishers, 1995, 710 pages, ISBN 1-55860-270-4, list price US\$44.95, <http://Literary.COM//mkp/pages/2704/index.html>
- J. Bhasker, **A VHDL Primer, Revised Edition**, Prentice Hall, ISBN 0-13-181447-8 (this edition is based on VHDL-93)
- Ott, **A Designer's Guide to VHDL Synthesis**, Kluwer Academic Publishers, ISBN 0-7923-9472-0
- J. Bhasker, **A Guide to VHDL Syntax**, Prentice Hall, ISBN 0-13-324351-6, \$45.00, Number of pages: 268
- Carlos Delgado Kloos, Peter T. Breuer, **Formal Semantics for VHDL**, Kluwer Academic Publishers, more info on: <http://www.dit.upm.es/~cdk/inv/euroform/sem-vhdl.html>
- J. Pick, **VHDL Techniques, Experiments, and Caveats**, McGraw-Hill, ISBN 0-07-049906-3, \$55.
- Ben Cohen, **VHDL Coding Styles and Methodologies, an In-depth Tutorial**, Kluwer Academic Publishers, 1995, 365 pp, \$94, Disk included, ISBN 0-7923-9598-0, Web page: <http://members.aol.com/vhdlcohen/vhdl/>
- Andrew Rushton, **VHDL for Logic Synthesis**, McGraw-Hill, 1995, ISBN: 0-07-709092-6 Hardback, 40 UK Pounds (no information yet on other currencies)
- D Hunter, T Johnson, **Introduction to VHDL**, Chapman & Hall, 246x189mm, 496 pages, 8 line illus, November 1995 Paperback:0-412-73130-4:UK o24.99
- Yu-Chin Hsu, **VHDL Modeling for Digital Design Synthesis**, 376p, Kluwer Academic Publishers, ISBN 0-7923-9597-2

- Ross, **Digital Design & Synthesis with VHDL**, 03/1994 Automata Publishing Company, Cloth Text, ISBN 0-9627488-3-8 300p
- Steve Wolfe and Fouad Kiamilev, **VHDL Buyer's Guide**, Trade Paper ISBN 0-934869-14-6 30p, 11/1992 Cad Cam Publishing, Incorporated
- J. Bhasker, **A VHDL Synthesis Primer**, Star Galaxy Publishing, February 1996, 250pp, Hardcover, \$49.95, ISBN: 0-9650391-0-2, Publisher/Order info: Star Galaxy Publishing, 1058 Treeline Drive, Suite 247, Allentown, PA 18103, 610-391-7296 (Phone/fax: 24 hrs), <http://members.aol.com/SGalaxyPub>
- J. Bhasker, **VHDL: Features and Applications**, A self-study course, IEEE, Contains: Self study course + final exam + A VHDL Primer(Revised Edition) text + IEEE Std 1076-1993 + IEEE Std 1164-1993 (all in one package), \$229.00 IEEE members, \$299 non-members, IEEE Product number: HL5712, To order: Call IEEE at (800) 678-IEEE
- Douglas J. Smith, **HDL Chip Design - A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog**, Large format (8.5 x 11) inches, case bound, 464 pages, 240 illustrations, 180 practical modeling examples showing VHDL and Verilog side-by-side followed by a common synthesized. \$65. ISBN 0-9651934-3-8, June 1996, Doone Publications, Tel./Fax: USA 205 837 0580. email: asmith@hiwaay.net Web: <http://fly.HiWAAY.net/~asmith>
- M. S. Ben Romdhane, V. K. Madiseti, J. W. Hines, **Quick-Turnaround ASIC Design in VHDL :: Core-Based Behavioral Synthesis** (Foreword by Prof. J. Allen, MIT), June 1996, \$95, ISBN 0-7923-9744-4, Kluwer Academic Publishers, Norwell, MA, (<http://www.ee.gatech.edu/users/215/book3.html>)
- Vijay K. Madiseti, **VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis** (using VHDL and other means), 524 pp., 1995, IEEE Press, ISBN 7506-9406-8, IEEE Order Number PC5599, \$59, <http://www.ee.gatech.edu/users/vijaykm/bl.html>
- David Pellerin and Douglas Taylor, **VHDL Made Easy**, 420 pages ISBN 0-13-650763-8 Prentice Hall, 1996, \$54.95. Includes CD-ROM containing a VHDL simulator, VHDL examples and other software. Available direct from Accolade Design Automation (800-470-2686), 26331 NE Valley Street, Suite 5-120, Duvall, WA 98019, FAX 206-788-3768, <http://www.acc-eda.com/>
- Ben Cohen, **VHDL Answers to Frequently Asked Questions**, Kluwer Academic Publishers, 1997, 291 pp, Disk included, ISBN 0-7923-9791-6, \$100, Web page: <http://members.aol.com/vhdlcohen/vhdl/>
- Stefan Sjöholm and Lennart Lindh, **VHDL for Designers**, ISBN 0-13-473414-9, Prentice Hall, Covers everything from basics of VHDL through test benches, synthesis, test methodology even to behavioral synthesis
- R. Airiau, J.M. Berge, V. Olive and J. Rouillard, **VHDL du langage a la modelisation**, Presses Polytechniques et Universitaires Romandes, Lausanne 1990
- Gunther Lehmann, Bernhard Wunder, Manfred Selz, **Schaltungsdesign mit VHDL**, 317 Seiten, mit Diskette, Franzis-Verlag, ISBN 3-7723-6163-3, Poing, 1994, DM 89,-
- K. ten Hagen, **Abstrakte Modellierung digitaler Schaltungen** (VHDL vom funktionalen Modell bis zur Gatterebene), Springer, ISBN 3-540-59143-5, August 1995
- Thomas Kropf, **VLSI-Entwurf** (Vorgehen, Methoden, Automatisierung), TAT (Thomson Aktuelle Tutorien) Nr.17, ISBN 3-8266-0163-7, International Thomson Publishing, (trotz des Titels ist das Buch zu ca. 80% ein VHDL-Buch)

- J. Bhasker, **Die VHDL-Syntax** (Deutsche Uebersetzung von "A Guide to VHDL Syntax"), Prentice Hall Verlag GmbH, ISBN: 3-8272-9528-9
- Jayaram Bhasker, **Transation of: A VHDL Primer**, CQ Publishing, ISBN4-7898-3286-4 C3055 P3200E
- Egbert Molenkamp, **VHDL, VHDL'87/'93 en voorbeelden**, 1996, 246 pp, ISBN 90-802634-2-7, Order via: Transfer EDS, Institutenweg 16, 7521 PK, Enschede, the Netherlands

A.3 Conferenties.

Tijdens veel conferenties op het terrein van hardware wordt aandacht besteed aan ontwikkelingen op het gebied van VHDL, o.a. de DAC, de EDAC, de EURO-DAC (inclusief EURO-VHDL). Vanaf 1998 worden deze conferenties samengevoegd tot de DATE-conferentie die jaarlijks in het voorjaar gehouden zal worden. Naast deze conferenties, die jaarlijks georganiseerd worden, zijn er ook conferenties die geheel in het teken van VHDL staan:

- VIUF (VHDL International User's forum). Wordt tweemaal per jaar gehouden en vindt plaats in Amerika. De conferentie in het voorjaar wordt gezamenlijk georganiseerd met de IVC (International Verilog Conference). Voor informatie kan contact opgenomen worden met de conference management services, email: cms@cis.stanford.edu.
- VHDL Forum for CAD in Europa (VFE). Wordt tweemaal per jaar gehouden en vindt plaats in Europa. Andreas Hohl (Chair) SIEMENS, Dept. ZFE IS EA Ref Otto-Hahn-Ring 6 81 739 Munich, Duitsland, Tel: +49-89-636-41895, Fax: +49-89-636-44950 Email: ah@ztivax.siemens.com. De voorjaarsbijeenkomst is vanaf 1998 ondergevracht in de DATE conferentie.

A.4 Nieuwsbrieven.

- VHDL Newsletter, editor of the US newsletter is Allen M. Dewey. email: adewey@vnet.IBM.COM
- European Newsletter, J. Mermet, J. Rouillard Phone: 3+91 05 44 44, FAX 33+91 05 43 43 Institut Mediterranee de Technologie Technopole de Chateau-Gombert 13415 Marseille cedex -FRANCE.

A.5 Documenten van ESA.

ESA (European Space agency) heeft een aantal interessante documenten ontwikkeld die via ftp te verkrijgen zijn. "ftp.estec.esa.nl" (anonymous ftp) in directory /pub/vhdl/doc, o.a.:

BoardLevel.ps, **VHDL Models for Board-level Simulation**, ref. WSM/SH/010
This document provides recommendations for development and usage of VHDL models intended for Board-level simulation. The purpose of these recommendations is to modelling criteria that will produce models that are highly accurate in both functionality and timing, and that will provide sufficient simulation performance to facilitate long simulation runs.

CompList.ps, **Support Component List**, ref. WDN/PS/822 Issue 2 This document contains summary information for European components for space applications. Their main characteristics are listed, including radiation tolerance. Coordinates to contact persons for further information are also included.

ModelGuide.ps, **VHDL Modelling Guidelines**, ref. ASIC/001 Issue 1 The ESA VHDL Modelling Guidelines have been established to ensure a good coding standard for VHDL, w.r.t. to readability, portability and extensive verification. There are separate sections dealing with specific requirements for models for component simulation, board-level simulation, system-level simulation and testbenches. The document is being used for education at several universities, as a base for company specific guidelines etc.

UseOfVHDL.ps, **The Usage of VHDL in the European Space Agency**, An overview of the current and envisaged (April 1995) usage of VHDL within ESA. Describes the background, scope and purpose of the ESA VHDL Modelling Guidelines, and introduces the upcoming document "VHDL Models for Board-level Simulation". Outlines a planned scheme for making VHDL models for Board-level simulation available including schemes for protecting the design information. The paper was presented at the Workshop on Libraries, Component Modeling and Quality Assurance in Nantes (F), April 1995.

VHDLReport.ps, **The VHDL Standard** (by E2S n.v.) An overview of the May 1994 status of the VHDL standard and associated activities within the IEEE, EIA, and ESPRIT projects. An extensive summary of VHDL repositories is included, together with a list of European VHDL tools.

B. VHDL activiteiten.

De IEEE verzorgt de standaardisatie van VHDL en packages. Enkele jaren geleden is bijvoorbeeld de STD_LOGIC_1164 package door de IEEE gestandaardiseerd en wordt inmiddels ondersteund door de meeste tools. Hiermee is er een eind gekomen aan de wildgroei van het 'logisch type'. De voorbereiding voor een standaardisatie wordt gecoördineerd door de DASC (Design Automation Standards Committee), welke op haar beurt het werk uitbesteed aan "working groups". Zo heeft de "Synthesis Working Group" de packages NUMERIC_BIT en NUMERIC_STD ontwikkeld.

De IEEE standaarden worden door de IEEE uitgegeven:

U.S.A.

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331 USA
Phone: 1-800-678-IEEE,
FAX: 908-981-9667

Europa

IEEE Computer Society, 13,
Avenue de l'Aquilon,
B-1200 Brussels BELGIUM
Telephone: 32.2.770.21.98
FAX: 32.2.770.85.05

Daarnaast kan via internet deelgenomen worden aan discussies op het gebied van VHDL. Op de newsgroup "comp.lang.vhdl" worden een groot aantal problemen met betrekking tot VHDL besproken. Maandelijks verschijnt er de "Frequently asked questions and answers". Onder anderen staan hierin de adressen van de leveranciers van VHDL tools en public domain software.

Internationaal is ook "VHDL International" (VI) actief.

