Assistance à l'Abstraction de Composants Virtuels pour la Vérification Rapide de Systèmes Numériques

W. Muhammad

To cite this version:

W. Muhammad. Assistance à l'Abstraction de Composants Virtuels pour la Vérification Rapide de Systèmes Numériques. Autre. Université Nice Sophia Antipolis, 2008. Français. NNT : . tel-00454617

HAL Id: tel-00454617
https://theses.hal.science/tel-00454617

Submitted on 9 Feb 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Mention : Informatique

présentée et soutenu par

Waseem MUHAMMAD

Assistance à l'Abstraction de Composants Virtuels pour la Vérification Rapide de Systèmes Numériques

Thèse dirigée par *Michel AUGUIN*

soutenue le *(19-12-2008)*

Jury :

M.	Jean-Marc DELOSME	Titre	Professeur	fonction	Président
Mme.	Laurence PIERRE	Titre	Professeur	fonction	Rapporteur
Mme.	Lirida NAVINER	Titre	Docteur	fonction	Rapporteur
Mme.	Sophie Coudert	Titre	Docteur	fonction	Examinateur
M.	Daniel GAFFE	Titre	Docteur	fonction	Examinateur

Acknowledgments

This work couldn't have been accomplished without the auspicious guidance of my research adviser Dr. Sophie Coudert, my PhD director Prof. Dr. Michel Auguin and co-advisers Mr. Renaud Pacalet and Dr. Rabéa Ameur-Boulifa who guided me throughout the course of this research challenge. Especially, I would like to thank my research adviser Dr. Sophie Coudert who kept an eye on the progress of my work and was always available whenever I needed her advices.

I express my sincere gratitude to my Ph.D reporters Prof. Dr. Laurence Pierre and Dr. Lírida Naviner to take interest in my work and spare their precious time for critically reviewing the thesis.

I am grateful to Dr. Danielle Gaffé who has helped me during reviewing and proof reading the final manuscript with a lot of technical improvements. I am thankful to all the members of Sys2RTL/CIM-PACA project team in particular Robert de Simone (INRIA Sophia Antipolis) and Yves Leduc (Texas Instruments, France) for sharing their thought-provoking ideas during discussion and talks with me time to time.

I am also indebted to Dr. M. Riaz Suddle (SI), Member SE, IST Lahore and Mr. Shakeel Zahid, General Manager IST Lahore for their unprecedented encouragement and support to continue doctoral studies abroad.

Continuous encouragement, moral support and prayers from all the family in particular my parents and my wife was always with me thousands of miles away from my homeland. I wish and pray for their lifelong happiness, safety and prosperity.

Thanks to all my friends and colleagues especially Najam, Farooq and Rizwan for their invigorating company.

Dedication

To my beloved parents, all the respected teachers who have ever taught me and my beloved wife.

Contents

1 Introduction **17**
1.1 État de l'art . . . 17
1.1.1 Vérification par simulation . . . 18
1.1.2 Vérification par model checking . . . 18
1.1.3 Les techniques d'abstraction . . . 19
1.2 Principaux objectifs de cette thèse . . . 20
1.2.1 Motivation . . . 21
1.2.2 Cadre de travail . . . 21
1.2.3 Structure et organisation de la thèse . . . 22

2 Separating Control and Data in hardware modules **23**
2.1 Introduction . . . 23
2.1.1 Basic idea . . . 24
2.1.2 Problematics . . . 25
2.2 State of the art . . . 25
2.2.1 Control and data flows in software . . . 25
2.2.2 Control and data flows in hardware . . . 26
2.3 Conceptual behavior of the "Control" . . . 28
2.3.1 Basic intuition . . . 28
2.3.2 Definition of Control . . . 30
2.3.3 Illustration of the definition . . . 31
2.4 Analysis without designer's intervention . . . 33
2.4.1 Semantics $\mathcal{S}_0$ . . . 34
2.4.2 Semantics $\mathcal{S}_1$ . . . 35
2.4.3 Semantics $\mathcal{S}_2$ . . . 36
2.4.4 Concluding remarks . . . 38
2.5 Analysis with designer's intervention . . . 38
2.6 Practical considerations of the definition of control . . . 41
2.7 Consequences of the research . . . 43

3 Control and Data separation using slicing **45**
3.1 Introduction . . . 45
3.2 Program slicing basics . . . 45
3.2.1 Dependence graph based slicing . . . 46
3.2.2 Slicing in hardware description languages . . . 47
3.3 Control-data separation using slicing . . . 48

3.3.1 VHDL description of modules 49
3.3.2 Basic slicing rules 51
3.3.3 Intermediate representation for VHDL modules 52
3.3.4 Slicing algorithm 54
3.4 Illustration of control-data slicing 55
3.5 Slicing modules with local variables 58
3.5.1 VHDL transformation 59
3.5.2 Step I : Transformation algorithm 60
3.5.3 Step II: Enhancement algorithm 62
3.5.4 Impact of VHDL transformation 66
3.6 Implementation of control-data slicing 66
3.7 Applications of control-data slicing 67
3.7.1 Assisting model checking by slicing 67
3.7.2 Assisting simulation by slicing 69
3.7.3 Miscellaneous applications of Control Data separation 75
3.8 Conclusions 75

4 Data dependency analysis using Significance 77
4.1 Introduction 77
4.1.1 Significance and intentionality of data 77
4.1.2 Boolean data dependencies 79
4.2 State of the art 79
4.3 Theoretical description of Significances 80
4.3.1 Dynamics of the module 80
4.3.2 Boolean data dependencies in modules 81
4.3.3 Approximations of static and dynamic dependencies 82
4.4 Realization of significances 84
4.4.1 Significance realization at low level 84
4.4.2 Correctness of the Realizations 86
4.4.3 Limitation of approximate significances 91
4.4.4 Significance realization at high level 92
4.4.5 Simplification of module 93
4.5 Refinements in significance computation 94
4.5.1 Refined significance : $\mathcal{S}_{ref}$ 94
4.5.2 Semantic computation of Significance 96
4.6 Applications of Significance 96
4.7 Conclusions 97

5 Verification based on Significance 99
5.1 Introduction 99
5.1.1 Significance and timing 99
5.1.2 Module's behavior and designer's anticipation 100
5.2 Description of verification technique 103
5.2.1 Automatic transformations 104
5.2.2 Specification of significance property 107
5.2.3 Logic optimization 111
5.2.4 Property-checking for optimized model 112

5.3 Illustration by example: Equivalence function 112
5.3.1 Module description 112
5.3.2 Significance extension and verification 112
5.4 Prototype implementation 115
5.5 Experimental results 116
5.6 Conclusions 118

6 Conclusion et perspectives 119

A 123
A.1 Extended boolean operators 123
A.2 Constraint system for MUX 125
A.3 Constraint system for AND 127

B 129
B.1 VHDL descriptions for Serial parallel multiplier (SPM) 129

List of Figures

2.1 Example: Accumulator module 24
2.2 A program extract 26
2.3 Half adder implementations 27
2.4 D flip-flop with Enable 29
2.5 VHDL description ($DFFE.RTL2$) 34
2.6 Two traces showing D as a control input 42
2.7 Two traces not showing D as a control input 42

3.1 Example of slicing 46
3.2 An example of Control Flow Graph (CFG) 47
3.3 Control/Data separation in IP modules by slicing 48
3.4 Data dependent separation of control and data 49
3.5 Accumulator example 55
3.6 Control and data flows in accumulator 56
3.7 CFG of control and data slices for accumulator 56
3.8 Slicing results of accumulator 57
3.9 Accumulator slices in VHDL 57
3.10 VHDL RTL description of accumulator with variable 59
3.11 Accumulator architecture declarative part after transformation 62
3.12 Accumulator architecture body after transformation 62
3.13 Combinational process in control slice after enhancement 64
3.14 Combinational process in data slice after enhancement 65
3.15 Accumulator: entities of slices after enhancement 65
3.16 Slicing result of accumulator with variable 65
3.17 Control-data slicing with local variables 66
3.18 Assisting model checking by slicing and monitors 69
3.19 Data processing phases in data slice 70
3.20 Functional data slice template 72
3.21 SPM after slicing 73
3.22 SPM after data abstraction 74

4.1 D flip-flop with Enable 78
4.2 Module M 81
4.3 $if - else$ structure in VHDL RTL 92
4.4 Syntax transformation required at RTL 93

5.1 A generic module under verification 100

5.2 A conceptual description of verification approach . . . 101
5.3 Processing steps during verification . . . 103
5.4 Module foo: Circuit diagram, original and sliced RTL descriptions . . . 105
5.5 Gate-level description for module foo . . . 106
5.6 Gate-level description for Data slice of module foo . . . 106
5.7 Generic significance extended model . . . 108
5.8 Generic property monitor . . . 108
5.9 Module $\mathcal{M}$ under verification . . . 109
5.10 Monitor: property 5.1 for model $\mathcal{M}'$ . . . 111
5.11 Equivalence function model: $\mathcal{M}_e$ . . . 113
5.12 Significance extended module $\mathcal{M}'_e$ of Equivalence function . . . 113
5.13 Monitor for equivalence function with property 5.1 . . . 114
5.14 Optimized model of equivalence function for property 5.1 . . . 115

B.1 Original SPM implementation (before slicing) . . . 129
B.2 VHDL process containing data processing of SPM (regenerated after slicing) . . . 130
B.3 VHDL Subprogram declaration to interface with foreign function . . . 130
B.4 Functional data computation in data slice . . . 131
B.5 Fast implementation of SPM . . . 131

List of Tables

1 Frequent notations used in the thesis 14

2.1 Two simulation experiments for DFFE with E as test input 32
2.2 Two simulation experiments for DFFE with D as test input 32
2.3 Truth table of $((D \wedge E) \vee (Q \wedge (\neg E)))_s = S_s$ 36
2.4 Control/Data propagation 39
2.5 Control and data separation for $DFFE.RTL$ with designer's intervention . . . 40
2.6 Control and data separation for $DFFE.RTL2$ with designer's intervention . . . 41

3.1 Model checking results with and without slicing 68
3.2 Simulation Results 74

4.1 Truth tables for data signals (Static notion) 85
4.2 Truth tables for control and data signals 85
4.3 Truth tables for data signals (Dynamic notion) 86
4.4 Mapping table for value and significance 86
4.5 Mapped truth tables for $\wedge$ operation (static notion) 87
4.6 Mapped truth tables for $\vee$ operation 88
4.7 Mapped truth tables for $\neg$ operation 89
4.8 Mapped truth tables for $\wedge$ operation (dynamic notion) 89
4.9 Mapped truth tables for $\vee$ operation (dynamic notion) 90
4.10 Mapped truth tables for $\wedge$ operation (Control/Data) 91
4.11 Mapped truth tables for $\vee$ operation (Control/Data) 91
4.12 Semantics of *equality test* 95

5.1 Model checking results with static semantics 117
5.2 Model checking results with dynamic semantics 117

A.1 MUX valuations 124

Notations	**Description**	**Defined in**	**Chapter No.**
$\mathtt{F}$, $\mathtt{T}$	False, True significant booleans	section 2.3, 4.4	2, 4
$\mathtt{f}$, $\mathtt{t}$	False, True non-significant booleans	section 2.3, 4.4	2, 4
$\wedge$, $\vee$, $\neg$ and $\oplus$	AND, OR, NOT and XOR boolean functions	–	2, 3, 4
A_s	The test 'A is significant'	section 2.4	2
A_n	The test 'A is non-significant'	section 2.4	2
$A_{n \to s}$	Non-significant to significant change of A	section 2.4	2
$A_{s \to n}$	Significant to non-significant change of A	section 2.4	2
M	A synchronous module	definition 2.1	2, 4
R	Set of registers	section 4.4	4
I	Set of inputs	definition 2.1	2, 4
O	Set of outputs	definition 2.1	2, 4
$\mathcal{E}$, $\mathcal{A}$, $\mathcal{P}$	VHDL entity, architecture, processes	definition 3.1	3
$\hat{S}$	Set of VHDL signals	definition 3.1	3
$\mathscr{V}$	Set of variables in VHDL process	definition 3.1	3
$\mathcal{L}$	Sensitivity list of VHDL process	definition 3.1	3
$\mathscr{S}$	Sequence of statements	definition 3.1	3
REG_P	Registers belonging to process P	section 3.5.1	3
$\mathcal{W}_P$	Signals written in process P	section 3.5.1	3
$\mathcal{R}_P$	Signals read in process P	section 3.5.1	3
Φ	Set of boolean formulas	section 4.3	4
φ	A boolean formula	section 4.3	4
t, $t+1$	Rising edges of clock (elements of $\mathbb{N}^*$)	section 4.3	4
$\mathbf{c}$	Combinational computation	section 4.3	4
$sig(a)$	Significance of boolean variable 'a'	section 4.4	4
$val(a)$	Value of boolean variable 'a'	section 4.4	4
$\mathcal{V}$	Value function	section 4.4	4
$\mathcal{S}$	Significance function	section 4.4	4
$\mathcal{S}_{stat}$	Static significance	section 4.3	4
$\mathcal{S}_{dyn}$	Dynamic significance	section 4.3	4
$\mathcal{S}_{ref}$	Refined significance	definition 4.2	4

Table 1: Frequent notations used in the thesis

Résumé

De nos jours la conception des IP (IP: Intellectual Property) peut bénéficier de nouvelles techniques de vérification symbolique: abstraction de donnée et analyse statique formelle. Nous pensons qu'il est nécessaire de séparer clairement le *Contrôle* des *Données* avant toute vérification automatique.

Nous avons proposé une définition du «contrôle» qui repose sur l'idée intuitive qu'il a un impact sur le *séquencement* de données. Autour de cette idée, le travail a consisté à s'appuyer sur la sémantique des opérateurs booléens et proposer une extension qui exprime cette notion de *séquencement*. Ceci nous a mené à la conclusion que la séparation parfaite du contrôle et des données est illusoire car les calculs dépendent trop de la représentation syntaxique. Pour atteindre notre objectif, nous nous sommes alors basés sur la connaissance fournie par le concepteur: séparation *a priori* des entrées contrôle et des entrées données. De cela, nous avons proposé un algorithme de «slicing» pour partitionner le modèle. Une abstraction fut alors obtenue dans le cas où le contrôle est bien indépendant des données. Pour accélérer les simulations, nous avons remplacé le traitement de données, défini au niveau bit par un modèle d'exécution fonctionnel, tout en gardant inchangé la partie contrôle. Ce modèle intègre des aspects temporels qui permet de se greffer sur des outils de *model checking*. Nous introduisons la notion de *significativité* support des données intentionnelles dans les modèles IP. La significativité est utilisée pour représenter des dépendances de données booléennes en vue de vérifier formellement et statiquement les flots de données. Nous proposons plusieurs approximations qui mettent en œuvre cette nouvelle notion.