VHDL International +1 415-329-0578 407
Chester Street 1 800-554-2550
Menlo Park, CA 94025 +1 415-324-3150 (Fax)
USA
cms@cis.stanford.edu (email)
<http://vhdl.org/> <http://www.e2w3.com/vi/>
ftp: vhdl.org met 'anonymous' login

De huidige Standards Groups kan gevonden worden op het VHDL International System op:

<http://www.vhdl.org/docs/groups.txt>, of
ftp vhdl.org in directory docs/groups.txt

Name	code-name
EIA DAD DIE ((bare) Die Info Exchange) Format	die
EIA DAD IBIS Open Forum (I/O Buffer Info Spec)	ibis
EIA DAD VHDL Commercial Component Model Spec	eia567
Free Model Foundation	fmf
IEEE DASC HW/SW Co-design SG	codesign
IEEE DASC VHDL Analog Extensions WG	analog
IEEE DASC VHDL EDIF Interoperability WG	vhdldef
IEEE DASC VHDL Library "library ieee;" WG	libieee
IEEE DASC VHDL Library - Utility WG	libutil
IEEE DASC VHDL Object Oriented SG	oovhdl
IEEE DASC VHDL Math Package WG	math

IEEE DASC VHDL Parallel Simulation WG	parallel
IEEE DASC VHDL Shared Variable WG	svwg
IEEE DASC VHDL Synthesis WG	vhdlsynth
IEEE DASC VHDL Test WG	vhdl_test
IEEE DASC Simulation Control Language SG	scl
IEEE DASC System Design & Description Lang. Study Group	sddl
IEEE DASC Timing (VHDL Initiative Towards ASIC Libraries)	vital
IEEE DASC VASG ISAC (VHDL)	*isac
IEEE DASC / SCC20 Waveform and Vector Exchange Std	waves
VHDL International (VI)	*vi
VHDL International Users' Forum (old VHDL Users' Group)	*viuf
VHDL Validation Suite Effort	validation
VI Marketing Advisory Committee	mac
VI Technical Advisory Committee	tac
VIUF Local Chapter - Boston, MA	boston_lc
VIUF Local Chapter - Mid-West	midwest_lc
VIUF Local Chapter - Phoenix, AZ	phoenix_lc
VIUF Local Chapter - Denver, CO	rockymnt_lc
VIUF Local Chapter - Sacramento, CA	sac_lc
VIUF Local Chapter - San Diego, CA	sandiego_lc
VIUF Local Chapter - Silicon Valley (Local Users Group)	svlug
VIUF Local Chapter - Southern California	socal_lc

Er zijn een drietal manieren om via email in kontakt te komen met een van de bovenstaande groepen:

1. Met “<code-name>-info@vhdl.org” wordt meer informatie verkregen.
2. Met “<code-name>-request@vhdl.org” kun je meedoen aan discussies die gevoerd worden binnen een groep.
3. Met “<code-name>@vhdl.org kun je een bericht versturen naar een groep.

C. Gereserveerde woorden.

The identifiers listed below are called *reserved words* and are reserved for significance in the language. For readability of this manual, the reserved words appear in lower case boldface.

abs	file	of	then
access	for	on	to
after	function	open	transport
alias		or	type
all	generate	others	
and	generic	out	unaffected
architecture	group		units
array	guarded	package	until
assert		port	use
attribute	if	postponed	
	impure	procedure	variable
begin	in	process	
block	inertial	pure	wait
body	inout		when
buffer	is	range	while
bus		record	with
	label	register	
case	library	reject	xnor
component	linkage	rem	xor
configuration	literal	report	
constant	loop	return	
		rol	
disconnect	map	ror	
downto	mod		
		select	
else	nand	severity	
elsif	new	signal	
end	next	shared	
entity	nor	sla	
exit	not	sll	
	null	sra	
		srl	
		subtype	

A reserved word must not be used as an explicitly declared identifier.

NOTES:

1. Reserved words differing only in the use of corresponding uppercase and lowercase letters are considered as the same. The reserved word **range** is also used as the name of a predefined attribute.
2. An extended identifier whose sequence of characters inside the leading and trailing backslashes is identical to a reserved word is not a reserved word. For example, `\next\` is a legal (extended) identifier and is not the reserved word **next**.

D. Syntax VHDL Std. 1076-1993.

This appendix provides a summary of the syntax for VHDL. Productions are ordered alphabetically by left-hand nonterminal name.

```

abstract_literal ::= decimal_literal | based_literal

access_type_definition ::= access subtype_indication

actual_designator ::=
    expression
    | signal_name
    | variable_name
    | open

actual_parameter_part ::= parameter_association_list

actual_part ::=
    actual_designator
    | function_name ( actual_designator )
    | type_mark ( actual_designator )

adding_operator ::= + | - | &

aggregate ::=
    ( element_association { , element_association } )

alias_declaration ::=
    alias alias_designator [ : subtype_indication ] is name [ signature ] ;

alias_designator ::= identifier | character_literal | operator_symbol

allocator ::=
    new subtype_indication
    | new qualified_expression

architecture_body ::=
    architecture identifier of entity_name is
        architecture_declarative_part
    begin
        architecture_statement_part
    end [ architecture ] [ architecture_simple_name ] ;

architecture_declarative_part ::=
    { block_declarative_item }

architecture_statement_part ::=
    { concurrent_statement }

array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition

assertion ::=
    assert condition
    [ report expression ]
    [ severity expression ]

```

```

assertion_statement ::= [ label : ] assertion ;

association_element ::=
    [ formal_part => ] actual_part

association_list ::=
    association_element { , association_element }

attribute_declaration ::=
    attribute identifier : type_mark ;

attribute_designator ::= attribute_simple_name

attribute_name ::=
    prefix [ signature ] ' attribute_designator [ ( expression ) ]

attribute_specification ::=
    attribute attribute_designator of entity_specification is expression ;

base ::= integer

base_specifier ::= B | O | X

based_integer ::=
    extended_digit { [ underline ] extended_digit }

based_literal ::=
    base # based_integer [ . based_integer ] # [ exponent ]

basic_character ::=
    basic_graphic_character | format_effector

basic_graphic_character ::=
    upper_case_letter | digit | special_character | space_character

basic_identifier ::=
    letter { [ underline ] letter_or_digit }

binding_indication ::=
    [ use entity_aspect ]
    [ generic_map_aspect ]
    [ port_map_aspect ]

bit_string_literal ::= base_specifier " [ bit_value ] "

bit_value ::= extended_digit { [ underline ] extended_digit }

block_configuration ::=
    for block_specification
        { use_clause }
        { configuration_item }
    end for ;

```

```

block_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | signal_declaration
  | shared_variable_declaration
  | file_declaration
  | alias_declaration
  | component_declaration
  | attribute_declaration
  | attribute_specification
  | configuration_specification
  | disconnection_specification
  | use_clause
  | group_template_declaration
  | group_declaration

block_declarative_part ::=
    { block_declarative_item }

block_header ::=
    [ generic_clause
    [ generic_map_aspect ; ] ]
    [ port_clause
    [ port_map_aspect ; ] ]

block_specification ::=
    architecture_name
  | block_statement_label
  | generate_statement_label [ ( index_specification ) ]

block_statement ::=
    block_label :
        block [ ( guard_expression ) ] [ is ]
            block_header
            block_declarative_part
        begin
            block_statement_part
        end block [ block_label ] ;

block_statement_part ::=
    { concurrent_statement }

case_statement ::=
    [ case_label : ]
        case expression is
            case_statement_alternative
            { case_statement_alternative }
        end case [ case_label ] ;

case_statement_alternative ::=
    when choices =>
        sequence_of_statements

character_literal ::= ' graphic_character '

```

```

choice ::=
    simple_expression
  | discrete_range
  | element_simple_name
  | others

choices ::= choice { | choice }

component_configuration ::=
    for component_specification
      [ binding_indication ; ]
      [ block_configuration ]
    end for ;

component_declaration ::=
    component identifier [ is ]
      [ local_generic_clause ]
      [ local_port_clause ]
    end component [ component_simple_name ] ;

component_instantiation_statement ::=
    instantiation_label :
      instantiated_unit
      [ generic_map_aspect ]
      [ port_map_aspect ] ;

component_specification ::=
    instantiation_list : component_name

composite_type_definition ::=
    array_type_definition
  | record_type_definition

concurrent_assertion_statement ::=
    [ label : ] [ postponed ] assertion ;

concurrent_procedure_call_statement ::=
    [ label : ] [ postponed ] procedure_call ;

concurrent_signal_assignment_statement ::=
    [ label : ] [ postponed ] conditional_signal_assignment
  | [ label : ] [ postponed ] selected_signal_assignment

concurrent_statement ::=
    block_statement
  | process_statement
  | concurrent_procedure_call_statement
  | concurrent_assertion_statement
  | concurrent_signal_assignment_statement
  | component_instantiation_statement
  | generate_statement

condition ::= boolean_expression

condition_clause ::= until condition

conditional_signal_assignment ::=
    target <= options conditional_waveforms ;

```

```

conditional_waveforms ::=
    { waveform when condition else }
    waveform [ when condition ]

configuration_declaration ::=
    configuration identifier of entity_name is
        configuration_declarative_part
        block_configuration
    end [ configuration ] [ configuration_simple_name ] ;

configuration_declarative_item ::=
    use_clause
    | attribute_specification
    | group_declaration

configuration_declarative_part ::=
    { configuration_declarative_item }

configuration_item ::=
    block_configuration
    | component_configuration

configuration_specification ::=
    for component_specification binding_indication ;

constant_declaration ::=
    constant identifier_list : subtype_indication [ := expression ] ;

constrained_array_definition ::=
    array index_constraint of element_subtype_indication

constraint ::=
    range_constraint
    | index_constraint

context_clause ::= { context_item }

context_item ::=
    library_clause
    | use_clause

decimal_literal ::= integer [ . integer ] [ exponent ]

declaration ::=
    type_declaration
    | subtype_declaration
    | object_declaration
    | interface_declaration
    | alias_declaration
    | attribute_declaration
    | component_declaration
    | group_template_declaration
    | group_declaration
    | entity_declaration
    | configuration_declaration
    | subprogram_declaration
    | package_declaration

```



```

delay_mechanism ::=
    transport
    | [ reject time_expression ] inertial

design_file ::= design_unit { design_unit }

design_unit ::= context_clause library_unit

designator ::= identifier | operator_symbol

direction ::= to | downto

disconnection_specification ::=
    disconnect guarded_signal_specification after time_expression ;

discrete_range ::= discrete_subtype_indication | range

element_association ::=
    [ choices => ] expression

element_declaration ::=
    identifier_list : element_subtype_definition ;

element_subtype_definition ::= subtype_indication

entity_aspect ::=
    entity entity_name [ ( architecture_identifier ) ]
    | configuration configuration_name
    | open

entity_class ::=
    

|                  |                     |                      |
|------------------|---------------------|----------------------|
| <b>entity</b>    | <b>architecture</b> | <b>configuration</b> |
| <b>procedure</b> | <b>function</b>     | <b>package</b>       |
| <b>type</b>      | <b>subtype</b>      | <b>constant</b>      |
| <b>signal</b>    | <b>variable</b>     | <b>component</b>     |
| <b>label</b>     | <b>literal</b>      | <b>units</b>         |
| <b>group</b>     | <b>file</b>         |                      |



entity_class_entry ::= entity_class [ <> ]

entity_class_entry_list ::=
    entity_class_entry { , entity_class_entry }

entity_declaration ::=
    entity identifier is
        entity_header
        entity_declarative_part
    [ begin
        entity_statement_part ]
    end [ entity ] [ entity_simple_name ] ;

```

```

entity_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration

entity_declarative_part ::=
    { entity_declarative_item }

entity_designator ::= entity_tag [ signature ]

entity_header ::=
    [ formal_generic_clause ]
    [ formal_port_clause ]

entity_name_list ::=
    entity_designator { , entity_designator }
    | others
    | all

entity_specification ::=
    entity_name_list : entity_class

entity_statement ::=
    concurrent_assertion_statement
    | passive_concurrent_procedure_call_statement
    | passive_process_statement

entity_statement_part ::=
    { entity_statement }

entity_tag ::= simple_name | character_literal | operator_symbol

enumeration_literal ::= identifier | character_literal

enumeration_type_definition ::=
    ( enumeration_literal { , enumeration_literal } )

exit_statement ::=
    [ label : ] exit [ loop_label ] [ when condition ] ;

exponent ::= E [ + ] integer | E – integer