Abstract

Hardware verification has become challenging due to ever-growing complexity of today's designs. We aim at assisting verification of hardware intellectual property (IP) modules at register transfer level (RTL) by means of data abstraction and static formal analysis techniques. We believe that before applying data abstraction, it is necessary to clearly define and separate the *Control* and *Data processing* of modules.

The consideration of control and data in hardware has previously been a subjective judgment of the designer, based on the syntax. We intuitively define the "Control" as an entity responsible for the *timings* of the data operations in IP modules. The proposed definition was envisaged for separating Control and Data, independent of the subjective choice or the specific syntax. We have worked around a few semantic issues of the definition and demonstrated by reasoning, that an ideal separation of control and data is not achievable according to the proposed definition due to the syntax dependent boolean computations. We therefore, separate the Control and Data based on designer's knowledge. A control-data slicing algorithm is proposed to split the module into a *control slice* and a *data slice*.

An abstraction is achieved in case of slicing with data-independent control. The bit accurate RTL data slice is replaced by a functional data computation model for fast simulations. The control slice being critical entity with timing information, remains intact during this process. This provides us a way of abstracting the data processing and considering only the timing information for formal verification. We have proposed the notion of *significance* to represent the intentional data in IP modules. Significance is used to represent boolean data dependencies in modules for formal verification of the data flows. Approximations to data dependencies in IP modules have been realized with demonstration of their correctness. The verification technique based on significance is realized which enables to formally verify properties related to the datapaths.

Chapter 1

Introduction

De nos jours, les systèmes numériques sont de plus en plus tellement complexes, qu'une conception directe est devenue difficile. Leur mise en œuvre fait aujourd'hui appel à l'interconnexion de plusieurs composants (IP: Intellectual Property) disponibles sur le marché, validés et developpés par ailleurs. Vu le nombre important de fournisseurs d'IPs de base, et le manque de standards, les services rendus deviennent de plus en plus hétérogènes, ce qui augmente la complexité d'une vérification globale.

Dans un cycle de conception normal, environ 70 % du temps de conception est dédié à la phase de test et de validation. La conception revêt différents aspects: tels que le délai de mise sur le marché (time-to-market) et les marges financières qui dépendent largement de la phase de vérification. En conséquence, les erreurs de conception non identifiées durant le cycle, peuvent engendrer des impacts financiers significatifs en augmentant le time-to-market et en réduisant la marge de gain.

Pour optimiser les profits, les techniques de vérification doivent être aussi efficaces que possibles. De plus un bogue de conception peut induire des conséquences graves pour le matériel voire funestes pour des utilisateurs. Habituellement les techniques de vérification et de validation formelle souffrent pour les gros systèmes, d'un temps de calcul long, d'une couverture de test limitée et d'une explosion combinatoire. Travailler à un niveau d'abstraction élevé est recommandé par la communauté scientifique pour faciliter les techniques de vérification. En fait, un haut niveau d'abstraction des IPs permet de gérer la complexité des systèmes qui comportent aujourd'hui plusieurs millions de portes logiques.

Dans cette thèse, nous proposons des techniques basées sur *l'abstraction de données* et *l'analyse statique formelle* pour aider le processus de vérification par simulation et model-checking. Le principal objectif de ce travail est d'une part d'identifier le flot de données dans une IP déjà définie et d'autre part d'appliquer les techniques d'abstraction de données en vue d'obtenir un modèle simplifié.

Ces objectifs et mes motivations sont décrits dans le paragraphe 1.2. Mais en premier lieu, il me semble important de présenter un rapide état de l'art relatif aux techniques de vérification afin de mieux esquisser les idées présentées dans la suite de ce manuscrit.

1.1 État de l'art

Les techniques de vérification pour les systèmes numériques industriels sont principalement divisées en deux catégories : *simulation* et *vérification formelle* comme décrits ci-dessous.

1.1.1 Vérification par simulation

La simulation est une exécution dynamique d'un modèle pour tester sa fiablilité. Pendant la simulation, un modèle bas niveau est injecté avec les séquences de données appelées les «vecteurs de test». Les sorties associées sont *observées*, pour s'assurer que les comportements du modèle sont conformes aux spécifications. Les vecteurs de test sont produits, soit automatiquement par génération de vecteurs aléatoires[33], soit par un outil de résolution de contraintes [60], soit manuellement par les utilisateurs. Il est important que les vecteurs de test caracterisent des séquences particulières introduisant des bogues. Comme il est impossible de parcourir tous les vecteurs de test possibles imaginable, un grand nombre de bogues potentiels restent indetectés. Pour palier ce phénomène, les techniques de génération automatique de test (ATPG: Automatic Test Pattern Generation) [64], [67], [44] et les méthodes de conception pour le test [48], [105] sont fréquemment utilisées pour assister la vérification basée sur la simulation en augmentant la couverture de test.

1.1.2 Vérification par model checking

Les simulations ne garantissant pas la correction fonctionnelle globale, une vérification formelle prend toute son importance: depuis quelques années, les avancées théoriques dans les méthodes formelles [45] se sont accompagnées d'outils logiciels, qui dépassent l'approche traditionnelle par simulation dans sa capacité d'identifier les erreurs dans les systèmes numériques critiques. La vérification formelle est une approche algorithmique et logique. L'objectif est de diagnostiquer statiquement et mathématiquement les erreurs. Dans l'industrie, vérification formelle rime souvent avec «*equivalence-checking*» [93] ou «*model-checking*» [31].

Les techniques d'equivalence-checking comparent formellement deux spécifications au niveau RTL (ou portes). Le model-checking est une approche pour vérifier la correction de la description fonctionnelle selon la spécification. Le model-checking comprend trois parties:

1. Un «cadre de travail» pour *modéliser les systèmes*, typiquement c'est un langage de description orienté,

2. Un langage de *spécification* pour décrire les propriétés à vérifier,

3. Un *algorithme de model-checking* pour déterminer si la description du système satisfait les spécifications.

Le model-checking permet à l'utilisateur de détecter les propriétés de sûreté et de vivacité d'un système tel que la situation d'impasse (puit) où une exclusion mutuelle induisant une situation de blocage du système (étreinte fatale). Le model-checking s'applique aux *systèmes de transition d'états* tels que les circuits séquentiels ou les protocoles de communication.

Les spécifications de propriétés décrivent *ce que* le système doit faire et ce qu'il ne doit pas faire, alors que le modèle décrit *comment* le système se comporte. Les propriétés à vérifier sont exprimées dans les langages de logique temporelles comme la logique temporelle linéaire (LTL: Linear Temporal Logic), la logique arborescente (CTL: Computation Tree Logic) [25], le langage de spécification de propriétés [26] (PSL: Property Specification Language) [59] ou les autres formalismes comme les automates de Büchi [40]. L'algorithme de model-checking traverse les états accessibles du modèle pour vérifier si les spécifications données sont satisfaites ou sont violées. En cas de violation de propriété, l'algorithme produit un contre-exemple sous forme de trace de simulation pour le débogage.

Dans l'état de l'art, les outils de vérification basés sur le model-checking, ne peuvent pas encore remplacer la méthode de vérification par simulation pour les raisons suivantes: premièrement, les algorithmes de model-checking ne passent pas à l'échelle à cause du problème de complexité (explosion combinatoire du nombre d'états [79]). En conséquence, les outils de vérification sont toujours limités aux modèles simples. Deuxièmement, ces outils exigent toujours une connaissance experte pour décrire les spécifications. Par ailleurs, les utilisateurs doivent investir un temps assez long pour modéliser l'environnement qui permetttra de valider toutes les propriétés de spécifications et ne pas en oublier!

Les améliorations du processus de vérification ont toujours été indispensables pour fournir une productivité élevée dans les systèmes numériques. Les techniques d'abstraction récentes, fournissant des améliorations dans la simulation et la vérification formelle, sont mentionnées dans le paragraphe suivant.

1.1.3 Les techniques d'abstraction

L'abstraction est un mécanisme de projection: parmi tous les comportements du système, seuls les comportements spécifiquement liés à la propriété en question, sont analysés. De nombreuses techniques d'abstraction existent aujourd'hui, nous mentionons dans cette thèse quelques contributions pertinentes. Pour les méthodes de conception de matériel, les techniques d'abstraction sont utilisées pour contrecarrer la complexité inhérente des systèmes actuels. Citons pour exemple la modélisation au niveau système en SystemC [1], OCAPI [95], metaRTL [109].

De plus, les industries s'efforcent, grâce à la modélisation au niveau transactionnel (TLM: Transaction Level Models), de simplifier leurs flots de conception ainsi que la vérification associée. S. Swan [98] décrit comment les industries adoptent les modèles TLM pour la conception et la vérification de SoC. De même quelques techniques ont été présentées pour jeter une passerelle entre la vérification basée sur RTL et les techniques de modélisation de haut niveau. F. Fummi et al. [14] ont présenté une abstraction de RTL vers les modèles TLM pour améliorer la simulation au niveau système.

En vérification formelle, les chercheurs ont proposé des techniques d'abstraction [66], [73], [61] en vue de modéliser les systèmes numériques. Ils ont ainsi réussi à améliorer la couverture des systèmes complexes. En général, les ingénieurs de validation utilisent des techniques d'abstraction de données pour réduire l'espace d'état. Ils regroupent au sein d'un même état abstrait (et invariant sur la propriété voulue), plusieurs états réels du système. Ainsi le système abstrait obtenu, conserve le comportement d'origine. Des progrès significatifs ont été faits par E.M. Clarke et al. sur les techniques d'abstraction concernant le model-checking [30], [12], [29].

L'interprétation abstraite [36], [37] est une technique d'analyse statique pour obtenir une approximation correcte de la sémantique d'un programme, basée sur les fonctions monotones des treillis: C. Hymans a traité l'interprétation abstraite d'un sous-ensemble de VHDL comportemental [54], [55]. De ses travaux découlent une analyse statique d'un sous-ensemble de VHDL indispensable pour calculer une approximation des états accessibles pendant une simulation VHDL classique. Il est montré que le parallélisme de VHDL est très faible et qu'il est inutile d'explorer tous les entrelacements possibles des processus pendant la simulation. Il est suffisant de fixer un ordre d'exécution statique des processus une fois pour toute. De cette manière, un simulateur abstrait est ensuite généré à partir d'une sémantique formelle de simulation VHDL. L'auteur a d'ailleurs donné une preuve de correction de son algorithme.

Y. Hsieh et al. [53], [52] proposent une représentation abstraite d'un modèle VHDL comportemental sous forme de machines d'états finis non-déterministes (NFSM: Nondeterministic Finite State Machine). Ils réduisent ainsi le nombre d'états et facilitent le model-checking. Le thème principal de ce travail est l'abstraction des compteurs [77]. Cette approche est bien-sûr adaptée dès que le comportement de l'application est fortement dépendante de compteurs internes. Des classes de NFSM prédéfinis existent: elles réduisent considérablement l'espace d'état.

Cependant ces méthodes sont fortement dépendantes des constructeurs syntaxiques utilisés par le concepteur et donc de son savoir-faire alors que nous cherchons une approche qui évite justement l'intervention humaine le plus possible. Le chapitre 2 souligne justement ces problèmes d'indépendance de syntaxe.

Abstraction de données

Les systèmes flot de données sont paradoxalement sensibles à l'explosion d'états à cause du grand nombre de registres internes. Des techniques d'abstraction de chemin de données ont été proposées dans la littérature pour s'attaquer à ce problème.

E. Macci et al. [76] ont proposés une sémantique pour réduire les gros chemins de données en une machine d'état finis non-déterministes à 4 états. Ceci permet d'enlever la redondance des actions sur ce chemin de données et de simplifier d'autant le contrôleur. Cependant la technique est basée sur certaines hypothèses: une distinction préalable entre contrôle et données doit être supposée. Le nombre de cycles d'horloge nécessaires aux opérations arithmétiques doit aussi être connu.

Parallèlement à ces travaux, une méthode spatiale d'abstraction a été présentée par V. paruthi et al. [90] pour abstraire des parties du chemin de données d'un microprocesseur. L'abstraction spatiale réduit la complexité en taille des opérateurs aritmétiques, laissant le contrôleur intact. Cette méthode utilise des techniques classiques de calcul d'intervalle [47] pour déterminer l'espace de définition. L'idée fondamentale sousjacente, est d'identifier en premier les éléments de stockage dans le chemin de données qui ne contribuent pas au flot du contrôle, puis de réduire leur taille à un «bit». Pour chaque élément de stockage sont ensuite identifiées les valeurs min et max par propagation du calcul d'intervalle, à travers la description syntaxique du modèle.

De même, R. Hojati et al. ont contribués de leur côté à abstraire les données en utilisant la notion de fonctions non interpretées [21], [9] toujours pour démontrer des propriétés liées aux chemins de données.

La plupart des techniques mentionnées dans les paragraphes ci-dessus sont adaptées à des domaines spécifiques d'applications. Certains sont fondés sur des hypothèses et exigent des interventions manuelles. La complexité croissante des systèmes numériques promet toujours de nouveaux défis en vérification, rendant les techniques existantes rapidement inefficaces et obsolètes! Notre contribution est une tentative d'enrichir les méthodes et les techniques d'abstraction existantes.

1.2 Principaux objectifs de cette thèse

Classiquement les systèmes numériques sont composés d'une partie *traitement de données* et d'une partie *contrôle*. Intuitivement, le traitement de données comporte des calculs assez complexes sur un grand nombre des registres. Le contrôle supporte tout le séquençage nécessaire

aux opérations diverses de traitement de donnée indispensable aux IP. Nous pensons qu'une bonne abstraction de données sans perte d'informations temporelles critiques nécessite une identification et une séparation du traitement des données.

Notre premier objectif dans cette thèse est justement de partitionner le plus automatiquement possible les composants en deux entités indépendantes (données/contrôle) mais communicantes. La structure d'un composant sous cette forme est aussi appelé «machine d'états finie avec chemin de donnée» : FSMD (Finite State machine with Datapath). Le FSMD [41] est un modèle qui facilite non seulement le test et la verification systematique des composants mais encore il est utile dans la synthèse de haut niveau.

1.2.1 Motivation

Dans le cadre de la séparation contrôle/données, la vérification et l'abstraction ont été motivé par les objectifs suivants:

- Pour identifier les séquences de contrôle des IP, réduire le contrôle en un modèle plus simple facilitant l'analyse statique et les vérifications formelles des propriétés temporelles critiques.
- Transformer le modèle bas niveau: RTL (Register Transfer Level) ou portes logiques du traitement des données, en un modèle plus abstrait.
- Analyser statiquement le flot de donnée en vue de vérifier formellement le traitement associé.
- Accroitre la connaissance à posteriori d'un composant IP externe (reverse engineering).

Notre deuxième objectif dans cette thèse est de fournir aux ingénieurs de validation un "framework" intégrant les techniques de vérification formelle dans le cycle de validation. Grâce á ce framework, les ingénieurs seront capable de vérifier les propriétés intéressantes du flot de donnée. La technique de vérification utilisée est basée sur létude mathématiques des dépendence de données booléennes dans les systèmes numériques et s'intéresse aussi à la séparation du contrôle et des données.

1.2.2 Cadre de travail

Nous allons traiter des modèles qui rendent un service spécifique et qui peuvent être utilisés en tant que sous-système dans les circuits numériques complexes. Nous considèrerons ainsi: les cœurs IP dans un système sur puce (SoC: System-on-Chip), les blocs fonctionnels dédiés dans un ASIC[1] et enfin les microprocesseurs. Le cadre des travaux présentés est limité aux *modèles évoluant sur une horloge unique (évolutions synchrones)* qui sont largement utilisés dans ces systèmes. Dans les circuits synchrones, tous les registres (partie séquentielle) sont synchronisés par cette horloge.

De manière classique les modèles synchrones contiennent de la logique combinatoire et des registres. En langages de descriptions matérielles (HDL: Hardware Description Languages) dont VHDL et Verilog, les utilisateurs peuvent décrire leurs applications suivant deux niveaux d'abstraction différents. Dans le niveau le général, ils utilisent les constructeurs de haut

[1] ASIC: application specific integrated circuits

niveau du langage tels que: les conditions, les boucles et les types de données ainsi que leurs opérateurs associés. Dans un niveau plus bas mais aussi plus près de la machine, ils décrivent directement la partie combinatoire et la partie séquentielle (niveau RTL: Register Transfert Level).

Nous avons choisi VHDL comme langage de modélisation dans cette thèse vu son importance dans l'industrie. VHDL a été préféré aux autres, par sa puissance de description, pour les types enumérés et la possiblité de surcharger les opérateurs. Nous aurions pu également choisir Verilog ou SystemC et suivre la même démarche méthodologique.

1.2.3 Structure et organisation de la thèse

Cette thèse se propose d'accompagner les techniques de vérification actuelles en facilitant leur travail d'analyse et en réduisant la complexité des systèmes étudiés. Cette contribution s'exprimera sous forme de deux parties : la séparation de contrôle et des données dans les modèles et la vérification statique formelle de flot de données.

Nous nous intéresserons en premier lieu dans les chapitres 2 et 3 au problème de séparation contrôle / données dans les systèmes numériques avec une approche différente des solutions classiques. La définition intuitive du contrôle avec les problèmes et les limitations sémantiques apparentées sera d'ailleurs présentée en détail.

Nous verrons aussi comment implementer la séparation contrôle/données basée sur les techniques de «slicing». La séparation obtenue par slicing pourra être exploitée pour abstraire le traitement de données en vue d'aider les outils de vérification et de simulations. Quelques problèmes syntaxiques seront d'ailleurs abordés à propos du processus de séparation et nous proposerons des solutions. Les prémisses de ces travaux ont été présentés dans [85], [84].

Pour la seconde partie de la thèse, (chapitre 4 et 5) portant sur validation statique formelle de flot de données, nous introduirons une nouvelle notion de *significativité* représentant l'idée que le concepteur a de la validité d'une donnée à un instant t dans les modèles IP. Selon cette notion et pendant une opération particulière sur le chemin de données, les entrées *significatives* seront utilisées pour calculer les sorties significatives et ainsi propager la significativité au sein du système. Les règles de propagation s'appuiront sur des modèles booléens classiques et permettront ainsi de s'intégrer aux outils de synthèse et de simulation classique. Ils offriront aux concepteurs des services de vérification formels de leurs systèmes. Dans ce chapitre sera aussi décrite une méthode de vérification: cette méthode nous permettra d'identifier de nouvelles directions intéressantes de recherche.

Chapter 2

Separating Control and Data in hardware modules

2.1 Introduction

In this chapter, we will describe the concept of separating *control state machines* and *data processing* in register transfer level (RTL) hardware modules. We believe that the first step towards assisting verification and abstraction is to separate the 'Control' and 'Data', so that two distinct behaviors of modules could be investigated in different ways. Our initial goal was to automatically extract control state machine in a hardware module without prior knowledge about its functionality, description style or any design hint.

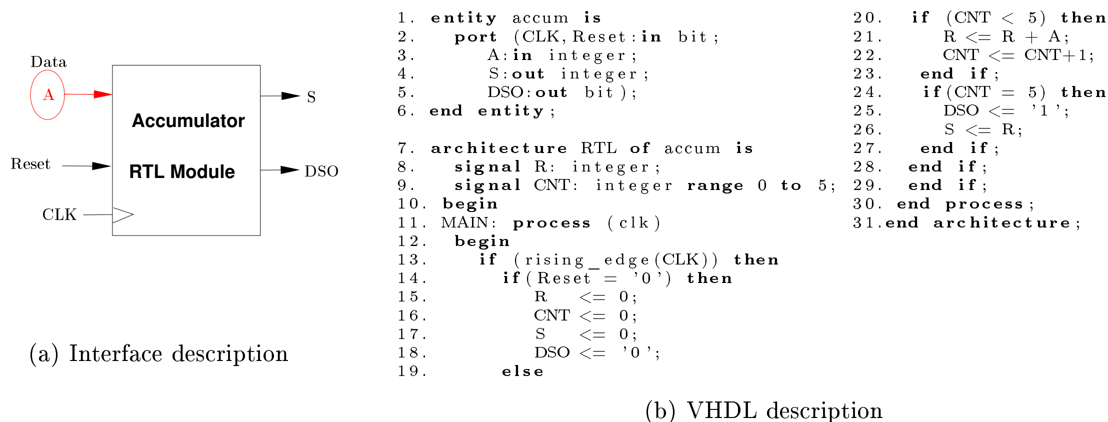
For this purpose, we first proposed an intuitive definition of *Control* and tried to develop some semantic analysis. According to the definition "*the control inputs and the related hardware determine the timing of the outputs of the module*". Based on this notion, we tried to automatically identify the control inputs of an unknown module, first without designer's interventions, and then with the help of designer's intervention in an analytical way. After working around various semantic aspects of the proposed definition, we reached a valuable conclusion that a unique separation of the control and data, independent of syntax, is not achievable according to the proposed definition.

We will first describe the fundamental concepts and the problems related to control and data separation. In the next section, we will present the control and data processing in software/hardware and highlight our envisaged notion of separating control and data compared to the state of the art techniques. In section 2.3, the intuitive definition of control is presented in a semi-formal way with illustration. In section 2.4, we try to identify control inputs automatically without designer's interventions and independent of description style. We describe an analysis based on intuitive semantics, and portray the limitations of the definition by reasoning on basic hardware blocks. Section 2.5 introduces the designer's interventions to find the desired solution. A critical discussion on the proposed definition is given in section 2.6, and finally we conclude the chapter by mentioning the impact of this research contribution on the state of the art and rest of the thesis.

2.1.1 Basic idea

Traditionally, hardware modules are designed with a notion of datapath and control unit. For a specific design, sometimes the designers seem to have a clear idea of the *data* and *control*. Some designers maintain a distinction of controller and datapath in their designs. Usually they design the controllers as finite state machines, and the datapaths as pipelines of registers with combinational operations. The datapath and controllers interact with each others. Such designs are also called finite state machines with datapath (FSMD) [41], [15].

FSMD design approach is used most of the times by the designers. However, many designers do not keep the control/data distinction during a design process, particularly while designing small functional blocks. The control and data processing are mixed in the final design delivered to the verification and synthesis team. As an example, consider the VHDL description of an accumulator circuit shown in Figure 2.1. It accumulates 5 samples at its input '`A`' in 5 clock cycles and produces the output result at '`S`' after 6 clock cycles in the absence of '`Reset`' signal. Intuitively, we see that the circuit consists in two parts. One part consisting in *data processing* and involves data input '`A`' and output '`S`' in operations. The other part consisting in a '`Reset`' input, and an internal logic driving the output signal '`DSO`' to indicate the presence of valid data at '`S`'. The timing logic is collectively called the "*Control*".



(a) Interface description

(b) VHDL description

Figure 2.1: Example: Accumulator module

We believe that the control or data oriented behavior of the hardware modules is determined by the input and output interfaces. In hardware modules such as accumulator of Figure 2.1, we observe that some inputs are control oriented and some inputs are data oriented. However, there exist modules in which some inputs are used for *Control* as well as *Data*. For instance in case of a UART receiver, a serial data input also contains the *start* and the *stop* bits before and after the 8-bit data stream, which contain the timing information of the data.

We are interested in separating the inputs and corresponding logic contributing to the *timings* of a hardware module as the "Control", based on the behavior of the inputs and output waveforms without considering the internal implementation styles and syntax of the descriptions.

2.1.2 Problematics

The choice of *Control* and *Data* inputs in hardware modules is usually a subjective and contextual judgment by the designer. Most of the times, this choice depends on the functionality of the module and the environment where module is used. For instance, in case of multiplexers, the selector inputs are usually considered as control inputs because their combination controls the flow of various incoming data inputs towards outputs. In case of a digital filter, the inputs carrying samples might be considered as data inputs whereas rest of the inputs might be control or timing signals. Similarly, in almost all hardware designs, a *reset* input is used to clear the internal registers of the design which is also considered intuitively as control input by the designers. Thus the way in which inputs are called as Control or Data, depends on the context of the application.

In various tasks during a computer aided design (CAD) flow such as verification, synthesis and physical design, it is sometimes required to separate the controller and datapath to study various aspects of the design and apply different techniques. To study data abstraction techniques and verification of the related properties, we also emphasized on separating the 'Control' from 'Data processing' at first place. However, the existing techniques for such separation are based on syntactic analysis. We have raised the question of defining and separating *Control* and *Data* in hardware modules, independent of the syntax.

Envisaged goal

We aim at addressing following issues in this chapter:

- Propose a definition of 'control' and/or 'data' to obtain an objective characterization of the two distinct but related concepts instead of their subjective judgment,
- Based on above definition, propose a control and data separation solution independent of syntactic representations of hardware modules.

To our knowledge, there is no research on precise formal definition of *control* and *data* on the basis of which, we can make a distinction between them. Therefore, we have tried to give an intuitive description of the *Control* and *Data* and their automatic separation on semantic basis rather than considering a specific syntax.

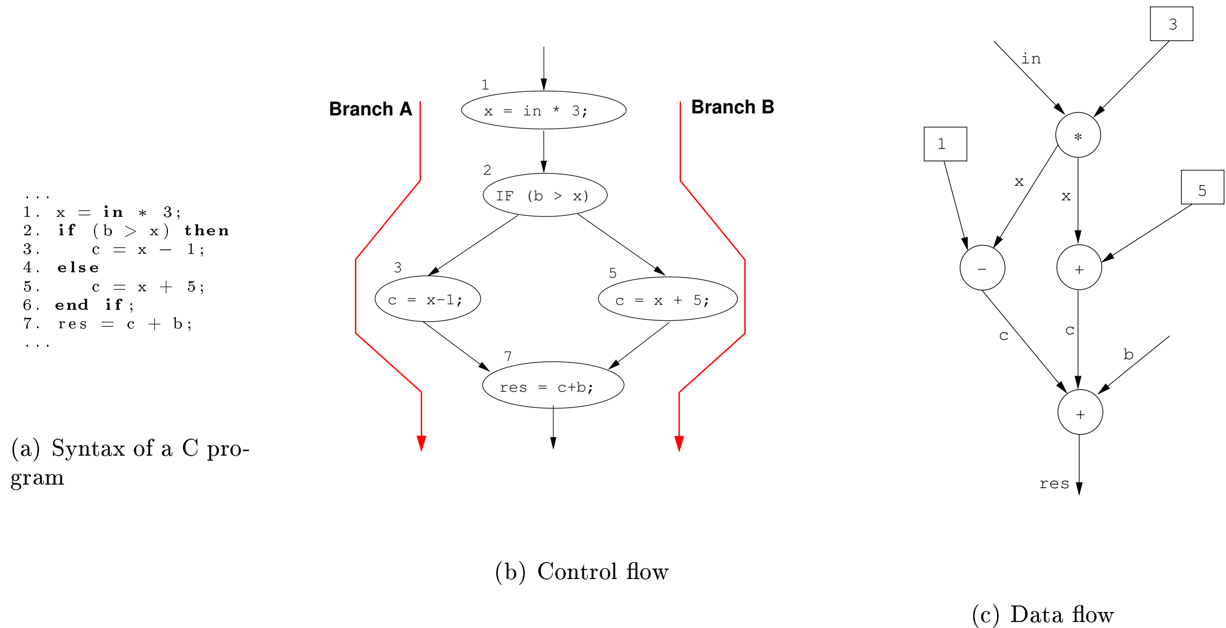
2.2 State of the art

We will briefly describe the classical concept of control and data flow in software and hardware designs and discuss some existing techniques in the literature about separation of control and data.

2.2.1 Control and data flows in software

The literature about the *control flow* and *data flow* is vast e.g. [83], [100], [51], [38], [61]. The terms are basically evolved from compiler design theory [8]. Control flow in programming languages represents the sequence of execution of a program. The execution of sequential instructions in a program one after an other, splitting the execution flows in two branches on conditional statements, and the repetition of execution of a subprogram by loop statements, collectively constitute the basic control flow in a program. Data flows in a program represent,

how the values are operated and exchanged among variables in a given execution (control flow) of a program. The concept of control and data flows is illustrated for an extract of a C program shown in Figure 2.2(a).



(a) Syntax of a C program

(b) Control flow

(c) Data flow

Figure 2.2: A program extract

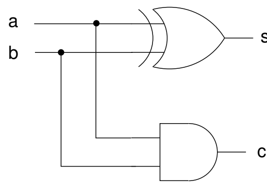
Depending upon the condition `b > x` at line 2, the two possible sequences of execution of the programs are either $1 \rightarrow 2 \rightarrow 3 \rightarrow 7$ or $1 \rightarrow 2 \rightarrow 5 \rightarrow 7$. An execution graph of this program is depicted in Figure 2.2(b), which is called the control flow graph (CFG). The two branches of execution are named as branch A and branch B. The flow of data among program variables during execution is also shown in Figure 2.2(c), which is called the data flow graph (DFG).

Programs are usually a mixture of control and data flows. To study some specific behavior related to either control or data flow, the internal representation of the program is transformed in terms of combinations of CFG and DFG [100].

2.2.2 Control and data flows in hardware

While talking about control and data flows in hardware, we usually imagine the *control state machines* and *datapaths*. Although in software programs or high level hardware programs (behavioral descriptions), conditional statements are supposed to represent the control, but in low level hardware (RTL or gate level descriptions), a conditional statement does not always represent control. For example consider a half adder circuit shown in Figure 2.3(a). With usual concept of control and data, most of designers will consider inputs `a` and `b` of the adder as *data* and would model it as shown by Syntax (I) in Figure 2.3(b). This represent a data flow from inputs '`a`' and '`b`' towards outputs '`s`' and '`c`'.