```

```

expression ::=
    relation { and relation }
  | relation { or relation }
  | relation { xor relation }
  | relation [ nand relation ]
  | relation [ nor relation ]
  | relation { xnor relation }

extended_digit ::= digit | letter

extended_identifier ::=
    \ graphic_character { graphic_character } \

factor ::=
    primary [ ** primary ]
  | abs primary
  | not primary

file_declaration ::=
    file identifier_list : subtype_indication
      [ file_open_information ] ;

file_logical_name ::= string_expression

file_open_information ::=
    [ open file_open_kind_expression ] is file_logical_name

file_type_definition ::=
    file of type_mark

floating_type_definition := range_constraint

formal_designator ::=
    generic_name
  | port_name
  | parameter_name

formal_parameter_list ::= parameter_interface_list

formal_part ::=
    formal_designator
  | function_name ( formal_designator )
  | type_mark ( formal_designator )

full_type_declaration ::=
    type identifier is type_definition ;

function_call ::=
    function_name [ ( actual_parameter_part ) ]

generate_statement ::=
    generate_label :
      generation_scheme generate
        [ { block_declarative_item }
      begin ]
        { concurrent_statement }
      end generate [ generate_label ] ;

```

```

generation_scheme ::=
    for generate_parameter_specification
    | if condition

generic_clause ::=
    generic ( generic_list ) ;

generic_list ::= generic_interface_list

generic_map_aspect ::=
    generic map ( generic_association_list )

graphic_character ::=
    basic_graphic_character | lower_case_letter | other_special_character

group_constituent ::= name | character_literal

group_constituent_list ::= group_constituent { , group_constituent }

group_template_declaration ::=
    group identifier is ( entity_class_entry_list ) ;

group_declaration ::=
    group identifier : group_template_name ( group_constituent_list ) ;

guarded_signal_specification ::=
    guarded_signal_list : type_mark

identifier ::=      [§ 13.3]
    basic_identifier | extended_identifier

identifier_list ::= identifier { , identifier }

if_statement ::=
    [ if_label : ]
        if condition then
            sequence_of_statements
        { elsif condition then
            sequence_of_statements }
        [ else
            sequence_of_statements ]
        end if [ if_label ] ;

incomplete_type_declaration ::= type identifier ;

index_constraint ::= ( discrete_range { , discrete_range } )

index_specification ::=
    discrete_range
    | static_expression

index_subtype_definition ::= type_mark range <>

indexed_name ::= prefix ( expression { , expression } )

```

```

instantiated_unit ::=
    [ component ] component_name
    | entity entity_name [ ( architecture_identifier ) ]
    | configuration configuration_name

instantiation_list ::=
    instantiation_label { , instantiation_label }
    | others
    | all

integer ::= digit { [ underline ] digit }

integer_type_definition ::= range_constraint

interface_constant_declaration ::=
    [ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]

interface_declaration ::=
    interface_constant_declaration
    | interface_signal_declaration
    | interface_variable_declaration
    | interface_file_declaration

interface_element ::= interface_declaration

interface_file_declaration ::=
    file identifier_list : subtype_indication

interface_list ::=
    interface_element { ; interface_element }

interface_signal_declaration ::=
    [signal] identifier_list : [ mode ] subtype_indication [ bus ] [ := static_expression ]

interface_variable_declaration ::=
    [variable] identifier_list : [ mode ] subtype_indication [ := static_expression ]

iteration_scheme ::=
    while condition
    | for loop_parameter_specification

label ::= identifier

letter ::= upper_case_letter | lower_case_letter

letter_or_digit ::= letter | digit

library_clause ::= library logical_name_list ;

library_unit ::=
    primary_unit
    | secondary_unit

```

```

literal ::=
    numeric_literal
  | enumeration_literal
  | string_literal
  | bit_string_literal
  | null

logical_name ::= identifier

logical_name_list ::= logical_name { , logical_name }

logical_operator ::= and | or | nand | nor | xor | xnor

loop_statement ::=
    [ loop_label : ]
    [ iteration_scheme ] loop
    sequence_of_statements
    end loop [ loop_label ] ;

miscellaneous_operator ::= ** | abs | not

mode ::= in | out | inout | buffer | linkage

multiplying_operator ::= * | / | mod | rem

name ::=
    simple_name
  | operator_symbol
  | selected_name
  | indexed_name
  | slice_name
  | attribute_name

next_statement ::=
    [ label : ] next [ loop_label ] [ when condition ] ;

null_statement ::= [ label : ] null ;

numeric_literal ::=
    abstract_literal
  | physical_literal

object_declaration ::=
    constant_declaration
  | signal_declaration
  | variable_declaration
  | file_declaration

operator_symbol ::= string_literal

options ::= [ guarded ] [ delay_mechanism ]

package_body ::= [§ 2.6]
    package body package_simple_name is
    package_body_declarative_part
    end [ package body ] [ package_simple_name ] ;

```

```

package_body_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | shared_variable_declaration
  | file_declaration
  | alias_declaration
  | use_clause
  | group_template_declaration
  | group_declaration

package_body_declarative_part ::=
    { package_body_declarative_item }

package_declaration ::=
    package identifier is
        package_declarative_part
    end [ package ] [ package_simple_name ] ;

package_declarative_item ::=
    subprogram_declaration
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | signal_declaration
  | shared_variable_declaration
  | file_declaration
  | alias_declaration
  | component_declaration
  | attribute_declaration
  | attribute_specification
  | disconnection_specification
  | use_clause
  | group_template_declaration
  | group_declaration

package_declarative_part ::=
    { package_declarative_item }

parameter_specification ::=
    identifier in discrete_range

physical_literal ::= [ abstract_literal ] unit_name

physical_type_definition ::=
    range_constraint
    units
        base_unit_declaration
        { secondary_unit_declaration }
    end units [ physical_type_simple_name ]

port_clause ::=
    port ( port_list ) ;

port_list ::= port_interface_list

```

```

port_map_aspect ::=
    port map ( port_association_list )

prefix ::=
    name
    | function_call

primary ::=
    name
    | literal
    | aggregate
    | function_call
    | qualified_expression
    | type_conversion
    | allocator
    | ( expression )

primary_unit ::=
    entity_declaration
    | configuration_declaration
    | package_declaration

procedure_call ::= procedure_name [ ( actual_parameter_part ) ]

procedure_call_statement ::=
    [ label : ] procedure_call ;

process_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration

process_declarative_part ::=
    { process_declarative_item }

process_statement ::=
    [ process_label : ]
    [ postponed ] process [ ( sensitivity_list ) ] [ is ]
    process_declarative_part
    begin
    process_statement_part
    end [ postponed ] process [ process_label ] ;

process_statement_part ::=
    { sequential_statement }

qualified_expression ::=
    type_mark ' ( expression )
    | type_mark ' aggregate

```



```

range ::=
    range_attribute_name
    | simple_expression direction simple_expression

range_constraint ::= range range

record_type_definition ::=
    record
        element_declaration
        { element_declaration }
    end record [ record_type_simple_name ]

relation ::=

    shift_expression [ relational_operator shift_expression ]

relational_operator ::= = | /= | < | <= | > | >=

report_statement ::=
    [ label : ]
    report expression
    [ severity expression ] ;

return_statement ::=
    [ label : ] return [ expression ] ;

scalar_type_definition ::=
    enumeration_type_definition | integer_type_definition
    | floating_type_definition      | physical_type_definition

secondary_unit ::=
    architecture_body
    | package_body

secondary_unit_declaration ::= identifier = physical_literal ;

selected_name ::= prefix . suffix

selected_signal_assignment ::=
    with expression select
        target <= options selected_waveforms ;

selected_waveforms ::=
    { waveform when choices , }
    waveform when choices

sensitivity_clause ::= on sensitivity_list

sensitivity_list ::= signal_name { , signal_name }

sequence_of_statements ::=
    { sequential_statement }

```

```

sequential_statement ::=
    wait_statement
  | assertion_statement
  | report_statement
  | signal_assignment_statement
  | variable_assignment_statement
  | procedure_call_statement
  | if_statement
  | case_statement
  | loop_statement
  | next_statement
  | exit_statement
  | return_statement
  | null_statement

shift_expression ::=
    simple_expression [ shift_operator simple_expression ]

shift_operator ::= sll | srl | sla | sra | rol | ror

sign ::= + | -

signal_assignment_statement ::=
    [ label : ] target <= [ delay_mechanism ] waveform ;

signal_declaration ::=
    signal identifier_list : subtype_indication [ signal_kind ] [ := expression ] ;

signal_kind ::= register | bus

signal_list ::=
    signal_name { , signal_name }
  | others
  | all

signature ::= [ [ type_mark { , type_mark } ] [ return type_mark ] ]

simple_expression ::=
    [ sign ] term { adding_operator term }

simple_name ::= identifier

slice_name ::= prefix ( discrete_range )

string_literal ::= " { graphic_character } "

subprogram_body ::=
    subprogram_specification is
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [ subprogram_kind ] [ designator ] ;

subprogram_declaration ::=
    subprogram_specification ;

```

```

subprogram_declarative_item ::=
    subprogram_declaration
  | subprogram_body
  | type_declaration
  | subtype_declaration
  | constant_declaration
  | variable_declaration
  | file_declaration
  | alias_declaration
  | attribute_declaration
  | attribute_specification
  | use_clause
  | group_template_declaration
  | group_declaration

subprogram_declarative_part ::=
    { subprogram_declarative_item }

subprogram_kind ::= procedure | function

subprogram_specification ::=
    procedure designator [ ( formal_parameter_list ) ]
  | [ pure | impure ] function designator [ ( formal_parameter_list ) ]
    return type_mark

subprogram_statement_part ::=
    { sequential_statement }

subtype_declaration ::=
    subtype identifier is subtype_indication ;

subtype_indication ::=
    [ resolution_function_name ] type_mark [ constraint ]

suffix ::=
    simple_name
  | character_literal
  | operator_symbol
  | all

target ::=
    name
  | aggregate

term ::=
    factor { multiplying_operator factor }

timeout_clause ::= for time_expression

type_conversion ::= type_mark ( expression )

type_declaration ::=
    full_type_declaration
  | incomplete_type_declaration

```

```

type_definition ::=
    scalar_type_definition
  | composite_type_definition
  | access_type_definition
  | file_type_definition

type_mark ::=
    type_name
  | subtype_name

unconstrained_array_definition ::=
    array ( index_subtype_definition { , index_subtype_definition } )
    of element_subtype_indication

use_clause ::=
    use selected_name { , selected_name } ;

variable_assignment_statement ::=
    [ label : ] target := expression ;

variable_declaration ::=
    [ shared ] variable identifier_list : subtype_indication [ := expression ] ;

wait_statement ::=
    [ label : ] wait [ sensitivity_clause ] [ condition_clause ] [ timeout_clause ] ;

waveform ::=
    waveform_element { , waveform_element }
  | unaffected

waveform_element ::=
    value_expression [ after time_expression ]
  | null [ after time_expression ]

```

E. Package STANDARD.

Package STANDARD predefines a number of types, subtypes and functions. An implicit context clause naming this package is assumed to exist at the beginning of each design unit. Package STANDARD may not be modified by the user.