However, half adder can also be modeled by the `if-else` structure as shown in Figure 2.3(c) which is semantically equivalent to that of Figure 2.3(b). If we consider the pattern matching



(a) Half adder circuit

```
1. process(a,b) begin
2.   s <= a xor b;
3.   c <= a and b;
4. end process;
```

(b) Syntax (I)

```
1.  process(a,b) begin
2.   if(a /= b) then
3.       s <= '1';
4.   else
5.       s <= '0';
6.   end if;
7.   if (a = '0') then
8.       c <= '0';
9.   else
10.      c <= b;
11.  end if;
12.  end process;
```

(c) Syntax (II)

Figure 2.3: Half adder implementations

rule mentioned above, we see that inputs `a` and `b` being used in conditional statements at line 2 and 7 are control inputs in this description.

Thus we can not always say on the basis of syntax, that the module's behavior is control oriented or data oriented. For various applications, separation between control state machines and datapaths has been explored in past based on the syntactic analysis.

R. Namballa et al. [86] present a control and data flow graph (CDFG) extraction technique from VHDL behavioral descriptions aimed at facilitating high level synthesis. The presented notion of control and data in behavioral descriptions is based purely on VHDL syntax which is similar to the classical control and data flow in software as described in the preceding subsection.

The controllers in hardware, are typically implemented as finite state machines (FSM). Procedures to detect and extract such FSM are implemented in logic synthesizers for optimization purpose [7]. An algorithm is proposed by C. Liu et al. [72] to extract controller in a Verilog descriptions by detecting topological patterns representing general FSM structures. Such topologies are the combinational paths from inputs towards outputs with feedback paths through state registers. However, not all feedback paths belong to control FSM because a datapath can also be considered as a large finite state machine, which might contain topological paths similar to FSM. Therefore, C. Liu's algorithm does not guarantee whether the extracted FSM contains purely the control circuit. Pattern recognition techniques are used in which a control statement corresponds to a branch point in the data flow graph of the model which makes the analysis of [72] syntax dependent.

F. Fummi et al. [34] have proposed a source code modification of a VHDL descriptions to partition a design into the reference model composed of a controller driving a datapath. A VHDL description with a mix of datapath and controller is considered as input with *known data inputs* and *known data registers*. The algorithm transforms the model into an equivalent VHDL description which appears to be a finite state machine with datapath (FSMD). The methodology involves structural operations such as variable to signal conversion, register inference and isolation of registers from combinational logic based on syntax matching routines used in general purpose synthesis tools. Many operations such as separation of assignments and conditions are constrainted by user-specific thresholds which requires manual interventions.

The spatial abstraction presented by V. Paruthi et al. [90] consists of converting the design into a *flow graph*. During this step, internal variables are classified into *control variables* and *data variables*. It is assumed, that all variables that are appearing in loop and branch

statements are considered as control variables. In other words, no data operation can involve a branching or loop statements. This makes the abstraction paradigm syntax-dependent and reduces the application of the approach to a limited class of models with specific syntax.

J.A. Abraham et al. [50] have presented a control and data separation to improve quality of simulation tests. A prior knowledge of control and data registers is assumed before analysis. The data registers involved in control flow are assumed as primary inputs of the extracted control flow machine (ECFM) whereas the rest of data registers are dropped. The algorithm is based on VHDL syntax due to the fact that control flow statements are considered as program points constituting the control flow machines.

Remarks

Existing approaches of control and data separation as described above, translate source code into FSM by control data flow analysis techniques of general purpose compilers [8] which translate predefined language constructs into other forms. Therefore, compiler-based approaches must also limit user's coding styles to obtain a precise distinction between control and data. Although many techniques proposed in the literature claim automatic separation of control FSM in hardware description language (HDL) code. However, most of these tools depend on a specific coding style and require user interventions.

We aimed at proposing a Control and Data separation independent of syntactic representations. Our envisaged approach starts first by defining the conceptual behavior of the "Control" in hardware modules and based on this definition, we aimed at apply program analysis techniques to automatically separate the two different behaviors in hardware modules.

2.3 Conceptual behavior of the "Control"

We attempt to find a semantic notion of '*control*' in hardware modules instead of syntactic analysis and subjective choice of the designer. Consider the interface description of a synchronous module similar to one shown in Figure 2.1(a). Such module reads inputs and writes outputs at rising (or falling) edge of clock signal CLK. The control or data oriented behavior of this module depends on the nature of the inputs of the module. In order to be able to identify control inputs, we first define the behavior of the inputs contributing to the 'Control'.

2.3.1 Basic intuition

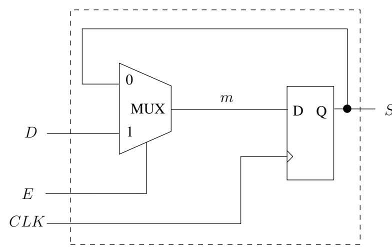
An input, output or internal wire (signal) in a synchronous module has two characteristics: the *value* and its *timing reference* with respect to clock signal. Intuitively, an input is said to be a *control* if its value change has an impact on the timing reference of the outputs, regardless of the correctness of the values of the outputs. An input is said to be a *data*, if it has an impact on the correctness of the outputs. Our notion of control and data separation deals with the inputs if the modules. The modules do not have combinational feedback paths from outputs towards inputs. Hence our first objective was to identify and separate control inputs in a standalone synchronous module.

One way to identify control inputs is to observe via simulation, whether a change in the value of an input, causes a change in the timings of the outputs. If an input signal has an impact on timings of the output waveforms, it can be identified as a control input. This observation could be achieved by extending the boolean character of hardware signals with an

additional attribute, which represents that a current signal value is either *significant* or *non-significant*. Intuitively, the ***significance*** attribute of a signal is the measure of its temporal behavior (timing).

Illustration of the Significance attribute

As described earlier, among the inputs of a model some are control oriented, some are data oriented and some are both. For the sake of simplicity we only consider that data input do not carry any control information. They only carry the data samples. A simple model that shows a control only and a data only input, is the enabled D flip-flop module ($DFFE.RTL$) shown in Figure 2.4.



(a) RTL diagram

```
entity DFFE is
  port(CLK, E, D: in bit;
        S: out bit);
end entity DFFE;

architecture RTL of DFFE is
  signal Q: bit;
begin
  process(CLK)
  begin
    if (CLK'event and CLK = '1') then
      if(E = '1') then
        Q <= D;
      end if;
    end if;
  end process;
  S <= Q;
end architecture RTL;
```

(b) VHDL description ($DFFE.RTL$)

Figure 2.4: D flip-flop with Enable

By intuition, CLK and E are control inputs while D is a data input. A way to verify this fact according to our notion of Control, is to imagine that the input signals are extended with a boolean attribute expressing their significance. During a simulation sequence, when an input signal is *significant* it means that the sender of this signal assigned a value to it on purpose. When an input signal is *non-significant*, it means that the sender had no usage of this signal at that particular time and the value of the signal is not relevant.

Values taken by a data input have no impact on the significance of the outputs. If the values were significant then the result they were contributing to, is probably also significant. Conversely, if they are non-significant then the result is also non-significant. But it's the only influence, the data inputs can have on the outputs significance. Control inputs, on the other hand, indicate whether a data input is relevant or not, that is, taken into account in the processing (as with the E input of the enabled D flip-flop example).

The value a control input taken at one particular time will decide whether a significant or non-significant data value will enter the computation. And this has an impact on the significance of the final output result. Assuming our enabled D flip-flop has significance extended inputs, one can see that a single value change of E can change the significance of S, while a single value change of D can not.

We will now relate the concept of significance with the behavior of hardware modules and propose a definition for the control inputs.

2.3.2 Definition of Control

We first build basic definitions of hardware modules and their behavior from interface viewpoint in a semi formal way to propose the definition of control inputs. We assume that modules considered in the analysis are synchronized on rising edge of a global clock signal and they have deterministic initial states.

We suppose that boolean values on inputs, outputs and internal signals in the hardware modules are extended with additional boolean attribute representing *significance*. Symbolically, we suppose that all signals are 4-valued, taking their values from the set {`F`, `T`, `f`, `t`}, such that, `F` and `T` represent *significant booleans*, whereas `f` and `t` represent *non-significant booleans*.

We define the state of the 4-valued signal at rising edge of clock as a **sample**. Let I be the set of 4-valued signals called *interface*. A **trace** is defind as a sequence of samples, intuitively corresponding to the sequence of values present on an interface at successive rising edges of the clock, starting from a deterministic initial state of the synchronous module. We define $\vec{T}(I)$ as the set of traces on an interface I.

We give a mathematical characterization of the synchronous module and its behavior in following definition.

Definition 2.1.
*Let I and O be two sets of signals. A **module** is a 3-tuple $M = (I, O, f)$, where*

- *I is **input interface** and O is **output interface***
- *$f : \vec{T}(I) \rightarrow \vec{T}(O)$ is a function called the **semantics** of the module: $\forall t \in \vec{T}(I)$, the length of $f(t)$ is equal to that of t*
- *A **simulation experiment**, labeled e on a module $M = (I, O, f)$, is a pair $e = (t, t')$, where $t \in \vec{T}(I)$, and $t' \in \vec{T}(O)$, such that $t' = f(t)$*

It is important to note that definition 2.1 describes only the external behavior of the module during a simulation with respect to input and output interfaces. This definition does not describe any formal interpretation of the internal behavior (semantics) of the module.

Based on the definition of module and its input and output behavior, we define control inputs as follows:

Definition 2.2 (Control input)**.**
*Given a module $M = (I, O, f)$ with input signal $i \in I$ of M, i is the **control input** of M, if and only if there exist two simulation experiments $e_1 = (t_1, t'_1)$ and $e_2 = (t_2, t'_2)$ such that:*

- *t_1 and t_2 are different and are different only in boolean values `F` and `T` associated to i*

and

- *t'_1 and t'_2 are different in significance*

Definition 2.2 intuitively says that

under a given context, the change in value of a* control input *causes a change in significance at the outputs.

Symbolically according to definition 2.2, the *change in value* is considered as a change between `F` and `T`. Whereas, a *change in significance* is either the change between `F` and `f` or the change between `T` and `t`. A change between `F` and `t`, or `T` and `f` comprises value, as well as significance change at the same time.

2.3.3 Illustration of the definition

We tested the impact of value change of control inputs on the output significance for some modules described in VHDL. For this purpose, we built a simulation environment in which we have extended the semantics of basic logical operators of VHDL in the form of truth tables with following assumptions:

- For a register, the time instant when output is significant at rising edge of clock, its input is significant at precedent rising edge of clock.
- All the constants inside the given module are assumed to be significant.
- All the internal registers are initialized with logic false significant value i.e. `F`.

From implementation viewpoint, one could also imagine that, the `std_ulogic` type in VHDL offers a don't care (`'-'`) value that could be used every time the value of a signal is not significant (irrelevant). However, there are some drawbacks making this impractical. One drawback is the differences between pre- and post-synthesis simulation results. Another is the `std_ulogic` truth table definitions stating, for instance, that `'-' and '-' = 'X'` (unknown) which is not exactly what we usually have in mind with significant/non-significant. So, most of the time, there is no way to decide whether the value carried by a signal is significant (relevant) or not. Thus an extension of VHDL signals and variables is considered.

The extended semantic rules are used to propagate the values and their significances from inputs towards outputs. The simulation environment is capable of supplying random sequences of inputs with value and significance. We iteratively provide inputs patterns as defined in definition 2.2 iteratively and observes the change in significance at the outputs.

As an illustrative example, we describe a simulation test on $DFFE.RTL$ of Figure 2.4.

Experimental testing

Considering the CLK as a global clock signal, the module $DFFE.RTL$ of Figure 2.4 has 2 inputs D and E, and one output S. To determine the control input, we have iteratively tested inputs D and E one by one, by changing value of one input while keeping the other input constant in value as well as significance during simulation experiments.

Table 2.1 shows, traces of the two experiments e_1 and e_2 with E as the *test input*. In both experiments, internal register Q is initialized with `F`. Therefore, at t_0, the output S is also `F`. At clock instant t_1, we have different value of E in both experiments as shown by blue symbolic values. In experiment e_1, we have the value `F`, whereas in experiment e_2 we have `T` at instant t_1. The variation of the context D (value as well as significance) must be same in both experiments e_1 and e_2. We can see that value change of test input E at clock instant

Experiments	CLK	t_0	t_1	t_2	t_3
e_1	**E**	F	**F**	F	F
	D	f	f	t	t
	S	F	F	**F**	**F**
e_2	**E**	F	**T**	F	F
	D	f	f	t	t
	S	F	F	**f**	**f**

Table 2.1: Two simulation experiments for DFFE with E as test input

t_1 causes a significance change at clock instant t_2 and later at t_3 as shown by red symbolic values. Thus E is a potential control input of the module.

Table 2.2 shows traces for two simulation experiments e_3 and e_4 with D as *test input*. In this case, we keep the input E as constant (the context for test input D), and only change the value of D at clock instant t_1 as shown by blue symbolic values. From these traces, we observe that there is no significance change on the output S at time instants t_2 and t_3 due to value change of D at instant t_1. The only change is the value of the output according to the value of D. This shows that, the value change of D does not impact the significance of the output S. The significance of S is changed only when the significance of D is also changed in this experiment.

Experiments	CLK	t_0	t_1	t_2	t_3
e_3	**E**	T	T	T	T
	D	F	**F**	F	F
	S	F	F	F	F
e_4	**E**	T	T	T	T
	D	F	**T**	F	F
	S	F	F	T	T

Table 2.2: Two simulation experiments for DFFE with D as test input

We have applied the definition 2.2 to gate level modules such as an 8-bit serial/parallel multiplier with `load` and `reset`, a register with `reset`, and a DCT input register stage with hand shaking control signals. The testing was only treated with small modules because the complexity of the number of tests exponentially depends on number of inputs. After a few number of tests on these module, we succeeded in identifying control inputs according to the proposed definition which are coherent with the intuitive and subjective selection of control inputs for these modules. For instance the value change of `reset` and `load` signals in 8-bit serial/parallel multiplier had an impact on the significance change of multiplication results. Therefore, they were detected as control inputs which is true according to intuition of the designer.

However in these experiments, we also observed over-approximations, where intuitive data inputs are detected as control and under-approximations where intuitive control inputs are detected as data. These spurious behaviors originate from the 'semantics' of the module which propagate value and significance from inputs towards outputs.

For control input identification, these semantics play a vital role. Therefore, it was necessary to search a suitable semantics which describes how the significance information propagates

through various combinational and sequential blocks in hardware module. A static analysis approach to find suitable semantics for automatic identification of right control inputs without any designer's intervention is described in following section.

2.4 Analysis without designer's intervention

We aimed at searching a suitable semantics of the module to propagate the significance and detect control inputs according to definition 2.2. For this purpose we overload the VHDL types and boolean operator semantics with significance and perform a static analysis on boolean equations and corresponding VHDL description. We use following notations for the demonstration.

- Conditions are expressions evaluating to `T` or `F`
- A in an expression means A has value `T`. In VHDL syntax, if A is of type `bit` then it represents the test $A =$ `'1'` .
- $\neg A$ in an expression means A has value `F`. In VHDL syntax, if A is of type `bit` then it represents the test $A =$ `'0'` .
- A_s denotes the test "A is significant", i.e. either $A =$ `'T'` or $A =$ `'F'`.
- A_n denotes the test "A is non significant", i.e. either $A =$ `'t'` or $A =$ `'f'`.
- $A_{n \to s}$ where A is a signal: a condition for A to go from non-significant to significant state, assuming A is in non-significant state.
- $A_{s \to n}$ where A is a signal: a condition for A to go from significant to non-significant state, assuming A is in significant state.

We will describe our static analysis on simple hardware entities. While solving the problem, we discovered that the multiplexer is a critical atomic hardware entity in our control and data separation analysis because it contains select lines as 'control' for selecting among different 'data' inputs. Therefore, we have worked each semantics with the multiplexer. A more interesting sequential module is the $DFFE.RTL$ circuit shown in Figure 2.4 which is composed of a 2-to-1 multiplexer and a D flip-flop with output connected back to the input.

For this analysis our main concern was the following *main rule of syntax independence*, which states that:

> ***The notion of Control and Data separation must be preserved across different but equivalent coding styles for the same functionality.***

Hence the VHDL module $DFFE.RTL$ of Figure 2.4 is equivalent to the $DFFE.RTL2$ description shown in Figure 2.5.

We propose some intuitive semantics to statically reason for control input identifications in modules $DFFE.RTL$ and $DFFE.RTL2$ as follows.

```
architecture RTL2 of DFFE is
  signal Q: bit;
begin
  process(CLK)
  begin
    if (CLK'event and CLK = '1') then
      Q <= (D and E) or (Q and (not E));
    end if;
  end process;
  S <= Q;
end architecture RTL2;
```

Figure 2.5: VHDL description ($DFFE.RTL2$)

2.4.1 Semantics $\mathcal{S}_0$

One of the simplest semantics for significance in VHDL is named as $\mathcal{S}_0$, composed of follows rules.

1. Constant literals are *significant*
2. Internal signals are initialized in *non-significant* state at elaboration time (before the simulation starts)
3. Boolean expressions are always *significant*. Equivalently the control structures like `if then elsif else` and `case` do not propagate significance to the nested statements
4. Expressions that evaluate as a `bit`, evaluate to *non-significant*, if at least one operand is *non-significant*, else to *significant*

In module $DFFE.RTL$ of Figure 2.4, since E is appeared as boolean expression, therefore according to rule 3, E is considered as always significant, however its value matters. With semantics $\mathcal{S}_0$, we can derive following equations for significance of the output Q.

$$\begin{aligned} Q_s &= ((E \wedge D_s) \vee (\neg E \wedge Q_s)) \ \wedge CLK \uparrow \\ Q_n &= ((E \wedge D_n) \vee (\neg E \wedge Q_n)) \ \wedge CLK \uparrow \end{aligned}$$

where $CLK \uparrow$ is a notation representing rising edge of signal CLK. If we try to build the conditions for change in significance of output signal Q for $DFFE.RTL$, we will obtain following:

$$\begin{aligned} Q_{n \to s} &= Q_n \wedge ((E \wedge D_s) \vee (\neg E \wedge Q_s)) \ \wedge CLK \uparrow \\ Q_{s \to n} &= Q_s \wedge ((E \wedge D_n) \vee (\neg E \wedge Q_n)) \ \wedge CLK \uparrow \end{aligned}$$

By considering the fact that a signal is not significant as well as non-significant at the same time, i.e $Q_n \wedge Q_s = \texttt{F}$, we end up with the following conditions:

$$Q_{n \to s} = E \wedge D_s \wedge Q_n \ \wedge CLK \uparrow \quad (2.1)$$

$$Q_{s \to n} = E \wedge D_n \wedge Q_s \ \wedge CLK \uparrow \quad (2.2)$$

where $Q_{n \to s}$ indicates non-significant to significant change of the output Q and $Q_{s \to n}$ indicates significant to non-significant change of signal Q. Condition 2.1 shows that Q changes from

non-significant to significant when E is high (T) and D is significant. Condition 2.2 shows that Q changes from significant to non-significant when E is high (T) and D is non-significant.

With syntax $DFFE.RTL2$, we obtain the following equations for significance of Q.

$$\begin{aligned} Q_s &= (E \wedge D)_s \wedge (Q \wedge \neg E)_s \ \wedge CLK \uparrow \\ Q_n &= (E \wedge D)_n \vee (Q \wedge \neg E)_n \ \wedge CLK \uparrow \end{aligned}$$

which are reduced by simplification, according to rules 4 as

$$\begin{aligned} Q_s &= D_s \wedge Q_s \wedge E_s \ \wedge CLK \uparrow \\ Q_n &= E_n \vee D_n \vee Q_n \ \wedge CLK \uparrow \end{aligned}$$

The conditions for change in significance after boolean simplification are given as

$$\begin{aligned} Q_{n \to s} &= (E_s \wedge D_s \wedge Q_s \wedge Q_n) \ \wedge CLK \uparrow \\ &= \mathtt{F}(false) \\ Q_{s \to n} &= ((E_n \vee D_n \vee Q_n) \wedge Q_s) \ \wedge CLK \uparrow \\ &= (E_n \vee D_n \wedge Q_s) \ \wedge CLK \uparrow \end{aligned}$$

These conditions show that a change in non-significant to significant value of Q does not exist, however Q may change from significant to non-significant value if either E or D is non-significant.

After these results are propagated to the Q, according to the proposed definition of a control signal, the model $DFFE.RTL$ leads to the conclusion that E is the control input and D is the data input, because significance change of Q depends on value of E and significance of D. The analysis of the model $DFFE.RTL2$ concludes that D and E both are data inputs because significance change of Q depends neither on value of E nor on that of D.

The analysis with syntax $DFFE.RTL2$ shows that semantics result an under-approximation with respect to control input recognition criteria. The intuitive control input E is detected as *data* rather than *control*. We have seen that the semantics $\mathcal{S}_0$ give different results for different syntactic representations of the same module. Therefore, semantics $\mathcal{S}_0$ are weak and do not fulfill the main rule of syntax independence.

2.4.2 Semantics $\mathcal{S}_1$

We see that static analysis on DFFE with semantics $\mathcal{S}_0$ gives ambiguous results. Semantics $\mathcal{S}_0$ needs refinements. The properties of basic boolean operators $\wedge$, $\vee$ are used to define a new semantics called $\mathcal{S}_1$. Thus rule 4 of the semantics $\mathcal{S}_0$ is refined for operators $\wedge$ and $\vee$ by following additional rules:

1. For operator $\wedge$, if one operand has the significant low value (that is F) then result is also *significant low* whatever is the significance of other operand. If all operands have the significant high value (that is T) then the result is *significant and high* otherwise the result is *non-significant*.

2. For operator $\vee$, if one operand has the significant high value (that is `T`) then result is also *significant high* whatever is the significance of other operand. If all operands have the significant low value (that is `F`) then the result is *significant and low* otherwise the result is *non-significant*.

With this new semantics the analysis of $DFFE.RTL$ is unchanged but the equations of the significance changes of Q in $DFFE.RTL2$ become, after simplification:

$$Q_{n\rightarrow s} = (Q_n \wedge D_s \wedge E \wedge E_s) \wedge CLK \uparrow \quad (2.3)$$

$$Q_{s\rightarrow n} = (Q_s \wedge (D_n \wedge E \vee E_n \wedge (Q \vee D))) \wedge CLK \uparrow \quad (2.4)$$

From condition 2.3, we can conclude that E is a control input because condition for change in significance $Q_{n\rightarrow s}$ involves value of E. But from significance change $Q_{s\rightarrow n}$ in condition 2.3, we can also conclude that D is a control input due to presence of value of D in the condition. Thus semantics $\mathcal{S}_1$ doesn't identify the right control inputs. In order to understand where it comes from, we list in table 2.3 all the valuations of Q, E and D satisfying S_s where:

$$S = ((D \wedge E) \vee (Q \wedge (\neg E)))$$

From the last two lines in the truth table (shown in bold), we see that when $Q =$ `F` and E

Q	**E**	**D**	**S**$_s$
F	F	F	F
F	F	f	F
F	F	T	F
F	F	t	F
F	T	F	F
F	T	T	T
T	F	F	T
T	F	f	T
T	F	T	T
T	F	t	T
T	T	F	F
T	T	T	T
F	**f**	**F**	**F**
F	**t**	**F**	**F**

Table 2.3: Truth table of $((D \wedge E) \vee (Q \wedge (\neg E)))_s = S_s$

is non-significant (either `t` or `f`), the formula is significant if and only if $D =$ `F`. In the other valuations, value of D has no impact on the resulting significance.

The results with semantics $\mathcal{S}_1$ are an over-approximation with respect to control input recognition criteria. According to the results of the model $DFFE.RTL2$, the control input E is detected correctly, however the data input D is also detected as control. Thus $\mathcal{S}_1$ is also unable to detect the right control inputs independent of syntactic representations.

2.4.3 Semantics $\mathcal{S}_2$

We consider more refinements in the semantics of the DFFE example. The multiplexer structure in the $DFFE.RTL$ of Figure 2.4(a) is the central entity in this problem. It has a boolean representation in the form of VHDL assignment:

```
m <= (Q and (not E)) or (D and E);
```

which can also be represented with an `if-else` control structure:

```
if(E = '0') then
  m <= Q;
else
  m <= D;
end if;
```

This simple structure should allow for control inputs identification for the multiplexer and also for the $DFFE$ example by composition with a D flip-flop. We derive a partial specification for the a new semantics $\mathcal{S}_2$ for the multiplexer as follows:

- A single value change of one of the two data inputs D, Q cannot change the significance of the output,
- There must exist a pair (D, Q) such that a value change of E changes the output significance.

The symmetry is also needed to ensure the proper control inputs recognition. So all significant, none significant cases, and the monotonic property of significance for multiplexer function $m(Q, D, E)$ are given as:

- **Symmetry:** $m(Q, D, E)_s = m(Q, D, \neg E)_s$
- **All significant:** $Q_s \wedge D_s \wedge E_s \Rightarrow m(Q, D, E)_s$
- **None significant:** $Q_n \wedge D_n \wedge E_n \Rightarrow m(Q, D, E)_n$
- **Monotony-1:** Changing the significance of any input cannot change the output significance the opposite way
- **Monotony-2:** For a given significance of the selector, if a data input is significant and the other is not then it cannot be that selecting the significant input produces a non-significant output while selecting the other produces a significant output.

We implemented an exhaustive search algorithm which finds a suitable semantics taking into account the above constraints. The exhaustive search with these constraints leaves us a semantics named $\mathcal{S}_2$ for the significance of the multiplexer which states that:

> "*the output of multiplexer is significant if and only if the selector and the selected input are significant*".

We then performed an other exhaustive search on the semantics of $\wedge$ and used the property $(Q \vee D) = \neg(\neg Q \wedge \neg D)$. The search explores all the semantics of $\wedge$ and computes the distance between $m(Q, D, E)$ and $(Q \wedge \neg E \vee D \wedge E)$, that is the number of valuations of Q, D and E (64 cases) for which $m(Q, D, E) \neq (Q \wedge \neg E \vee D \wedge E)$. We conclude from this exhaustive test that there is no semantics which satisfies that $m(Q, D, E)_s = (Q \wedge \neg E \vee D \wedge E)_s$. The detailed implementation of these constraint systems is provided as C functions in Appendix A.

Thus the problem of boolean syntax dependency persisted even with $\mathcal{S}_2$. The intuitive semantics of multiplexer $\mathcal{S}_2$ also does not fulfill the main rule of syntax independence.

2.4.4 Concluding remarks

We concluded from the analysis of basic boolean structures that it is impossible to overload the boolean operators for the extended boolean logic in a way that allows for control input recognition and fulfill the main objective of syntax independence. We have used exhaustive simulations to prove this fact, however it can also readily be proved mathematically that significance extended logic does not allow control input recognition in a satisfiable manner. A few suggestions are proposed to overcome this problem.

- Use pattern recognition techniques to identify multiplexer like structures in their boolean form.
- Replace each of them by an `if-else` equivalent structure.
- Apply semantics $\mathcal{S}_2$ to the `if-else` structures and semantics $\mathcal{S}_1$ to the boolean operators `and` ($\wedge$), `or` ($\vee$), `not` ($\neg$).
- When the condition of a `if-else` evaluates as non significant propagate this to all the signal assignments nested in one or the other branch.

Implementing such pattern recognition techniques to identify right control inputs would lead to similar syntactic analysis used in existing techniques mentioned in previous section whereas we wanted to avoid such analysis. Therefore, the definition of control and mentioned semantics are not sufficient to extract control in the way we intended. We either need to redefine the control and data inputs in an other way or additional information from the designer could serve us to obtain more refinements.

In the next section, we describe an analysis in which we take into account the knowledge provided by the designer about the control and data inputs of the module and observe the coherence between the designer's information and the obtained results.

2.5 Analysis with designer's intervention

We discovered in the preceding section that no suitable semantics exists for the basic building block (multiplexer) responsible for control operations of digital circuits by overloading the classical boolean signals with significance. We therefore, introduced the designer's interventions in the analysis. We suppose that a separation of control and data inputs is already provided. We use this hypothesis with our notion of significance and develop an analysis for control input recognition, and conclude that the results obtained by the analysis are coherent with the hypothesis.

We suppose that a given set of inputs is tagged as 'data' and 'control'. This is called *input hypothesis*. By this knowledge of control and data inputs, we make an analysis to verify that the given separation of control and data inputs conforms to our intuitive definition.

According to the intuitive definition of Control, the data inputs do not carry timing information and their values do not impact the timing of the outputs. Only their significance can have an impact on the output significance. We therefore take into account only the significance of labeled data inputs. However, values of control inputs have always impact on the timing of outputs. In other words they are always active (significant), and have an impact on significance of output results. Therefore, control inputs are always significant.

Based on the input hypothesis, we analyze the basic logic functions to obtain *significance extended boolean equations* representing significance of output. The criteria for control input recognition is following

> ***The presence of value of the supposed control signal in normalized (minimized) equation of output significance will be an indication that significance of the output depends on the value of the supposed control signals.***

We will derive the conditions for the *change* in output significance, and reason about the impact of value of control signals on the output significance change with different hypothesis of control and data inputs. With 'significance' attribute associated with the given data inputs, and 'value' attribute associated with the given control inputs, we try to build semantics to analyze the basic boolean functions.