The operators that are predefined for the types declared for package STANDARD are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *universal_integer*), formal parameters, and undefined information (such as *implementation_defined*).

package STANDARD is

-- Predefined enumeration types:

type BOOLEAN is (FALSE, TRUE);

-- The predefined operators for this type are as follows:

```
-- function "and"      (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function "or"       (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function "nand"     (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function "nor"      (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function "xor"      (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function "xnor"     (anonymous, anonymous: BOOLEAN) return BOOLEAN;
```

```
-- function "not"      (anonymous: BOOLEAN) return BOOLEAN;
```

```
-- function "="        (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function "/="       (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function "<"         (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function "<="        (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function ">"         (anonymous, anonymous: BOOLEAN) return BOOLEAN;
-- function ">="        (anonymous, anonymous: BOOLEAN) return BOOLEAN;
```

type BIT is ('0', '1');

-- The predefined operators for this type are as follows:

```
-- function "and"      (anonymous, anonymous: BIT) return BIT;
-- function "or"       (anonymous, anonymous: BIT) return BIT;
-- function "nand"     (anonymous, anonymous: BIT) return BIT;
-- function "nor"      (anonymous, anonymous: BIT) return BIT;
-- function "xor"      (anonymous, anonymous: BIT) return BIT;
-- function "xnor"     (anonymous, anonymous: BIT) return BIT;
```

```
-- function "not"      (anonymous: BIT) return BIT;
```

```
-- function "="        (anonymous, anonymous: BIT) return BOOLEAN;
-- function "/="       (anonymous, anonymous: BIT) return BOOLEAN;
-- function "<"         (anonymous, anonymous: BIT) return BOOLEAN;
-- function "<="        (anonymous, anonymous: BIT) return BOOLEAN;
-- function ">"         (anonymous, anonymous: BIT) return BOOLEAN;
-- function ">="        (anonymous, anonymous: BIT) return BOOLEAN;
```

type CHARACTER is *ISO 8859-1-1987. Eight-bit coded character set.*¹

-- The predefined operators for this type are as follows:

```
-- function "="      (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function "/="     (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function "<"      (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function "<="     (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function ">"      (anonymous, anonymous: CHARACTER) return BOOLEAN;
-- function ">="     (anonymous, anonymous: CHARACTER) return BOOLEAN;
```

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

-- The predefined operators for this type are as follows:

```
-- function "="      (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function "/="     (anonymous, anonymous: SEVERITY_LEVEL) return
BOOLEAN;
-- function "<"      (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function "<="     (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function ">"      (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
-- function ">="     (anonymous, anonymous: SEVERITY_LEVEL) return BOOLEAN;
```

-- **type universal_integer is range** *implementation_defined*;

-- The predefined operators for this type are as follows:

```
-- function "="      (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function "/="     (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function "<"      (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function "<="     (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function ">"      (anonymous, anonymous: universal_integer) return BOOLEAN;
-- function ">="     (anonymous, anonymous: universal_integer) return BOOLEAN;
```

```
-- function "+"      (anonymous: universal_integer) return universal_integer;
-- function "-"      (anonymous: universal_integer) return universal_integer;
-- function "abs"    (anonymous: universal_integer) return universal_integer;
```

```
-- function "+"      (anonymous, anonymous: universal_integer
--                      return universal_integer;
-- function "-"      (anonymous, anonymous: universal_integer
--                      return universal_integer;
-- function "*"      (anonymous, anonymous: universal_integer
--                      return universal_integer;
-- function "/"      (anonymous, anonymous: universal_integer
--                      return universal_integer;
-- function "mod"    (anonymous, anonymous: universal_integer
--                      return universal_integer;
-- function "rem"    (anonymous, anonymous: universal_integer
--                      return universal_integer;
```

-- **type universal_real is range** *implementation_defined*;

¹ In the VHDL Standard Std. 1076-1993 the enumeration type is given.

```

-- The predefined operators for this type are as follows:

-- function "="      (anonymous, anonymous: universal_real) return BOOLEAN;
-- function "/="     (anonymous, anonymous: universal_real) return BOOLEAN;
-- function "<"      (anonymous, anonymous: universal_real) return BOOLEAN;
-- function "<="     (anonymous, anonymous: universal_real) return BOOLEAN;
-- function ">"      (anonymous, anonymous: universal_real) return BOOLEAN;
-- function ">="     (anonymous, anonymous: universal_real) return BOOLEAN;

-- function "+"      (anonymous: universal_real) return universal_real;
-- function "-"      (anonymous: universal_real) return universal_real;
-- function "abs"    (anonymous: universal_real) return universal_real;

-- function "+"      (anonymous, anonymous: universal_real) return universal_real;
-- function "-"      (anonymous, anonymous: universal_real) return universal_real;
-- function "*"      (anonymous, anonymous: universal_real) return universal_real;
-- function "/"      (anonymous, anonymous: universal_real) return universal_real;

-- function "*"      (anonymous: universal_real; anonymous: universal_integer)
--                    return universal_real;
-- function "*"      (anonymous: universal_integer; anonymous: universal_real)
--                    return universal_real;
-- function "/"      (anonymous: universal_real; anonymous: universal_integer)
--                    return universal_real;

-- Predefined numeric types:

type INTEGER is range implementation_defined;

-- The predefined operators for this type are as follows:

-- function "***"    (anonymous: universal_integer; anonymous: INTEGER)
--                    return universal_integer;
-- function "***"    (anonymous: universal_real; anonymous: INTEGER)
--                    return universal_real;
-- function "="      (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function "/="     (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function "<"      (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function "<="     (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function ">"      (anonymous, anonymous: INTEGER) return BOOLEAN;
-- function ">="     (anonymous, anonymous: INTEGER) return BOOLEAN;

-- function "+"      (anonymous: INTEGER) return INTEGER;
-- function "-"      (anonymous: INTEGER) return INTEGER;
-- function "abs"    (anonymous: INTEGER) return INTEGER;

-- function "+"      (anonymous, anonymous: INTEGER) return INTEGER;
-- function "-"      (anonymous, anonymous: INTEGER) return INTEGER;
-- function "*"      (anonymous, anonymous: INTEGER) return INTEGER;
-- function "/"      (anonymous, anonymous: INTEGER) return INTEGER;
-- function "mod"    (anonymous, anonymous: INTEGER) return INTEGER;
-- function "rem"    (anonymous, anonymous: INTEGER) return INTEGER;

```

```
-- function "***"      (anonymous: INTEGER; anonymous: INTEGER)
--                               return INTEGER;
```

type REAL is range implementation_defined;

-- The predefined operators for this type are as follows:

```
-- function "="      (anonymous, anonymous: REAL) return BOOLEAN;
-- function "/="     (anonymous, anonymous: REAL) return BOOLEAN;
-- function "<"      (anonymous, anonymous: REAL) return BOOLEAN;
-- function "<="    (anonymous, anonymous: REAL) return BOOLEAN;
-- function ">"      (anonymous, anonymous: REAL) return BOOLEAN;
-- function ">="    (anonymous, anonymous: REAL) return BOOLEAN;
```

```
-- function "+"      (anonymous: REAL) return REAL;
-- function "-"      (anonymous: REAL) return REAL;
-- function "abs"     (anonymous: REAL) return REAL;
```

```
-- function "+"      (anonymous, anonymous: REAL) return REAL;
-- function "-"      (anonymous, anonymous: REAL) return REAL;
-- function "*"      (anonymous, anonymous: REAL) return REAL;
-- function "/"      (anonymous, anonymous: REAL) return REAL;
```

```
-- function "***"      (anonymous: REAL; anonymous: INTEGER) return REAL;
```

-- Predefined type TIME:

type TIME is range implementation_defined

units

fs;		-- femtosecond
ps	= 1000 fs;	-- picosecond
ns	= 1000 ps;	-- nanosecond
us	= 1000 ns;	-- microsecond
ms	= 1000 us;	-- millisecond
sec	= 1000 ms;	-- second
min	= 60 sec;	-- minute
hr	= 60 min;	-- hour

end units;

-- The predefined operators for this type are as follows:

```
-- function "="      (anonymous, anonymous: TIME) return BOOLEAN;
-- function "/="     (anonymous, anonymous: TIME) return BOOLEAN;
-- function "<"      (anonymous, anonymous: TIME) return BOOLEAN;
-- function "<="    (anonymous, anonymous: TIME) return BOOLEAN;
-- function ">"      (anonymous, anonymous: TIME) return BOOLEAN;
-- function ">="    (anonymous, anonymous: TIME) return BOOLEAN;
```

```
-- function "+"      (anonymous: TIME) return TIME;
-- function "-"      (anonymous: TIME) return TIME;
-- function "abs"     (anonymous: TIME) return TIME;
```



```

-- function "+"      (anonymous, anonymous: TIME) return TIME;
-- function "-"      (anonymous, anonymous: TIME) return TIME;

-- function "*"      (anonymous: TIME;      anonymous: INTEGER) return TIME;
-- function "*"      (anonymous: TIME;      anonymous: REAL)    return TIME;
-- function "*"      (anonymous: INTEGER;    anonymous: TIME)    return TIME;
-- function "*"      (anonymous: REAL;      anonymous: TIME)    return TIME;
-- function "/"      (anonymous: TIME;      anonymous: INTEGER) return TIME;
-- function "/"      (anonymous: TIME;      anonymous: REAL)    return TIME;

-- function "/"      (anonymous, anonymous: TIME) return universal_integer;

subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;

-- A function that returns the current simulation time, TC:

impure function NOW return DELAY_LENGTH;

-- Predefined numeric subtypes:

subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

-- Predefined array types:

type STRING is array (POSITIVE range <>) of CHARACTER;

-- The predefined operators for this type are as follows:

-- function "="      (anonymous, anonymous: STRING) return BOOLEAN;
-- function "/="      (anonymous, anonymous: STRING) return BOOLEAN;
-- function "<"      (anonymous, anonymous: STRING) return BOOLEAN;
-- function "<="      (anonymous, anonymous: STRING) return BOOLEAN;
-- function ">"      (anonymous, anonymous: STRING) return BOOLEAN;
-- function ">="      (anonymous, anonymous: STRING) return BOOLEAN;

-- function "&"      (anonymous: STRING;      anonymous: STRING)
--                      return STRING;
-- function "&"      (anonymous: STRING;      anonymous: CHARACTER)
--                      return STRING;
-- function "&"      (anonymous: CHARACTER;    anonymous: STRING)
--                      return STRING;
-- function "&"      (anonymous: CHARACTER;    anonymous: CHARACTER)
--                      return STRING;

type BIT_VECTOR is array (NATURAL range <>) of BIT;

-- The predefined operators for this type are as follows:

-- function "and"     (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "or"      (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "nand"     (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "nor"      (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;

```

```

-- function "xor"      (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;
-- function "xnor"     (anonymous, anonymous: BIT_VECTOR) return BIT_VECTOR;

-- function "not"      (anonymous: BIT_VECTOR) return BIT_VECTOR;

-- function "sll"      (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                    return BIT_VECTOR;
-- function "srl"      (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                    return BIT_VECTOR;
-- function "sla"      (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                    return BIT_VECTOR;
-- function "sra"      (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                    return BIT_VECTOR;
-- function "rol"      (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                    return BIT_VECTOR;
-- function "ror"      (anonymous: BIT_VECTOR; anonymous: INTEGER)
--                    return BIT_VECTOR;

-- function "="        (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function "/="        (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function "<"         (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function "<="        (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function ">"         (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;
-- function ">="        (anonymous, anonymous: BIT_VECTOR) return BOOLEAN;

-- function "&"         (anonymous: BIT_VECTOR; anonymous: BIT_VECTOR)
--                    return BIT_VECTOR;
-- function "&"         (anonymous: BIT_VECTOR; anonymous: BIT)
--                    return BIT_VECTOR;
-- function "&"         (anonymous: BIT; anonymous: BIT_VECTOR)
--                    return BIT_VECTOR;
-- function "&"         (anonymous: BIT; anonymous: BIT)
--                    return BIT_VECTOR;

-- The predefined types for opening files:

type FILE_OPEN_KIND is (
    READ_MODE,      -- Resulting access mode is read-only.
    WRITE_MODE,     -- Resulting access mode is write-only.
    APPEND_MODE);   -- Resulting access mode is write-only; information
                    -- is appended to the end of the existing file.

-- The predefined operators for this type are as follows:

-- function "="        (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
-- function "/="        (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
-- function "<"         (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
-- function "<="        (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
-- function ">"         (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;
-- function ">="        (anonymous, anonymous: FILE_OPEN_KIND) return BOOLEAN;

```

```

type FILE_OPEN_STATUS is (
    OPEN_OK,           -- File open was successful.
    STATUS_ERROR,      -- File object was already open.
    NAME_ERROR,        -- External file not found or inaccessible.
    MODE_ERROR);       -- Could not open file with requested access mode.

-- The predefined operators for this type are as follows:

-- function "="      (anonymous, anonymous: FILE_OPEN_STATUS)
--                  return BOOLEAN;
-- function "/="      (anonymous, anonymous: FILE_OPEN_STATUS)
--                  return BOOLEAN;
-- function "<"       (anonymous, anonymous: FILE_OPEN_STATUS)
--                  return BOOLEAN;
-- function "<="      (anonymous, anonymous: FILE_OPEN_STATUS)
--                  return BOOLEAN;
-- function ">"       (anonymous, anonymous: FILE_OPEN_STATUS)
--                  return BOOLEAN;
-- function ">="      (anonymous, anonymous: FILE_OPEN_STATUS)
--                  return BOOLEAN;

-- The 'FOREIGN' attribute:

attribute FOREIGN: STRING;
end STANDARD;

```

The 'FOREIGN' attribute may be associated only with architectures or with subprograms. In the latter case, the attribute specification must appear in the declarative part in which the subprogram is declared.