Control/Data propagation

We define how the control and data information provided at the inputs is traversed through boolean functions of the module. For two inputs A and B we give semantics of control and data signals as shown in table 2.4, where $\Diamond$ represents control/data propagation semantics for AND ($\wedge$) and OR ($\vee$) functions, 'c' stands for *Control* and 'd' stands for *Data*. For NOT($\neg$)

A $\Diamond$ B	c	d
c	c	d
d	d	d

Table 2.4: Control/Data propagation

function, if input is control then output is also control, if input is data then output is also data. In case of `if-else` structure if a data signal appeared in condition then all the nested assignments are also considered as data.

Note that by this semantics all the internal registers and outputs would become 'data' except those who are totally independent of tagged data inputs. Thus control registers and outputs are calculated only and only from the tagged control inputs.

Analysis for DFFE module

With control/data propagation semantics and the input hypothesis, we analyze $DFFE.RTL$ and $DFFE.RTL2$ modules. For $DFFE.RTL$ of Figure 2.4, we have 2 inputs, and thus 4 possible hypothesis. Current state of Q coincides to input D due to direct assignment from D to Q. Therefore, D and Q will both either be 'data' or 'control'. Output significance change conditions are given in table 2.5 for 4 cases.

In table 2.5, we have considered that $CLK \uparrow$, is implicitly present in each of the condition. For control and data separation, only hypothesis 1 and 2 are of interest. In hypothesis 1, presence of E in both conditions implies that E *is control* which is coherent with the hypothesis. Thus we can say that the separation, E *is control* and D *is data* is correct. However in case of hypothesis 2, we see that supposed control signal D does not appear in significance change conditions due to the semantics of `if-else`. Thus results do not correspond to supposition. We can say that the separation, E *is data* and D *is control* might not be valid.

Hypothesis	**D**	**Q**	**E**	$Q_{n\to s}$	$Q_{s\to n}$
1	d	d	c	$D_s \wedge E \wedge Q_n$	$D_n \wedge E \wedge Q_s$
2	c	c	d	$E_s \wedge Q_n$	$E_n \wedge Q_s$
3	d	d	d	F	$(E_n \wedge Q_s) \vee (D_n \wedge Q_s)$
4	c	c	c	–	–

Table 2.5: Control and data separation for $DFFE.RTL$ with designer's intervention

For $DFFE.RTL2$ shown in Figure 2.5, we have output composed of AND, OR and NOT functions. For AND and OR functions with 2 inputs A and B, there are four possible hypothesis given as follows.

1. Both A and B are control
2. Both A and B are data
3. A is control and B is data
4. B is control and A is data

Similarly for NOT function we have two possible hypothesis: either A is control or A is data.

Using control data semantics of table 2.4 and above hypothesis, we obtain *output significance conditions* of AND, OR and NOT functions for each case. These conditions describe which input combinations produce significance or non-significant output. Based on output significance conditions for basic logical functions, we analyze $DFFE.RTL2$ with different combinations of input signals E, D and internal signal Q. For eight possible cases of control and data with 3 signals, we obtain conditions for change in significance of output Q after simplifications. The conditions are simplified automatically using boolean functions simplification (logic minimization) program based on Quine-McCluskey [87] and BDD [20] methods and are listed in table 2.6.

In hypothesis 1, where E is 'control' and D and Q are supposed as 'data', we see that change in significance $n \to s$ as well as $s \to n$ for the output Q are possible only in the presence of value of E. We can say that value of E has always impact on the change in significance of the output in both ways. These conditions don't contain any value dependency term of D, we can conclude that E is a control signal and D is the data. Thus given hypothesis that D is data and E is control, is valid.

According to hypothesis 2, although D and Q are 'control', we observe that a change $n \to s$ of output Q might happen even in the absence of value of D and Q when the term $E_s \wedge Q_n$ becomes true. So the values of D and Q do not always have an impact on significance of Q. Similarly we see that the change $s \to n$ occurs either in the presence of D or Q, but not in the presence of both. These terms do not fulfill the hypothesis that D is control and E is data. So the given hypothesis of control and data does not correspond to the resulting analysis.

In hypothesis 3, we see that due to contradiction $Q_n \wedge Q_s = \texttt{F}$, the non-significant to significant change of output Q does not exist. Therefore, it is ambiguous to reason about the change in significance in this case. In hypothesis 4, we have output as always significant and hence no condition of significance change.

In case of hypothesis 5, we see that significance change $n \to s$ does not depend on value of supposed control input D therefore hypothesis is not coherent with the result. In hypothesis 6, we see that both changes in significance are independent of value of Q while Q was supposed

to be a control signal. Hence the hypothesis is not coherent with the derived results. In case 7 and 8, change in significance $n \to s$ does not exist.

Hypothesis	D	Q	E	$Q_{n\to s}$	$Q_{s\to n}$
1	d	d	c	$D_s \wedge E \wedge Q_n$	$D_n \wedge E \wedge Q_s$
2	c	c	d	$(\neg D \wedge \neg Q \wedge Q_n) \vee (E_s \wedge Q_n)$	$(D \wedge E_n \wedge Q_s) \vee (Q \wedge E_n \wedge Q_s)$
3	d	d	d	F	$(E_n \wedge Q_s) \vee (D_n \wedge Q_s)$
4	c	c	c	–	–
5	c	d	c	$E \wedge Q_n$	F
6	d	c	c	$(\neg E \wedge Q_n) \vee (D_s \wedge Q_n)$	$E \wedge D_n \wedge Q_s$
7	c	d	d	F	$D \wedge E_n \wedge Q_s$
8	d	c	d	F	$Q \wedge E_n \wedge Q_s$

Table 2.6: Control and data separation for $DFFE.RTL2$ with designer's intervention

We have observed in this analysis that all the hypothesis from 2 to 8 have ambiguous resulting equations of change in significance. Either we have the change in significance only in one way, or the hypothesis of control inputs does not correspond with the results of the analysis. The only hypothesis in which results are coherent is the hypothesis 1, in which the change in significance in both ways is possible only in the presence of value of the supposed control signal E. Hence the hypothesis that E *is control* and D *is data* is the correct separation and rest of the hypothesis might not be the correct separations for $DFFE.RTL2$.

Drawback

The main drawback of the analysis with designer's interventions is also the syntax dependency. With different syntactic representations of same boolean function, we obtain different output significance change conditions. For instance, consider two syntactic forms of a 3-variable boolean function $f(x, y, z)$ as follows

$$f(x, y, z) = (x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z) \tag{2.5}$$

If we supppose that x is *control*, and y and z are *data* then by evaluating both forms with $x = \texttt{T}$, $y = \texttt{f}$, and $z = \texttt{F}$, we obtain $f(x, y, z) = \texttt{F}$ with syntactic form on left side, and $f(x, y, z) = \texttt{f}$ with syntactic form on right side. This limitation might result a wrong separation of control and data to be a right one. In this way, the analysis using designer's interventions also does not fulfill our main objective of syntax independence.

2.6 Practical considerations of the definition of control

We describe some practical issues of significance extension in relation with VHDL simulation semantics. According to our proposed definition, value of control inputs should have an impact on the timing of the outputs. By contrast the value of the data inputs should not have an impact on the timing of the outputs. This definition is somehow difficult to use as the careful study of the D input of $DFFE$ will show. In order to change the significance of the output S, following two condition must hold:

- The process assigning S must resume,
- The current and the assigned value must have different significances.

The equivalent process of the concurrent signal assignment `S <= Q;` is:

```
process(Q)
begin
  S <= Q;
end process;
```

So, the first condition is an event (a value change) on the internal signal Q, that is, during the previous simulation step:

$$E \wedge (D \neq Q)$$

And this involves the value of D, not only its significance! The simulation traces in Figure 2.6 illustrate this point and prove that a single value-only change of D between two identical simulation sequences has an impact on the significance of S. The delta cycles are shown as small time shifts to improve readability. Q and D are supposed initialized by `f` at the beginning of the simulation. A solution to this problem could be to adapt the VHDL simulation

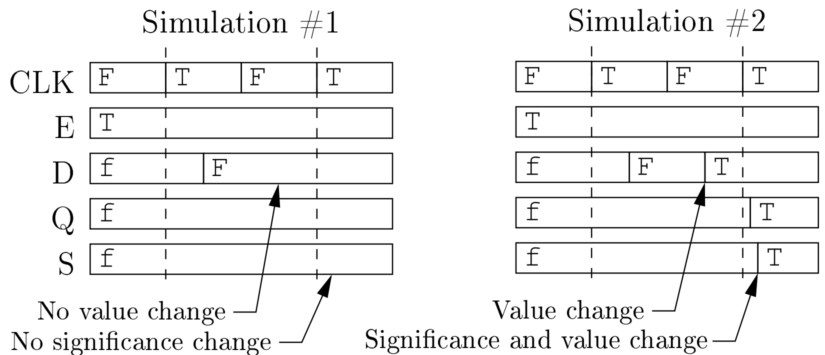


Figure 2.6: Two traces showing D as a control input

algorithm, and decide that a process resumes on events and / or significance changes of any signal in its sensitivity list. With adaptive simulation, traces would become those of Figure 2.7. With this new simulation paradigm the single value change of D doesn't cause a significance

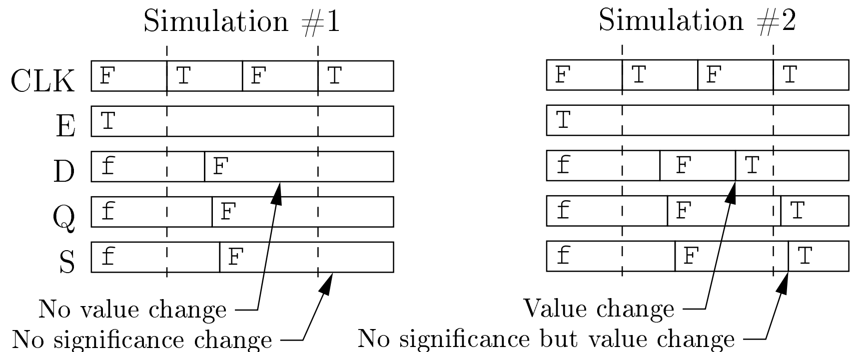


Figure 2.7: Two traces not showing D as a control input

change of S. There must also be a significance change of D.

2.7 Consequences of the research

We have discussed the problem of separating control and data processing in hardware modules. Existing techniques to distinguish control and data processing are based on the subjective choice between the control and data. Classical control and data flow techniques of software have previously been applied on HDL syntax to partition control and data targetting different applications. We envisaged to obtain a precise and formal separation between control and data processing as a preliminary step towards assisting data abstraction and functional verification.

We tried to semantically define an abstract notion of control in low level hardware modules based on the fact that control inputs impact the timing of the outputs of the module. An attribute called *significance* is introduced to represent the timing impact of the control inputs on the outputs. Some experiments and static reasoning have been developed to check the suitability of the definition of control using intuitive semantics.

We worked with elementary hardware blocks without and with designer's intervention to find a suitable semantics for Control recognition. We concluded by reasoning that according to the proposed definition of control, it is impossible to identify the control and data inputs of an unknown hardware module in a satisfiable manner because of syntax dependent significance computations for a boolean function. The analysis does not result a unique and natural separation between control and data for different syntactic forms. The proposed definition of control needs additional refinements.

The question of precise control and data separation independent of designer's hints and syntax has been raised, but remains open for the research community. We believe that the proposed definition and analysis could possibly serve to open new track of research. The problem of control-data separation irrespective of syntax could either be tackled by enriching the definition of control or by redefining the control and/or data in a way other than what we have proposed with the notion of control containing timing oriented behavior of the hardware modules.

To fulfill the objective of functional verification and abstraction, we considered that designer indicates the data inputs of his design. Based on this knowledge we benefit from existing syntax analysis techniques such as *slicing* to separate control and data in hardware modules as explained in the subsequent chapter.

The "*Significance*" introduced in this chapter, did not fulfill our envisaged goal of Control/-Data separation. However, the notion of significance serves us to study important properties of hardware modules concerning boolean data dependencies to assist static formal verification of data flows as explored in chapter 4 and 5.

Chapter 3

Control and Data separation using slicing

3.1 Introduction

We have shown in the precedent chapter that a unique separation of control and data is not achievable based on the intuitive definition of control due to syntax dependent behavior of the extended semantics for boolean operators. We therefore suppose, that control and data inputs are known from the designer. Based on this knowledge, we present a control and data separation solution using program slicing techniques. A *control-data slicing algorithm* is presented to split a given VHDL description into a control state machine and a data processing machine.

The proposed solution makes use of existing HDL slicing techniques proposed in the literature to obtain the objective of control and data separation for assisting abstraction and verification. The technique is based on the conventional syntactic analysis with customizations, which enables us to obtain the desired goal of control/data separation.

A background of the existing slicing techniques is first presented in section 3.2. We describe control-data slicing algorithm for synthesizable subset of VHDL in section 3.3. We talk about the synchronization problem while slicing VHDL modules with local variables in section 3.5 and propose a solution in terms of structural transformation and enhancements. Implementation of slicing and its applications are provided in section 3.6 and 3.7 respectively.

3.2 Program slicing basics

Program slicing originally proposed by M. Weiser [106] is a classical static analysis technique to isolate a smaller piece of program from a larger program such that the semantics are preserved. It is a program decomposition technique to extract a set of statements from a given program relevant to a particular computation called the "*slice*".

During slicing process, a slice from a given program is extracted with respect to a *slicing criteria*, which specifies a location and a variable. The slicing criteria specifies the impact of a program variable at a specific point on rest of the program, or the impact on a program variable at a specific point by a piece of the program.

Let v be a variable, and s be a label of some program statement, the slicing criteria is denoted by the pair $< v, s >$. There are two kind of slices: a *backward slice* of a program

with respect to a slicing criteria $< v, s >$, is the set of all program elements that might affect (directly or transitively) the value of variable v at statement s. A *forward slice* with respect to slicing criteria $< v, s >$ is the set of all program statements that might be affected by the computations performed on v at s.

Figure 3.1 illustrates the slicing by an example. Original program is shown on left side (Figure 3.1(a)), and on the right side (Figure 3.1(b)), we have shown two slicing criteria with a forward and backward resulting slices. If we refer each program statement by its line number, then slicing criteria for slice # 1 is $< SUM, 6 >$ i.e. value of variable `SUM` at statement on line 6. We see variable `SUM` being defined at line 6 is only used at line 11. No other statement impacts the value of `SUM` between lines 6 and 11. Thus only statement `WRITE(TOTAL,SUM);` at line 11 is affected by `SUM` at line 6. Therefore, forward slice according to criteria $< SUM, 6 >$ is single statement at line 11.