Notes:

1. The ASCII mnemonics for file separator (FS), group separator (GS), record separator (RS), and unit separator (US) are represented by FSP, GSP, RSP, and USP, respectively, in type CHARACTER in order to avoid conflict with the units of type TIME.
2. The declarative parts and statement parts of design entities whose corresponding architectures are decorated with the 'FOREIGN' attribute and subprograms that are likewise decorated are subject to special elaboration rules.

F. Package TEXTIO.

Package TEXTIO contains declarations of types and subprograms that support formatted I/O operations on text files.

package TEXTIO is

-- Type definitions for text I/O:

type LINE is access STRING;	-- a LINE is a pointer to a STRING value
type TEXT is file of STRING;	-- A file of variable-length ASCII records.
type SIDE is (RIGHT, LEFT);	-- For justifying output data within fields.
subtype WIDTH is NATURAL;	-- For specifying widths of output fields.

-- Standard text files:

```
file INPUT:    TEXT open READ_OPEN "STD_INPUT";
file OUTPUT:   TEXT open WRITE_OPEN "STD_OUTPUT";
```

-- Input routines for standard types:

```
procedure READLINE (file F: TEXT; L: out LINE);
procedure READ (L: inout LINE;  VALUE: out BIT;           GOOD: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out BIT);
procedure READ (L: inout LINE;  VALUE: out BIT_VECTOR; GOOD: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out BIT_VECTOR);
procedure READ (L: inout LINE;  VALUE: out BOOLEAN;    GOOD: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out CHARACTER; GOOD: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out CHARACTER);
procedure READ (L: inout LINE;  VALUE: out INTEGER;    GOOD: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out INTEGER);
procedure READ (L: inout LINE;  VALUE: out REAL;        GOOD: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out REAL);
procedure READ (L: inout LINE;  VALUE: out STRING;      GOOD: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out STRING);
procedure READ (L: inout LINE;  VALUE: out TIME;        GOOD: out BOOLEAN);
procedure READ (L: inout LINE;  VALUE: out TIME);
```

-- Output routines for standard types:

```
procedure WRITELINE (file F: TEXT; L: inout LINE);
procedure WRITE (L: inout LINE;  VALUE: in BIT;           JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE;  VALUE: in BIT_VECTOR;    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE;  VALUE: in BOOLEAN;      JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE;  VALUE: in CHARACTER;    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE;  VALUE: in INTEGER;      JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
```

```

procedure WRITE (L: inout LINE;  VALUE:                in                REAL;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                DIGITS: in NATURAL:= 0);
procedure WRITE (L: inout LINE;  VALUE:                in                STRING;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE;  VALUE:                in                TIME;
                JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
                UNIT: in TIME:= ns);

-- File position predicate:
-- function ENDFILE (file F: TEXT) return BOOLEAN;
end TEXTIO;

```

Procedures READLINE and WRITELINE declared in package TEXTIO read and write entire lines of a file of type TEXT. Procedure READLINE causes the next line to be read from the file and returns as the value of parameter L an access value that designates an object representing that line. If parameter L contains a non-null access value at the start of the call, the object designated by that value is deallocated before the new object is created. The representation of the line does not contain the representation of the end of the line. It is an error if the file specified in a call to READLINE is not open or, if open, the file has an access mode other than read-only. Procedure WRITELINE causes the current line designated by parameter L to be written to the file and returns with the value of parameter L designating a null string. If parameter L contains a null access value at the start of the call, then a null string is written to the file. It is an error if the file specified in a call to WRITELINE is not open or, if open, the file has an access mode other than write-only.

The language does not define the representation of the end of a line. An implementation must allow all possible values of types CHARACTER and STRING to be written to a file. However, as an implementation is permitted to use certain values of types CHARACTER and STRING as line delimiters, it may not be possible to read these values from a TEXT file.

Each READ procedure declared in package TEXTIO extracts data from the beginning of the string value designated by parameter L and modifies the value so that it designates the remaining portion of the line on exit.

The READ procedures defined for a given type other than CHARACTER and STRING begin by skipping leading *whitespace characters*. A whitespace character is defined as a space, a non-breaking space, or a horizontal tabulation character (SP, NBSP, or HT). For all READ procedures, characters are then removed from L and composed into a string representation of the value of the specified type. Character removal and string composition stops when a character is encountered that cannot be part of the value, according to the lexical rules of Section 13.2 (Std. 1076-1993); this character is not removed from L and is not added to the string representation of the value. The READ procedures for types INTEGER and REAL also accept a leading sign; additionally, there can be no space between the sign and the remainder of the literal. The READ procedures for types STRING and BIT_VECTOR also terminate acceptance when VALUE'LENGTH characters have been accepted. Again using the rules of Section 13.2 (Std. 1076-1993), the accepted characters are then interpreted as a string representation of the specified type.

The READ does not succeed if the sequence of characters removed from L is not a valid string representation of a value of the specified type or, in the case of types STRING and BIT_VECTOR, if the sequence does not contain VALUELENGTH characters.

The definitions of the string representation of the value for each data type are as follows:

- The representation of a BIT value is formed by a single character, either 1 or 0. No leading or trailing quotation characters are present.
- The representation of a BIT_VECTOR value is formed by a sequence of characters, either 1 or 0. No leading or trailing quotation characters are present.
- The representation of a BOOLEAN value is formed by an identifier, either FALSE or TRUE.
- The representation of a CHARACTER value is formed by a single character.
- The representation of both INTEGER and REAL values is that of a decimal literal (see Section 13.4.1, (Std. 1076-1993)), with the addition of an optional leading sign. The sign is never written if the value is non-negative, but is accepted during a read even if the value is non-negative. No spaces can occur between the sign and the remainder of the value. The decimal point is absent in the case of an INTEGER literal and present in the case of a REAL literal. An exponent may optionally be present; moreover, the language does not define under what conditions it is or is not present. However, if the exponent is present, the “e” is written as a lower case character. Leading and trailing zeroes are written as necessary to meet the requirements of the FIELD and DIGITS parameters, and are accepted during a read.
- The representation of a STRING value is formed by a sequence of characters, one for each element of the string. No leading or trailing quotation characters are present.
- The representation of a TIME value is formed by an optional decimal literal composed following the rules for INTEGER and REAL literals described above, one or more blanks, and an identifier that is a unit of type TIME, as defined in package STANDARD (see Section 14.2, (Std. 1076-1993)). The identifier is expressed in lower case characters.

Each WRITE procedure similarly appends data to the end of the string value designated by parameter L; in this case, however, L continues to designate the entire line after the value is appended. The format of the appended data is defined by the string representations defined above for the READ procedures.

The READ and WRITE procedures for the types BIT_VECTOR and STRING respectively read and write the element values in left to right order.

For each predefined data type there are two READ procedures declared in package TEXTIO. The first has three parameters: L, the line to read from; VALUE, the value read from the line; and GOOD, a boolean flag that indicates whether the read operation succeeded or not. For example, the operation READ (L, IntVal, OK) would return with OK set to FALSE, L unchanged, and IntVal undefined if IntVal is a variable of type INTEGER and L designates the line "ABC". The success indication returned via parameter GOOD allows a process to gracefully recover from unexpected discrepancies in input format. The second form of read operation has only the parameters L and VALUE. If the requested type cannot be read into VALUE from line L, then an error occurs. Thus the operation READ (L, IntVal) would cause an error to occur if IntVal is of type INTEGER and L designates the line "ABC".

For each predefined data type there is one WRITE procedure declared in package TEXTIO. Each of these has at least two parameters: L, the line to write to; and VALUE, the value to be written. Additional parameters JUSTIFIED, FIELD, DIGITS, and UNIT control the formatting of output data. Each write operation appends data to a line formatted within a *field* that is at least as long as required to represent the data value. Parameter FIELD specifies the desired field width. Since the actual field width will always be at least large enough to hold the string representation of the data value, the default value 0 for the FIELD parameter has the effect of causing the data value to be written out in a field of exactly the right width (i.e., no leading or trailing spaces). Parameter JUSTIFIED specifies whether values are to be right- or left-justified within the field; the default is right-justified. If the FIELD parameter describes a field width larger than the number of characters necessary for a given value, blanks are used to fill the remaining characters in the field.

Parameter DIGITS specifies how many digits to the right of the decimal point are to be output when writing a real number; the default value 0 indicates that the number should be output in standard form, consisting of a normalized mantissa plus exponent (e.g., 1.079236E-23). If DIGITS is non-zero, then the real number is output as an integer part followed by '.' followed by the fractional part, using the specified number of digits (e.g., 3.14159).

Parameter UNIT specifies how values of type TIME are to be formatted. The value of this parameter must be equal to one of the units declared as part of the declaration of type TIME; the result is that the TIME value is formatted as an integer or real literal representing the number of multiples of this unit, followed by the name of the unit itself. The name of the unit is formatted using only lower case characters. Thus the procedure call WRITE(Line, 5 ns, UNIT=>us) would result in the string value "0.005 us" being appended to the string value designated by Line, whereas WRITE(Line, 5 ns) would result in the string value "5 ns" being appended (since the default UNIT value is ns).

Function ENDFILE is defined for files of type TEXT by the implicit declaration of that function as part of the declaration of the file type.

Notes:

1. For a variable L of type Line, attribute L.Length gives the current length of the line, whether that line is being read or written. For a line L that is being written, the value of L.Length gives the number of characters that have already been written to the line; this is equivalent to the column number of the last character of the line. For a line L that is being read, the value of L.Length gives the number of characters on that line remaining to be read. In particular, the expression L.Length = 0 is true precisely when the end of the current line has been reached.
2. The execution of a read or write operation may modify or even deallocate the string object designated by input parameter L of type Line for that operation; thus a dangling reference may result if the value of a variable L of type Line is assigned to another access variable and then a read or write operation is performed on L.

G. IEEE Package MATH_REAL.

G.1 Predefined Constants.

The predefined constants are all of type REAL.

Name	Value	Meaning
MATH_E	2.71828_18284_59045_23536	e
MATH_1_OVER_E	0.36787_94411_71442_32160	1/e
MATH_PI	3.14159_26535_89793_23846	π
MATH_TWO_PI	6.28318_53071_79586_47692	$2*\pi$
MATH_1_OVER_PI	0.31830_98861_83790_67154	$1/\pi$
MATH_HALF_PI	1.57079_63267_94896_61923	$\pi/2$
MATH_Q_PI	0.78539_81633_97448_30961	$\pi/4$
MATH_3_HALF_PI	4.71238_89803_84689_85769	$3*\pi/2$
MATH_LOG_OF_2	0.69314_71805_59945_30942	$\ln(2)$
MATH_LOG_OF_10	2.30258_50929_94045_68402	$\ln(10)$
MATH_LOG2_OF_E	1.44269_50408_88963_4074	$^2\log(e)$
MATH_LOG10_OF_E	0.43429_44819_03251_82765	$^{10}\log(e)$
MATH_SQRT2	1.41421_35623_73095_04880	$\sqrt{2}$
MATH_SQRT1_2	0.70710_67811_86547_52440	$\sqrt{1/2}$
MATH_SQRT_PI	1.77245_38509_05516_02730	$\sqrt{\pi}$
MATH_DEG_TO_RAD	0.01745_32925_19943_29577	conversion factor from degree to radian
MATH_RAD_TO_DEG	57.29577_95130_82320_87685	conversion factor from radian to degree

G.2 Predefined Functions.

Angles in radians.

	result	Meaning
SIGN (real)	real	returns sign
CEIL (real)	real	returns smallest integer not less
FLOOR (real)	real	returns largest integer not greater
ROUND (real)	real	returns nearest integer
TRUNC (real)	real	truncates towards zero
FMAX (real, real)	real	returns larger one
FMIN (real, real)	real	returns smaller one
SQRT (real)	real	square root
CBRT (real)	real	cube root
EXP (real)	real	e^x
LOG (real)	real	$\ln(x)$
LOG (real, positive)	real	$^{\text{base}}\log(x)$
SIN (real)	real	
COS (real)	real	
TAN (real)	real	
ARCSIN (real)	real	
ARCCOS (real)	real	
ARCTAN (real)	real	

ARCTAN (real, real)	real
SINH (real)	real
COSH (real)	real
TANH (real)	real
ARCSINH (real)	real
ARCCOSH (real)	real
ARCTANH (real)	real
ARCTANH (real, real)	real

G.3 Overloaded Operators.

Left	Op	Right	Return	Meaning
real	MOD	real	real	modulus
integer	**	real	real	x^y

G.4 Predefined Procedure.

	Meaning
UNIFORM (integer, integer, real)	pseudo-random number. Seed values seed1 and seed2 (both of type integer) should be initialized between [1, 2147483562] and [1, 2147483398] respectively.

H. IEEE Package MATH_COMPLEX.

H.1 Predefined Types.

COMPLEX is record RE, IM : REAL;
 COMPLEX_POLAR is record MAG, ARG : REAL;

H.2 Predefined Constants.

The predefined constants are all of type REAL.

CBASE_1 COMPLEX'(1.0, 0.0);
 CBASE_j COMPLEX'(0.0, 1.0);
 CZERO COMPLEX'(0.0, 0.0);

In the remainder of this section:

c complex
 cp complex_polar

H.3 Predefined Functions.

	result	Meaning
CMPLX (real, [real])	c	$x + iy$
COMPLEX_TO_POLAR (c)	cp	
POLAR_TO_COMPLEX (cp)	c	
ARG (c)	real	returns angle (radians)
ARG (cp)	real	returns angle (radians)
CONJ (c)	c	complex conjugate, $x-jy$ for $z=x+jy$
CONJ (cp)	cp	complex conjugate, (z.mag, -z.arg)
SQRT (c)	c	
SQRT (cp)	cp	
EXP (c)	c	e^z
EXP (cp)	cp	e^z
LOG (c)	c	$\ln(z)$
LOG (cp)	cp	$\ln(z)$

H.4 Overloaded Operators.

Left	Op	Right	Return
	ABS	c	real
	ABS	cp	real
	-	c	c
	-	cp	cp
c	+ - * /	c	c
real	+ - * /	c	c
c	+ - * /	real	c
cp	+ - * /	cp	cp
real	+ - * /	cp	cp
cp	+ - * /	real	cp



I. VHDL Quick Reference Card

()	Grouping	[]	Optional
{ }	Repeated		Alternative
bold	As is	CAPS	User Identifier
<i>italic</i>	VHDL-1993		

I.1 Library Units

```
[{use_clause}]
entity ID is
  [generic ({ID : TYPEID [:=expr];});]
  [port ({ID : in | out | inout TYPEID [:=expr];});]
  [{declaration}]
begin
  [{parallel_statement}]
end [entity] ENTITYID;

[{use_clause}]
architecture ID entity ENTITYID is
  [{declaration}]
begin
  [{parallel_statement}]
end [architecture] ARCHID;

[{use_clause}]
package ID is
  [{declaration}]
end [package] PACKID;

[{use_clause}]
package body ID is
  [{declaration}]
end [package body] PACKID;

[{use_clause}]
configuration ID entity ENTITYID is
for ARCHID
  [{block_config | comp_config}]
end for;
end [configuration] CONFID;

use_clause::=
  library ID;
  [{use LIBID.PKGID.all;}]

block_config::=
for LABELID
  [{block_config | comp_config}]
end for;

comp_config::=
```

```
for all | LABELID : COMPID
  (use entity [LIBID.]ENTITYID [( ARCHID )]
    [[generic map( {GENID => expr,} )]
    port map ( {PORT ID => SIGID | expr,} ) ];
  [for ARCHID
    [{block_config | comp_config}]
  end for; ]
end for; |
  (use configuration [LIBID.]CONFID
    [[generic map( {GENID => expr,} )]
    port map ( {PORTID => SIGID | expr,} ) ];
end for;
```

I.2 Declarations

I.2.1 Type Declarations

```
type ID is ( {ID,} );

type ID is range number downto | to number ;

type ID is array ( {range | TYPEID,})
  of TYPEID | SUBTYPID ;

type ID is record
  {ID : TYPEID ;}
end record ;

type ID is access TYPEID ;

type ID is file of TYPEID ;

subtype ID is SCALARTYPID [ range range ] ;

subtype ID is ARRAYTYPID [ ( {range,} ) ] ;

subtype ID is RESOLVFCTID TYPEID;

range::=
  (integer | ENUMID to | downto
    integer | ENUMID) | OBJID'[reverse_]range) |
  (TYPEID range <> )
```

I.2.2 Other Declarations

```
constant ID : TYPEID := expr;

[shared] variable ID : TYPEID [:= expr];

signal ID : TYPEID [:= expr];

file ID : TYPEID (is in | out string ;) |
  (open read_mode | write_mode
  / append_mode is string;) )
```

alias ID : TYPEID **is** OBJID ;

attribute ID : TYPEID ;

attribute ATTRIB **of** OBJID | **others** | **all** : class
is expr ;

class::=
entity | **architecture** | **configuration** |
procedure | **function** | **package** | **type** |
subtype | **constant** | **signal** | **variable** |
component | **label**

component ID [*is*]
[**generic** ({ID : TYPEID [:=expr];});]
[**port** ({ID : **in** | **out** | **inout** TYPEID [:=expr];});]
end component [*COMPID*];

[*impure* / [*pure*]] **function** ID
[({ [**constant** | **variable** | **signal** | *file*] ID :
in | **out** | **inout** TYPEID [:=expr];});]
return TYPEID [*is*]
begin
{sequential_statement}
end [*function*] ID];

procedure ID
[({ [**constant** | **variable** | **signal**] ID :
in | **out** | **inout** TYPEID [:=expr];});]
[**is begin**
[{sequential_statement}]
end [*procedure*] ID];

for LABELID | **others** | **all** : COMPID **use**
(**use entity** [LIBID.]ENTITYID [(ARCHID)]) |
(**use configuration** [LIBID.]CONFID
[[**generic map** ({GENID => expr,})]
port map ({PORTID => SIGID | *expr*,})]);
end for;

I.3 Expressions

expression::=
(relation **and** relation) |
(relation **or** relation) |
(relation **xor** relation) |
(relation **nand** relation) |
(relation **nor** relation) |
(*relation xnor relation*)

relation::= sexpr [relop shexpr]

shexpr::= sexpr [*shop sexpr*]

sexpr::= [+|-] term {addop term}

term::= factor {mulop factor}

factor::= (prim[**prim]) | (**abs** prim) | (**not** prim)

prim::=
literal | OBJID | OBJID'ATTRIB
| OBJID({expr,})
| OBJID(range) | ({[choice [{ choice }] =>] expr,})

| FCTID({[PARID =>] expr, }) | TYPEID'(expr)
| TYPEID(expr) | **new** TYPEID['(expr)] | (expr)

choice::= sexpr | range | RECFID | **others**

I.3.1 Operators, Increasing Precedence

logop **and** | **or** | **xor** | **nand** | **nor** | *xnor*
relop = | /= | < | <= | > | >=
shop *sll* / *srl* / *sla* / *sra* / *rol* / *ror*
addop + | - | &
mulop * | / | **mod** | **rem**
miscop ** | **abs** | **not**

I.4 Sequential Statements

wait [**on** {SIGID,}] [**until** expr] [**for** time];

assert expr
[**report** string] [severity note | **warning** | **error** |
failure];

report string [severity note / *warning* / *error* /
failure];

SIGID <= [**transport**] | [[**reject** TIME] **inertial**]
{expr [**after** time]};

VARID := expr;

PROCEDUREID [({ [PARID =>] expr, })];

[*LABEL*:] **if** expr **then**
{sequential_statement}
[**elsif** expr **then**
{sequential_statement}]
[**else**
{sequential_statement}]
end if [*LABEL*] ;

[*LABEL*:] **case** expr **is**
{ **when** choice [{ choice }] =>
{sequential_statement} }
end case [*LABEL*] ;

[*LABEL*:] [**while** expr] **loop**
{sequential_statement}
end loop [*LABEL*] ;

[*LABEL*:] **for** ID **in** range **loop**
{sequential_statement}
end loop [*LABEL*] ;

[*LABEL*:] **next** [LOOPLBL] [**when** expr] ;

[*LABEL*:] **exit** [LOOPLBL] [**when** expr] ;

[*LABEL*:] **return** [expression] ;

[*LABEL*:] **null** ;

I.5 Parallel Statements

LABEL: **block** [*is*]
[**generic** ({ID : TYPEID; }) ;

```

    [generic map ( { GENID => expr, } ); ]
    [port ( { ID : in | out | inout TYPEID } );
    [port map ( { [PORTID =>] SIGID | expr, } ); ]
    [{ declaration }]
begin
    [{ parallel_statement }]
end block [LABEL];

[LABEL:] [postponed] process [(SIGID, ) ]
    [{ declaration }]
begin
    [{ sequential_statement }]
end [postponed] process [ LABEL ];

[LABEL:] [postponed]
    PROCID ( [{ PARID=>} expr, );

[LABEL:] [postponed] assert expr
    [report string] [severity note | warning | error |
    failure ];

[LABEL:] [postponed] report string [severity note /
    warning / error / failure ];

[LABEL:] [postponed] SIGID <=
    [ transport ] | [ reject TIME inertial ]
    [ { { expr [after TIME] } | unaffected when expr
    else } ] { { expr [after TIME] } | unaffected };

[LABEL:] [postponed] with expr select
    SIGID <= [ transport ] | [ [ reject TIME ] inertial ]
    { { expr [after TIME] } |
    unaffected when choice [ { choice } ] };

LABEL: COMPID
    [ [generic map ( { GENID => expr, } ) ]
    port map ( { [ PORTID => ] SIGID | expr, } ); ]

LABEL: entity [ LIBID. ] ENTITYID [ ( ARCHID ) ]
    [ [generic map ( { GENID => expr, } ) ]
    port map ( { PORTID => SIGID | expr, } ); ]

LABEL: configuration [ LIBID. ] CONFID
    [ [generic map ( { GENID => expr, } ) ]
    port map ( { PORTID => SIGID | expr, } ); ]

LABEL: if expr generate
    [{ parallel_statement }]
end generate [LABEL];

LABEL: for ID in range generate
    [{ parallel_statement }]
end generate [LABEL];