For slice # 2, slicing criteria is $< TOTAL, 11 >$. The statement `WRITE(TOTAL,SUM);` at line 11 has direct dependency on statement at line 9, and line 3 due to variable `TOTAL` . Since variable `TOTAL` at line 9 is also dependent on variable `X` and `Y` , therefore we have further dependency on line 2 where `X` and `Y` are being read. The resulting backward slice is given on bottom of Figure 3.1(b).

```
1. BEGIN
2.    READ (X,Y);
3.    TOTAL := 0;
4.    SUM := 0;
5.    IF (X < 1) THEN
6.          SUM := Y;
7.    ELSE BEGIN
8.          READ(Z);
9.          TOTAL := X*Y;
10.   END
11.   WRITE(TOTAL,SUM);
12. END
```

(a) Original program

```
// Forward slice::- (slicing criteria= <SUM,6>)

1. BEGIN
2.        WRITE(TOTAL,SUM);
3. END

//**********************************************

// Backward slice::- (slicing criteria= <TOTAL,11>)

1. BEGIN
2.     READ(X,Y);
3.     TOTAL := 0;
4.     IF (X < 1) THEN
5.           NOP     // No operation
6.     ELSE
7.         TOTAL := X*Y;
8. END
```

(b) Slices

Figure 3.1: Example of slicing

3.2.1 Dependence graph based slicing

Program slicing is done by performing control and data flow analysis on some suitable intermediate representation of programs such as abstract syntax trees (AST) or control flow graph (CFG) [8]. The CFG of a program is a directed graph in which nodes represent statements, and directed edges represent flow of control of the program execution. We mention an overview of the basic sequential program slicing. A substantial amount of research has been done in this domain such as [71], [70], [75], [104].

Ottenstein et al. [89] defined slicing as the reachability problem in program dependence graph (PDG). A PDG is a directed graph in which statements of the program constitute nodes, and edges among nodes represent control and data dependencies among statements. In all dependence graph based approaches, slicing criteria is identified by a node in PDG. For single procedure programs, the slice with respect to v consists of all nodes from which v is reachable. To incorporate procedures, Horowitz et al.[49] introduced notion of system dependence graph

(SDG) in program slicing. An SDG contains a set of PDGs for each procedure with control and data dependencies marked among PDGs.

3.2.2 Slicing in hardware description languages

E.M. Clarke in [27], [28] has presented first VHDL slicer and its applications. The VHDL slicing consists of converting source code into an appropriate intermediate representation (IR) in the form of control flow graphs (CFG) and dependency graphs (PDG and SDG) using syntactic and semantic analysis. VHDL description is a collection of concurrently executing processes. Each process is basically an infinitely executing sequential program. Therefore, a process can be represented by the control flow graph (CFG).

An example VHDL process with corresponding CFG is shown in Figure 3.2. Each node corresponds to VHDL sequential statement within process which contains two sets of variables named DEF and USE [8]. DEF is singleton which contains variable *written* at the corresponding statement, whereas USE contains variables that are *read* in that statement. The corresponding line numbers of statements in source code are marked with each node in the CFG. Local control-data flow dependency analysis is performed on the CFG to obtain process dependence graph (PDG) whereas global control-data flow dependency analysis results the system dependence graph (SDG) of the module.

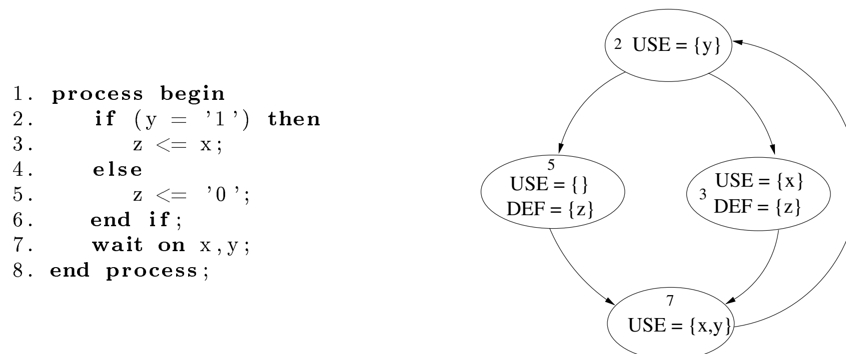


Figure 3.2: An example of Control Flow Graph (CFG)

The slicing criteria, as proposed in [28], is a signal or variable assignment statement in the description according to which program is to be sliced.

Vedula et al. [103] presented Verilog slicing to improve hierarchical test pattern generation for simulations. S. Ramesh et al. [92] have also made a similar contribution for synchronous language Esterel and VHDL. Confora et al. [23] introduced the notion of *conditioned slicing.* Conditioned slicing augments static slicing by introducing a condition that specifies the initial set of states in the slicing criteria [102].

For our concerned IP modules with a mix of control and data processing, we were interested in slicing techniques proposed for HDL. Therefore, we have adopted the basic technique proposed in [27] and [28].

We have proposed a control-data slicing algorithm which uses control flow graphs (CFG) as intermediate form to represent VHDL processes, and implements a similar dependency analysis to separate control and data in VHDL modules as described below.

3.3 Control-data separation using slicing

We will present a Control and Data separation technique in synchronous VHDL modules based on slicing techniques. The objective is to obtain separate models for the two different behaviors of the module communicating with each other in terms of *controller* and *datapath*. Proposed slicing operation has been depicted in Figure 3.3. We assume that some inputs are declared as data. This information is used as slicing criteria. Note that the declared data inputs are not supposed to carry any control information in this case. In other words, we are not treating those modules in which data inputs are used to carry sometimes the control information. The control information is supposed to be provided by declared control inputs only. Starting from the declared data inputs, we traverse the VHDL description recursively by data flow analysis to identify the outputs and intermediate registers, directly or transitively affected by data inputs.

We then partition the design into a *data slice* and a *control slice*. The data slice contains registers and outputs affected by data inputs with corresponding combinational logic. The *control slice* contains registers independent of data inputs with corresponding combinational logic. We regenerate VHDL code of control and data slices, and interconnect them with each other. This representation collectively preserves the functionality of the original module. The

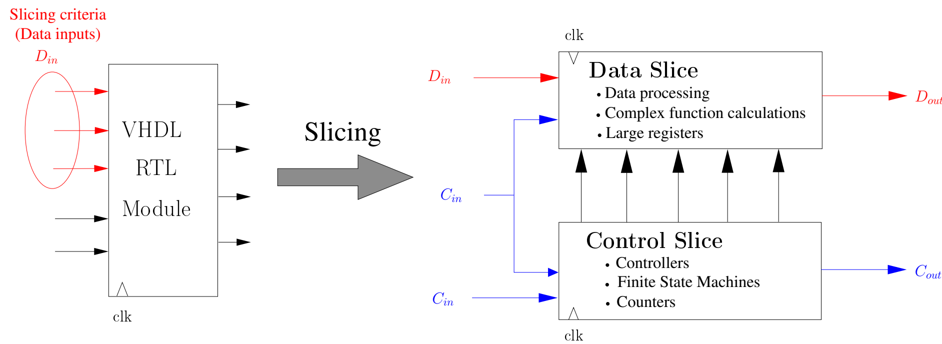


Figure 3.3: Control/Data separation in IP modules by slicing

sliced representation of the module resembles a class of FSMD representations [15] in which signals originated from control slice are called *commands* for various data operations in the data slice.

We have shown a generic form of control and data separation in Figure 3.4 which is closer to the designer's intuition of the hardware in the form of communicating control state machines (control slice) and datapath (data slice). The model shown in Figure 3.4 is a representation of a system with data dependencies, since the controller behavior depends on the data register values, as indicated by arrows from data slice towards control slice. Such controllers are called *data dependent controllers*. Signals originated from data slice are *status* of the data register.

In relation to the discussion on definition of "control" in precedent chapter, we have preserved our basic intuition of control with restrictions that control is never affected by data. Therefore, in proposed slicing, every register or combinational element within the module depending on data inputs directly or transitively would become part of the data slice. If the designer anticipates that there is a control state machines dependent on data inputs or data registers, then it would be present in the data slice. This criteria is adopted, so that

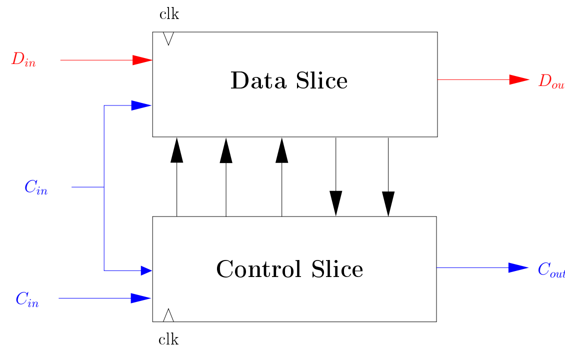


Figure 3.4: Data dependent separation of control and data

we are able to apply data abstraction techniques on the data slice without losing the control information.

However, it is possible to incorporate the control/data separation according to the designer's anticipation in terms of control state machines and datapaths as shown in Figure 3.4. In that case, we would need to revise the definition of Control given in chapter 2, which would also describe the data dependent behavior of the control.

3.3.1 VHDL description of modules

For our analysis we are considering modules described in synthesizable subset of VHDL. We will give basic definition of module described in VHDL as follows.

Definition 3.1 (Entity, Architecture and Process).

- *An* ***Entity*** $\mathcal{E}$ *of a module is a 3-tuple* $(\mathcal{I}, \mathcal{O}, \mathcal{A})$ *where* $\mathcal{I}$ *is the set of input interface signals (input ports),* $\mathcal{O}$ *is the set of output interface signals (output ports), and* $\mathcal{A}$ *is the architecture associated with entity.*

- *An* ***architecture*** $\mathcal{A}$ *is a 2-tuple* $(\hat{S}, \mathcal{P})$ *where* $\hat{S}$ *is the set of signals, and* $\mathcal{P}$ *is the set of processes in the architecture.*

- *Each* ***process*** $P \in \mathcal{P}$ *is a 3-tuple* $(\mathcal{L}, \mathscr{V}, \mathscr{S})$ *where* $\mathcal{L}$ *is the set of signals called 'sensitivity list' of process,* $\mathscr{V}$ *is the set of 'local variables' of process, and* $\mathscr{S}$ *is the 'sequence of sequential statements' within process body. An* execution path *of the process is a subset of* $\mathscr{S}$ *starting from the first statement to the last statement of the sequence*

We represent a process description with three basic sequential statements as given below:

1. *assignment statements*: `s <= exp;`, `v := exp;`

2. *predicate statements*: `if`, `if-else`, `if-elsif-else`

3. *suspension statements*: `wait on sig;`

We also consider high level sequential statements such as `case` statements, deterministic `for` loops, procedure calls, and concurrent `generate` statements. However, they are implicitly represented by statements of above three kinds. The `case` statement is represented by the

predicate statement with equivalent `if-elsif-else` structure. Deterministic `for` statement is unrolled with a sequence of sequential statements within the loop. Similarly, concurrent `generate` statements are also flattened to obtain a plain VHDL model with single architecture, and a set of processes with statements of above three kinds.

Synchronous VHDL description

Synchronous behavior can be implemented in VHDL in numerous ways. In most of the designs at RTL we describe it by **synchronous** and **combinational processes**. We denote the set of synchronous and combinational processes as $\mathcal{P}_s$, and $\mathcal{P}_c$ respectively, such that $\mathcal{P}_s \cap \mathcal{P}_c = \emptyset$.

A synchronous process is sensitive only to the *rising edge* (or falling edge) of a global signal known as *clock*. In our VHDL subset, A synchronous process $P_s \in \mathcal{P}_s$ is defined as follows

Definition 3.2 (Synchronous process)**.**
A synchronous process is a 3-tuple $P_s = (\mathcal{L}, \mathscr{V}, \mathscr{S})$ *where*

1. $\mathcal{L} = \{clk\}$ *is a singleton, where* $clk \in \mathcal{I}$ *is an input signal known as the "clock"*
2. $\mathscr{V}$ *is a set of local variables*
3. $\mathscr{S} = \{s_{pr}\}$ *is singleton where* s_{pr} *stands for predicate statement of following form*
   - s_{pr} ::= ***if*** *(bexp)* ***then*** $\mathscr{S}_{synch}$ *where*
     - *bexp is the syntax for rising edge on signal clk*[1]
     - $\mathscr{S}_{synch}$ *is a list of synchronous statements*[1]

We detect synchronous processes by using pattern matching methods used in general purpose logic synthesizers. Memory elements (registers) are only associated with synchronous processes. There exist other syntax patterns to describe synchronous processes as well. However, we consider only those patterns which are defined by IEEE standard [58], and are commonly used by VHDL designers.

A combinational process $P_c \in \mathcal{P}_c$ is defined as follows

Definition 3.3 (Combinational process)**.**
A combinational process is a 3-tuple $P_c = (\mathcal{L}, \mathscr{V}, \mathscr{S})$ *in which following conditions are fulfilled:*

- $\mathcal{L}$ *contains all signals being read inside the process body such that* $clk \notin \mathcal{L}$ *where clk is the clock input*
- *For all* $s \in \hat{S} \cup \mathcal{O}$*,* s *is assigned in all possible execution paths of the process*
- *A local variable* $v \in \mathscr{V}$ *is never read before it is written*

Any violation to the conditions mentioned in definition 3.3 may result latch inference which could cause different behavior of the circuit before and after synthesis. Other forms of combinational processes such as process statements with sensitivity list, concurrent signal assignment, concurrent conditional signal assignment or concurrent procedure calls are semantically equivalent, and are implicitly converted to standard form according to definition 3.3.

Since we are dealing only with synthesizable synchronous modules, therefore we do not treat models containing any process other than synchronous or combinational process in the descriptions.

[1] According to IEEE synthesis standard [58]

3.3.2 Basic slicing rules

We have implemented dependency analysis based slicing algorithm based on simple propagation rules to identify, and separate the *control* from *data*. The basic idea is to propagate the *data information* provided by designer at the inputs of the module to the internal description, and separate it from the circuits not depending on data information in all respects.

We suppose that the clock input is implicitly considered as global *control* input, and some of the inputs are labeled as *data*. Rest of the inputs are labeled as *control*.

Dependency analysis rules

1. Initially all the internal signals and registers are considered as *control*.

2. For any arithmetic and logical operation on the right hand side expression in an assignment statement, if any operand is *data* then the assigned signal also becomes data.

3. Individual constants assigned to internal signals are considered as *control* assignments.

4. For multiple assignments to a signal in a process, if any assignment results that signal as *data* according to rule 2, then all assignments to that signal are also considered as *data*.

5. Constants concatenated with data signals are considered as *data* constants.

6. If a *data* signal is appeared in the predicate statement, then all the statement inside the branches are also considered as data.

Partitioning rules

7. All assignments marked as *data* with corresponding predicate statements (control flow) are copied to a new module called *data slice*.

8. All assignments marked as *control* with corresponding predicate statements (control flow) are copied to a new module called *control slice*.