```

I.6 Predefined Attributes

TYPEID'base	Base type
TYPEID'left	Left-bound value
TYPEID'right	Right-bound value
TYPEID'high	Upper-bound value
TYPEID'low	Lower-bound value
TYPEID'pos(expr)	Position within type
TYPEID'val(expr)	Value at position

TYPEID'succ(expr)	Next value in order
TYPEID'pred(expr)	Previous value in order
TYPEID'leftof(expr)	Value to the left in order
TYPEID'rightof(expr)	Value to the right in order
TYPEID'ascending	Ascending type predicate
TYPEID'image(expr)	String image of value
TYPEID'value(expr)	Value of string image
ARYID'left [(expr)]	Left-bound of [nth] index
ARYID'right [(expr)]	Right-bound of [nth] index
ARYID'high [(expr)]	Upper-bound of [nth] index
ARYID'low [(expr)]	Lower-bound of [nth] index
ARYID'range [(expr)]	'left downto / to 'right
ARYID'reverse_range [(expr)]	'right downto / to 'left
ARYID'length [(expr)]	Length of [nth] dimension
ARYID'ascending[(expr)]	'right >= 'left ?
SIGID'delayed [(TIME)]	Delayed copy of signal
SIGID'stable [(TIME)]	Signals event of signal
SIGID'quiet [(TIME)]	Signals activity of signal
SIGID'transaction	Toggles if signal active
SIGID'event	Event on signal ?
SIGID'active	Activity on signal ?
SIGID'last_event	Time since last event
SIGID'last_active	Time since last active
SIGID'last_value	Value before last event
SIGID'driving	Active driver predicate
SIGID'driving_value	Value of driver
OBJID'simple_name	Name of object
OBJID'instance_name	Pathname of object
OBJID'path_name	Pathname to object

I.7 Predefined Types

BOOLEAN	True or false
INTEGER	minimal range -2147483647 to 2147483647
NATURAL	INTEGERS >= 0
POSITIVE	INTEGERS > 0
REAL	Floating-point
BIT	'0', '1'
BIT_VECTOR(NATURAL Array of bits)	
CHARACTER	7-bit ASCII
STRING(POSITIVE)	Array of characters
TIME	hr, min, sec, ms, us, ns, ps, fs
DELAY_LENGTH	TIME >= 0

I.8 Predefined Functions

Now	Returns current simulation time
Deallocate(ACCESS_TYPOBJ)	Deallocate dynamic object
FILE_OPEN([status,] FILEID, string, mode)	Open file
FILE_CLOSE(FILEID)	Close file

I.9 Lexical Elements

identifier ::= letter { [underline] alphanumeric }

decimal literal ::= integer [. integer] [E[+|-] integer]

based literal ::=

integer # hexint [. hexint] # [E[+|-] integer]

bit string literal ::= **B** | **O** | **X** “hexint”

comment ::= -- comment text

Qualis Design Corporation.

Beaverton, OR USA

Phone: +1 - 503 - 531 - 0377

Fax: +1 - 503 - 629 - 5525

Email: info@qualis.com



J. 1164 PACKAGES QUICK REFERENCE CARD

()	Grouping	[]	Optional
{ }	Repeated		Alternative
bold	As is	CAPS	User Identifier
<i>italic</i>	VHDL-1993		
b	::=	BIT	
u/l	::=	STD_ULOGIC/STD_LOGIC	
bv	::=	BIT_VECTOR	
uv	::=	STD_ULOGIC_VECTOR	
lv	::=	STD_LOGIC_VECTOR	
un	::=	UNSIGNED	
sg	::=	SIGNED	
na	::=	NATURAL	
in	::=	INTEGER	
sm	::=	SMALL_INT	
		(subtype INTEGER range 0 to 1)	
c	::=	commutative	

J.1 IEEE's STD_LOGIC_1164

J.1.1 Logic Values

'U'	Uninitialized
'X'/'W'	Strong/Weak unknown
'0'/'L'	Strong/Weak 0
'1'/'H'	Strong/Weak 1
'Z'	High Impedance
'-'	Don't care

J.1.2 Predefined Types

STD_ULOGIC	Base type
Subtypes:	
STD_LOGIC	Resolved STD_ULOGIC
X01	Resolved X, 0 & 1
X01Z	Resolved X, 0, 1 & Z
UX01	Resolved U, X, 0, 1
UX01Z	Resolved U, X, 0, 1 & Z
STD_ULOGIC_VECTOR (na to downto na)	Array of STD_ULOGIC
STD_LOGIC_VECTOR (na to downto na)	Array of STD_LOGIC

J.1.3 Overloaded Operators

Description	Left	Operator	Right
bitwise-and	u/l,uv,lv	and	u/l,uv,lv
bitwise-or	u/l,uv,lv	or	u/l,uv,lv
bitwise-nand	u/l,uv,lv	nand	u/l,uv,lv
bitwise-nor	u/l,uv,lv	nor	u/l,uv,lv
bitwise-xor	u/l,uv,lv	xor	u/l,uv,lv
<i>bitwise-xnor</i>	<i>u/l,uv,lv</i>	<i>xnor</i>	<i>u/l,uv,lv</i>
bitwise-not		not	u/l,uv,lv

J.1.4 Conversion Functions

From	To	Function
u/l	b	TO_BIT (from, [xmap])
uv,lv	bv	TO_BITVECTOR (from, [xmap])
b	u/l	TO_STDULOGIC (from, [xmap])
bv,uv	lv	TO_STDLOGICVECTOR (from)
bv,lv	uv	TO_STDULOGICVECTOR (from)

J.2 IEEE's NUMERIC_STD

J.2.1 Predefined Types

UNSIGNED(na to downto na)	Array of STD_LOGIC
SIGNED(na to downto na)	Array of STD_LOGIC

J.2.2 Overloaded Operators

Left	Op	Right	Return
	abs	sg	sg
	-	sg	sg
un	+, -, *, /, rem , mod	un	un
sg	+, -, *, /, rem , mod	sg	sg
un	+, -, *, /, rem , mod _c	na	un
sg	+, -, *, /, rem , mod _c	in	sg
un	<, >, <=, >=, =, /=	un	bool
sg	<, >, <=, >=, =, /=	sg	bool
un	<, >, <=, >=, =, /= _c	na	bool
sg	<, >, <=, >=, =, /= _c	in	bool

J.2.3 Predefined Functions

SHIFT_LEFT (un, na)	un
SHIFT_RIGHT (un, na)	un
SHIFT_LEFT (sg, na)	sg
SHIFT_RIGHT (sg, na)	sg
ROTATE_LEFT (un, na)	un
ROTATE_RIGHT (un, na)	un
ROTATE_LEFT (sg, na)	sg
ROTATE_RIGHT (sg, na)	sg
RESIZE (un, na)	un
RESIZE (sg, na)	sg
STD_MATCH (u/l, u/l)	bool
STD_MATCH (ul, ul)	bool
STD_MATCH (lv, lv)	bool
STD_MATCH (un, un)	bool
STD_MATCH (sg, sg)	bool

J.2.4 Conversion Functions

From	To	Function
un,lv	sg	SIGNED (from)
sg,lv	un	UNSIGNED (from)
un,sg	lv	STD_LOGIC_VECTOR (from)
un,sg	in	TO_INTEGER (from)
na	un	TO_UNSIGNED (from, size)
in	sg	TO_SIGNED (from, size)

J.3 IEEE's NUMERIC_BIT

J.3.1 Predefined Types

UNSIGNED(na to | **downto** na)
Array of BIT

SIGNED(na to | **downto** na)
Array of BIT

J.3.2 Overloaded Operators

Left	Op	Right	Return
	abs	sg	sg
	-	sg	sg
un	+, -, *, /, rem , mod	un	un
sg	+, -, *, /, rem , mod	sg	sg
un	+, -, *, /, rem , mod _c	na	un
sg	+, -, *, /, rem , mod _c	in	sg
un	<, >, <=, >=, =, /=	un	bool
sg	<, >, <=, >=, =, /=	sg	bool
un	<, >, <=, >=, =, /= _c	na	bool
sg	<, >, <=, >=, =, /= _c	in	bool

J.3.3 Predefined Functions

SHIFT_LEFT (un, na)	un
SHIFT_RIGHT (un, na)	un
SHIFT_LEFT (sg, na)	sg
SHIFT_RIGHT (sg, na)	sg
ROTATE_LEFT (un, na)	un

ROTATE_RIGHT (un, na)	un
ROTATE_LEFT (sg, na)	sg
ROTATE_RIGHT (sg, na)	sg
RESIZE (sg, na)	sg
RESIZE (un, na)	un

J.3.4 Conversion Functions

From	To	Function
un,bv	sg	SIGNED (from)
sg,bv	un	UNSIGNED (from)
un,sg	bv	BIT_VECTOR (from)
un,sg	in	TO_INTEGER (from)
na	un	TO_UNSIGNED (from)
in	sg	TO_SIGNED (from)

J.4 Synopsys' STD_LOGIC_ARITH

J.4.1 Predefined Types

UNSIGNED(na to | **downto** na)
Array of STD_LOGIC

SIGNED(na to | **downto** na)
Array of STD_LOGIC

SMALL_INT Integer, 0 or 1

J.4.2 Overloaded Operators

Left	Op	Right	Return
	abs	sg	sg,lv
	+	un	un,lv
	+, -	sg	sg,lv
un	+, -, *, /	un	un,lv
sg	+, -, *, /	sg	sg,lv
sg	+, -, *, / _c	un	sg,lv
un	+, - _c	in	un,lv
sg	+, - _c	in	sg,lv
un	+, - _c	u/l	un,lv
sg	+, - _c	u/l	sg,lv
un	<, >, <=, >=, =, /=	un	bool
sg	<, >, <=, >=, =, /=	sg	bool
un	<, >, <=, >=, =, /= _c	in	bool
sg	<, >, <=, >=, =, /= _c	in	bool

J.4.3 Predefined Functions

SHL (un, un)	un	
SHR (un, un)	un	
SHL (sg, un)	sg	
SHL (sg, un)	sg	
EXT (lv, in)	un	zero-extend
SEXT (lv, in)	un	sign-extend

J.4.4 Conversion Functions

From	To	Function
un,lv	sg	SIGNED (from)
sg,lv	un	UNSIGNED (from)
sg,un	lv	STD_LOGIC_VECTOR (from)
un,sg	in	CONV_INTEGER (from)
in,un,sg,u	un	CONV_UNSIGNED (from, size)
in,un,sg,u	un	CONV_SIGNED (from, size)
in,un,sg,u	lv	CONV_STD_LOGIC_VECTOR (from, size)

J.5 Synopsys' STD_LOGIC_MISC

J.5.1 Predefined Functions

AND_REDUCE (lv uv)	u/l
OR_REDUCE (lv uv)	u/l
XOR_REDUCE (lv uv)	u/l

J.6 Synopsys' STD_LOGIC_UNSIGNED

J.6.1 Overloaded Operators

Left	Op	Right	Return
	+	lv	lv
lv	+, -, *	lv	lv
lv	+, - _c	in	lv
lv	+, - _c	u/l	lv
lv	<, >, <=, >=, /=	lv	bool
lv	<, >, <=, >=, /= _c	in	bool

J.6.2 Conversion Functions

From	To	Function
lv	in	CONV_INTEGER (from)

J.7 Synopsys' STD_LOGIC_SIGNED

J.7.1 Overloaded Operators

Left	Op	Right	Return
	abs	lv	lv
	+, -	lv	lv
lv	+, -, *	lv	lv
lv	+, - _c	in	lv
lv	+, - _c	u/l	lv
lv	<, >, <=, >=, /=	lv	bool
lv	<, >, <=, >=, /= _c	in	bool

J.7.2 Conversion Functions

From	To	Function
lv	in	CONV_INTEGER (from)

J.8 Synopsys' STD_LOGIC_TEXTIO

J.8.1 Overloaded Operators

Read/write binary values

READ(line, u/l [,good])
READ(line, uv [,good])
READ(line, lv [,good])
WRITE(line, u/l [,justify] [,width])
WRITE(line, uv [,justify] [,width])
WRITE(line, lv [,justify] [,width])

Read/write octal values

OREAD(line, uv [,good])
OREAD(line, lv [,good])
OWRITE(line, uv [,justify] [,width])
OWRITE(line, lv [,justify] [,width])

Read/write hexadecimal values

HREAD(line, uv [,good])
HREAD(line, lv [,good])
HWRITE(line, uv [,justify] [,width])
HWRITE(line, lv [,justify] [,width])

J.9 Cadence's STD_LOGIC_ARITH

J.9.1 Overloaded Operators

Left	Op	Right	Return
	+	uv	uv
	+	lv	lv
u/l	+, -, *, /	u/l	u/l
lv	+, -, *, /	lv	lv
lv	+, -, *, / _c	u/l	lv
lv	+, - _c	in	lv
uv	+, -, *	uv	uv
uv	+, -, * _c	u/l	uv
uv	+, - _c	in	uv
lv	<, >, <=, >=, /= _c	in	bool
uv	<, >, <=, >=, /= _c	in	bool

J.9.2 Predefined Functions

C-like ?: replacements:

COND_OP(bool, lv, lv) lv
COND_OP(bool, uv, uv) uv
COND(bool, u/l, u/l) u/l

Shift operations:

SH_LEFT(lv, na) lv
SH_LEFT(uv, na) uv
SH_RIGTH(lv, na) lv
SH_RIGHT(uv, na) uv

Resize functions:

ALIGN_SIZE (lv, na)	lv
ALIGN_SIZE (uv, na)	uv
ALIGN_SIZE (u/l, na)	lv
ALIGN_SIZE (u/l, na)	uv

J.9.3 Conversion Functions

From	To	Function
lv,uv,u/l	in	TO_INTEGER (from)
	lv	TO_STDLOGICVECTOR (from, size)
	uv	TO_STDULOGICVECTOR (from, size)

<p>Qualis Design Corporation. Beaverton, OR USA Phone: +1 - 503 - 531 - 0377 Fax: +1 - 503 - 629 - 5525 Email: info@qualis.com</p>
--

- " , 31, 38, 87
- %, 87
- <>, 31
- access, 28, 78
- active, 37
- actuele poorten, 27
- after, 54
- alias, 32, 102
- analyse, 47
 - volgorde, 47
- AND, 39
- architecture, 5, 6
- array, 30
 - constrained, 31
 - unconstrained, 31, 32
- ascending, 105
- assert statement, 19
 - concurrent, 21, 22
 - report, 99
- assignment
 - signal, 8, 13, 21
 - variable, 14
- association
 - named, 27, 31, 32
 - positional, 27, 32
- association list, 97
- attribute, 35, 104
 - active, 37
 - ascending, 105
 - behaviour, 104
 - delayed, 37
 - driving, 105
 - driving_value, 105
 - event, 36
 - foreign, 104
 - gedefinieerd door gebruiker, 87
 - high, 35
 - image, 105
 - instance_name, 105
 - last_active, 37
 - last_event, 37
 - last_value, 37
 - left, 35
 - low, 35
 - path_name, 105
 - quiet, 37
 - range, 35
 - reverse_range, 36
 - right, 35
 - simple_name, 105
 - stable, 37
 - structure, 104
 - transaction, 36
 - value, 105
- B, 31, 96
- besturingsstructuren
 - concurrent
 - conditional, 100
 - conditioneel, 21
 - selected, 21
 - sequentieel, 16
 - conditoneel, 16
 - iteratief, 18
 - for, 18, 36
 - loop, 18
 - while, 18
- bit string literal, 96
- bit_vector, 31
- block statement, 71
- boolean, 28
- buffer mode, 25, 84
- bus, 62, 64, 69
- BUS_RESOLVE, 61
- case statement, 17
- communicatie
 - handshake, 65, 67, 68
- compilatie. zie: analyse
- componenten, 23, 41
 - instantiatie, 96
- composite, 28, 30
- concurrent
 - procedure call, 58
- concurrent assert statement, 21, 22
- concurrent procedure call, 21, 58
- concurrent signal assignment, 13, 21, 100
- conditional signal assignment statement, 21
- configuratie, 49
- configuratie-declaratie, 49
- configuratie-specificatie, 24, 49, 72
 - default, 24, 49
- configuration
 - overruling, 97
- constanten, 14
- constrained, 31
- context clause, 43
- conversie
 - binair, 31, 96
 - hexadecimaal, 31, 96
 - octaal, 31, 96
- data type
 - access, 28
 - composite, 28, 30
 - array, 31
 - record, 30
 - file, 28

- volgens std. 1076-1993, 103
- scalar, 28
 - enumeration, 28
 - floating point, 28
 - fysieke, 29
 - integer, 28
 - subtypes, 28, 33, 60
- data-bus, 65
- deallocate, 79
- declaratie, 13, 29, 32
 - componenten, 23, 41
 - configuratie-specificatie, 49
 - constanten, 13, 14
 - file
 - volgens Std 1076-1993, 103
 - signalen, 7, 14
 - variabelen, 13, 14
- decompositie, 72
- default
 - delay, 54
 - initialisatie
 - signalen, 7, 31
 - variabelen, 31
- deferred constant declaration, 43
- delay
 - default, 54
 - delta, 12, 48, 53
 - inertial, 48, 53, 54, 99
 - transport, 48, 53, 54, 99
- delayed, 37
- delta-delay, 12, 48, 53
- design unit, 47
- disconnect, 77
- disconnection specification, 76
- documentatie, 7
- DRIE_NIVO_LOGICA, 41
- driver, 54, 59, 63
- driving, 105
- driving_value, 105
- elaboration, 49, 111
- entity, 5, 77
- enumeration, 28
- event, 36
- event-driven simulatie, 5, 9
- exit, 19
- expliciete disconnect, 77
- FACULTEIT, 18
 - EXIT_LUS, 18
 - FOR_LUS, 18, 22
 - WHILE_LUS, 18
- FEEDBACK, 9
- file, 28, 80, 81
 - volgens Std 1076-1993, 103
- FLIPFLOP, 15
- GEDRAG, 15

- floating point, 28, 34
- for, 18, 36
- foreign, 104
- formele poorten, 27
- functie, 20, 55, 58
 - impure, 95
 - pure, 95
 - resolutie, 60, 71
 - type-conversie, 83
- fysieke, 29, 34
- geformateerde file, 80
- generate statement, 26, 101
- generic, 5, 72
- generic map, 23, 24, 27, 72
- globale variabele, 92
- group, 96
- guarded, 74
- guarded signal, 62
- guarded signal assignment, 74
- handshake, 65, 67, 68
- hierarchie, 23
- high, 35
- hold, 77
- identifiers, 102
 - extended, 102
- ieee, 43
- if then else, 16
- image, 105
- impliciet disconnect, 77
- impliciete
 - functiedeclaratie, 40
 - signalen, 36
 - wait, 16
- impliciete functies, 38
- impure function, 95
- in mode, 25
- incomplete
 - constante-declaratie, 43
 - type-declaratie, 78
- inertial-delay, 48, 53, 54, 99
- infix, 38, 40
- initialisatie
 - signalen, 7, 31
 - variabelen, 31
- inout mode, 25
- instance_name, 105
- instantiatie, 23, 27, 72, 96
- integer, 28, 34
- interface, 48
- iteraties, 10
 - oneindige herhaling, 11
- karakterset*, 28
 - ASCII, 102

- ISO 8859-1, 102
- label, 13, 19, 23, 99
- last_active, 37
- last_event, 37
- last_value, 37
- left, 35
- library, 40, 43
- linkage mode, 25, 86
- locaal statisch, 18
- logische operator, 35
- loop, 18
- looplabel, 19
- low, 35
- MAXIMUM, 21
 - MAX_FUNCTION, 21
 - MAX_PROCEDURE, 21, 58
- mode, 83, 84
 - buffer, 25, 84
 - FOUT_GEBRUIK, 25
 - in, 25, 83
 - inout, 25, 83, 84
 - linkage, 25, 86
 - out, 25, 83
- MODES, 25
- multiple sources, 59
- MULTIPLEXER, 17
 - CONDITIONAL_SIGNAL, 22
 - GEDRAG1, 17
 - GEDRAG2, 17
 - SELECTED_SIGNAL, 22
- N_OUTPUT_BUFFERS, 27, 52
 - STRUKTUUR, 27, 52
- named association, 27, 31, 32
- natural, 31, 33
- new, 79
- NOR_GATE, 5
 - GEDRAG, 5
- now
 - impure, 95
- null, 63
- null-transaction, 63
- O, 31, 96
- open, 27, 72
- operator, 34, 98
 - infix, 38, 40
 - logische, 35
 - schuif, 98
- out mode, 25
- output_buffer, 24, 27
 - GEDRAG, 27
- overloading, 30, 38
- package, 20, 40
- package body, 40
- passieve processen, 77
- path_name, 105
- pointer. zie: access
- poorten
 - actueel, 27
 - formeel, 27
- port, 72
- port map, 23, 24, 27, 28, 72
- positional association, 27, 32
- positive, 31, 33
- postponed proces, 100
- primary unit, 29
- procedōre, 57
- procedure, 20, 55, 57
 - concurrent, 58
 - sequential, 58
- proces
 - besturingsstructuren, 16
 - expliciet, 12
 - impliciet, 13, 21
 - postponed, 100
- processen
 - communicerende, 5, 8, 12
 - passieve, 77
- pure function, 95
- qualified expression, 17
- quiet, 37
- random getallen, 94
- range, 31, 35
- record, 30
- regelmatige structuren, 26
- register, 62, 64
- reject, 100
- report statement, 99
- resolutiefunctie, 60, 69
 - eisen gesteld aan, 71
- RESOLVE, 61
- RESOLVE_BIT, 64
- resolved signal, 60
- reverse_range, 36
- right, 35
- scalar, 28
- schuifoperator, 98
- secondary unit, 29
- selected signal assignment statement, 21
- sensitivity list, 15, 16
- sequential
 - procedure call, 58
- setup, 77
- severity_level, 19, 29
- shared variabele, 92
- signal assignment, 8
- signalen, 8, 14

- guarded, 62
 - bus, 62, 64, 69
 - register, 62, 64
 - resolved, 60
- simple_name, 105
- simulatie, 5, 8, 9
- simulatie cyclus, 8
- slice, 32
- source, 59, 60
 - multiple, 59
- spanning, 30, 39
- SR_LATCH, 7, 25
 - BIJNA_STRUKTUUR, 23
 - CONF_1, 50
 - GEDRAG, 7
 - GEDRAG1, 13
- stable, 37
- std_logic, 111
- std_logic_1164, 111
- std_ulogic, 111
- string, 31
- stroom, 30, 39
- structuurbeschrijving, 23
- subprogram, 38, 49, 55
- subtypes, 28, 33, 60

- tekst file, 81
- time, 29
- transaction, 8, 36
- transport, 54
 - transport-delay, 48, 53, 54, 99
 - TRI-STATE, 61, 62
 - TRI-STATES
 - DEMO, 62
 - type-conversie-functie, 83

 - unaffected, 101
 - unconstrained, 31, 32
 - use-clause, 40

 - value, 105
 - variabelen, 14
 - globaal, 92
 - shared, 92
 - vital, 3
 - volgorde analyse, 47

 - wait, 15
 - for, 15
 - impliciet, 16
 - on, 14
 - until, 14
 - waveform, 8
 - weerstand, 30, 39
 - while, 18
 - WIRED_OR, 60

 - X, 31, 96