9. For each intermediate control signal being used in data slice, a new input port is created in data slice, and corresponding output port is created in control slice entity. Update the output port with corresponding signals

Above rules result a separation of control and data in such a way that all the assignments to control registers are moved to control slice, and all the assignments to data registers are moved to data slice. Pure data operations (operations among data signals only) reside in data slice. Pure control operations (operations among control signals only) reside in control slice. Operations between control and data are moved to data slice. Rules 1 to 6 implement a *dependency analysis*, and rules 7 to 9 implement a *partitioning* into control and data slice.

Slicing rules to obtain Data-dependent control

As discussed earlier, by applying above slicing rules, we would not be able to obtain a *data dependent separation* as shown in Figure 3.4. Thus one might think that all data dependent control would be moved to data slice using these slicing rules. But as remarked in chapter 2 this would be implied from the anticipation of the designer who would require that control should depend on data. One way to obtain the data dependent separation is to introduce some syntactic rules proposed in the literature [86], [50] which could be described in our context as below.

- If a predicate statement contains a *data* signal then it is considered as *data predicate statement*. A nested assignment statement inside that predicate statement assigning only control signals is considered as *control assignment statement*. It is moved to the control slice along with corresponding data predicate statement.
- For each data signal being used in control slice, a new input port is created in control slice and corresponding output port is created in data slice entity. The output ports are updated with corresponding signals.

These additional rules are used in place of rule 6. With these rule, operations between control and data are normally moved to data slice except the case in which a control assignment statement is nested (controlled) by a conditional statement with data signal in the condition. In that case, the *control assignment statement* with corresponding *data predicate statement* is moved to control slice after restructuring. We have implemented these additional rules separately to obtain the slicing with data dependent Control. However, such modules are not treated further for proposed data abstraction techniques because timing information depending on data would possibly be lost.

All of the above rules are implemented in the form of control-data slicing algorithm on intermediate representation of processes as described below.

3.3.3 Intermediate representation for VHDL modules

The control-data slicing is implemented on intermediate representation of VHDL source code. We have considered Control Flow Graphs (CFG) as a convenient data structure for our static dependency analysis. Each VHDL process $P \in \mathcal{P}$ is uniquely represented by a control flow graph in which nodes represent sequential statements, and edges represent the control flow among statements.

Definition 3.4 (Control Flow Graph).
A control flow graph $CFG = (N, E)$ of a VHDL process in a module is a labeled directed graph in which

- *N is a set of nodes that represent statements in a process. $N_A \subset N$ is set of* ***assignment nodes****. $N_P \subset N$ is set of* ***predicate nodes****. $n_s \in N$ is* ***suspension node****. Therefore, $N = N_A \cup N_P \cup \{n_s\}$*
- *$E \subseteq N \times N$ is a set of labeled edges that represent the control flow between nodes where each $n_p \in N_P$ has two outgoing edges labeled T (True) and F (False) respectively, and each $n_a \in N_A$ has one outgoing edge labeled Q (sequential). n_s has one outgoing edge labeled R (Resume)*

There are some associated terms in CFG which are used to establish the control-data slicing rules.

- We denote a node $n_0 \in N$ as *entry node* which corresponds to first statement of the process.
- A ***path*** in CFG is a sequence of nodes such that from each of the nodes there is an edge to the next node in the sequence. Paths in a CFG are cyclic where nodes n_0 and n_s are repeated.
- Node m ***dominates*** node n, if every path from entry node n_0 to n goes through m.
- Node m ***postdominates*** node n, if every path from n to the suspension node n_s has to pass through node m. The n_s postdominates all nodes in the CFG.
- Node m is ***nested*** by node n, if n dominates m, and all the nodes in the path between n and m including m, do not postdominate n.

DEF and USE sets:

Two sets named DEF and USE are associated with each node of CFG whose elements are signals defined or used in the corresponding statements. DEF is singleton associated to an $n_a \in N_A$ containing the signal written in the corresponding assignment statement. The $DEF = \emptyset$ (empty set) is associated with other nodes. When associated to an $n_a \in N_A$, USE represents set of all the signals present in the right hand side expression of corresponding signal assignment statement. For each predicate node $n_p \in N_P$, USE is the set of signals present in the boolean expression of corresponding predicate statement. For suspension node n_s, USE is the set of signals appeared in the sensitivity list of the statement. We denote DEF_n and USE_n as sets associated with the node $n \in N$.

Since some of the inputs are labeled as *data* by the designer, therefore for entity $\mathcal{E}$ of the module, $\mathcal{I}$ is divided into two subsets $\mathcal{D}_{in} \subseteq \mathcal{I}$, and $\mathcal{C}_{in} \subseteq \mathcal{I}$ called *data inputs* and *control inputs* respectively such that $\mathcal{D}_{in} \cap \mathcal{C}_{in} = \emptyset$.

With the knowledge of data inputs $\mathcal{D}_{in}$ and a set of CFGs of a given module, we perform a *dependency analysis* in which we investigate the flow dependencies among various signals within module's entity. This is done by constructing a dependency graph in which nodes represent the nodes of CFGs and the input signals, whereas edges represent flow dependencies among nodes calculated using DEF and USE sets of the nodes.

Let each input $d_i \in \mathcal{D}_{in}$ is represented by a dummy node called ***data input node***. We have therefore, a new set of data input nodes denoted as N_d. Let us denote $CFG_x = (N_x, E_x)$ as a CFG associated to a process $x \in \mathcal{P}$. The Global Flow Dependency Graph (GFD) associated to $\mathcal{E}$ is defined as below:

Definition 3.5 (Global Flow Dependence Graph).
Global Flow Dependence Graph is tuple $GFD = (\mathbf{P}, \mathbf{D})$ where

- $\mathbf{P} = (\bigcup_{x \in \mathcal{P}} N_x) \cup N_d$ *is set of nodes*
- $\mathbf{D} \subseteq \mathbf{P} \times \mathbf{P}$ *with* $(n_1, n_2) \in \mathbf{D}$ *such that*
  - *either there is a signal* $s \in DEF_{n_1}$, *and also* $s \in USE_{n_2}$ *that is,* $DEF_{n_1} \cap USE_{n_2} \neq \emptyset$

– *or there is a signal* $s \in \mathcal{D}_{in}$, *and also* $s \in USE_{n_2}$.

Each edge $d \in \mathbf{D}$ is labeled as D (Data). We define the 'dependence path' to represent static data dependence from input nodes to various nodes within GFD as follow.

Definition 3.6 (Dependence path).
A **dependence path** π_d *from node* $n_i \in \mathbf{P}$ *to* $n_k \in \mathbf{P}$ *is a sequence of nodes* $n_i, n_{i+1}, n_{i+2}, ..., n_k \in \mathbf{P}$ *such that for every consecutive pair of nodes* (n_j, n_{j+1}),*we have* $(n_j,\ n_{j+1}) \in \mathbf{D}$.

3.3.4 Slicing algorithm

We describe the slicing algorithm which implements control-data slicing rules mentioned in 3.3.2 based on formal definitions of CFG and GFD as shown in Algorithm 1.

Algorithm 1 Slicing algorithm

Require: A set of CFG, set of data inputs
Ensure: Control slice and Data slice

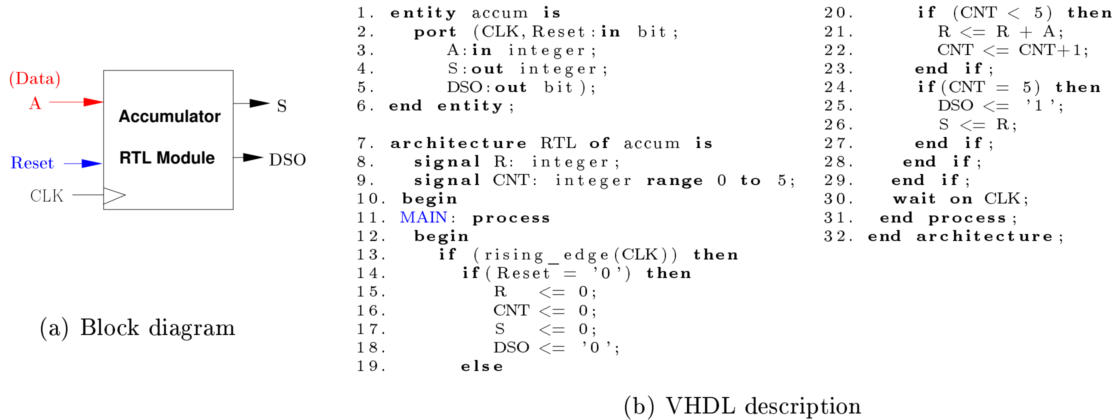
– Initially mark all nodes as "control". (rule 1)
– For $n_i, n_j \in \mathbf{P}$ if $\exists \pi_d$ from $n_j \in \mathcal{D}_{in}$ to n_i, then mark n_i as "data". (rules 2, 3, 4, 5)
– For all assignment nodes $n_1, ..., n_k$ such that $DEF_{n_1} \cap DEF_{n_2} \cap ... \cap DEF_{n_k} \neq \emptyset$, if $\exists i, 1 \leq i \leq k$ and n_i is marked as "data", then mark $n_1, ..., n_k$ as "data". (rules 2, 3, 4, 5)
– For each $n_i \in \mathbf{P}$ such that n_i is a predicate node, if $\exists x \in USE_{n_i}$, and x is data then for any $n_j \in \mathbf{P}$ if n_j is nested by n_i then mark n_j as "data". (rule 6)
– Copy all assignment nodes marked as "data" with corresponding predicate nodes to build a new set of CFGs called *data slice*. (rule 7)
– Copy all assignment nodes marked as "control" with corresponding predicate nodes to build a new set of CFGs called *control slice*. (rule 8)
– Create a new entity $\mathcal{E}_d$ for data slice with input port $\mathcal{D}_{in} \cup c_{in}$ where $c_{in} \subseteq \mathcal{C}_{in}$ are control inputs being used in data slice. (rule 8)
– For each signal s in data slice, such that s is marked as "control" and $s \notin c_{in}$, create a new input port for s_i in data slice and corresponding output port s_o in control slice. (rule 7)
– Create a new entity $\mathcal{E}_c$ for control slice. (rule 9)

Algorithm 1 recursively sets internal signals as data until outputs are reached. Once a subset of outputs of the module is marked as data, and there are no more iterations in which further signals or outputs are being marked, we reach a fixed point where dependency analysis is finished. The slicing algorithm is implemented as recursive functions with the help of depth first search (DFS) algorithm, and algorithms due to Lengauer and Tarjan [69] to compute paths and dominators, in CFGs and GFD.

The slicing algorithm is followed by some restructuring. The new entities for control and data slice are created with additional intermediate I/O ports. To drive the new ports, additional processes are created in relevant slices which read corresponding signals and update these ports. In case of process splitting during slicing, statements are moved between newly created processes in control and data slice. This causes new signals being read inside the processes. Therefore, sensitivity list of new processes in the two slices is also revised according to the signals being read in the new processes.

3.4 Illustration of control-data slicing

In this section we will apply the proposed slicing algorithm to a VHDL module. We have considered an integer accumulator circuit as case study.



(a) Block diagram

(b) VHDL description

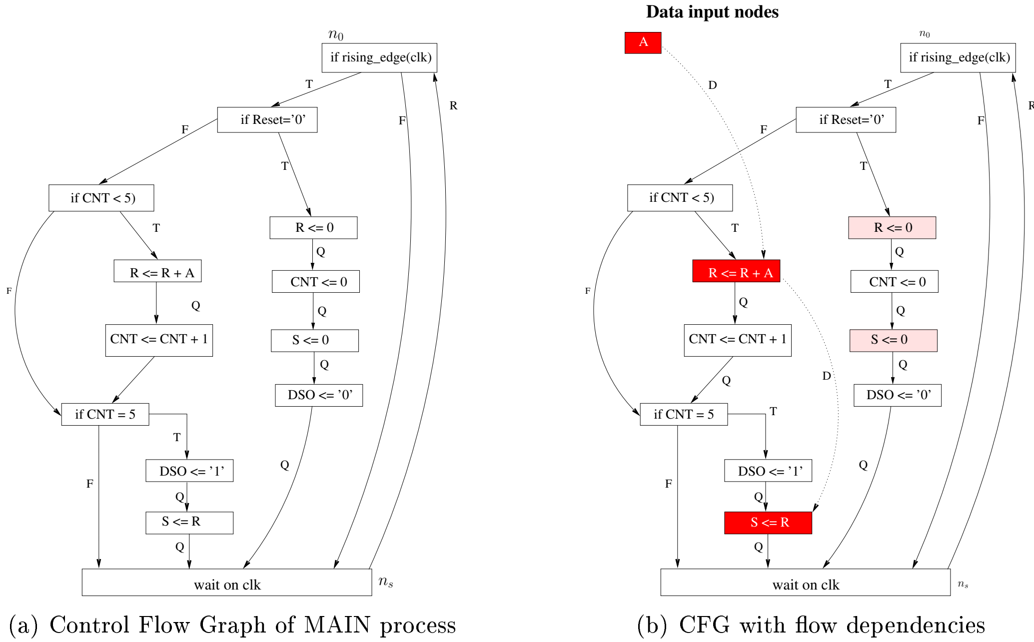
Figure 3.5: Accumulator example

The block diagram of accumulator is shown in Figure 3.5(a). It accumulates 5 samples at its input '`A`' in an internal register '`R`', and writes the output to an output register '`S`' asserting the output signal '`DSO`'. An input '`Reset`' clears the internal registers. The RTL description of accumulator in VHDL is given in Figure 3.5(b). We will suppose that '`A`' is labeled as *data* by the designer.

The VHDL description of accumulator is first parsed to an abstract syntax tree, and processes are transformed to the CFGs. In Figure 3.6(a), we have shown CFG of `MAIN` process obtained according to definition 3.4. Each rectangular block is a CFG node with corresponding VHDL statement. Labels n_0 and n_s represent entry node and suspension node respectively.

Figure 3.6(b) shows flow dependence graph obtained from CFG of Figure 3.6(a) by applying dependency analysis rules (rules 2 to 6) of the slicing algorithm. Flow dependency marked as dotted edges and labeled as D, represents the data dependency from input node A towards node `R <= R+A;`, and is called *direct dependence*. There is a flow dependence from node `R <= R+A;` to node `S<=R;` because `R` being written at node `R <= R+A;` is used at node `S<=R;`. This dependence is marked in subsequent iteration of the algorithm, and is called *indirect dependence*. The direct and indirect dependencies are highlighted in dark color. Nodes `R<=0;` and `S<=0;` highlighted in light color represent the assignments which apparently constitute the assigned signal by a constant as control signal but are considered as data assignments due to the fact that same target signal is assigned at some other node which makes it a data signal due to direct or indirect flow dependency on some data input as mentioned in rule 4 in subsection 3.3.2.

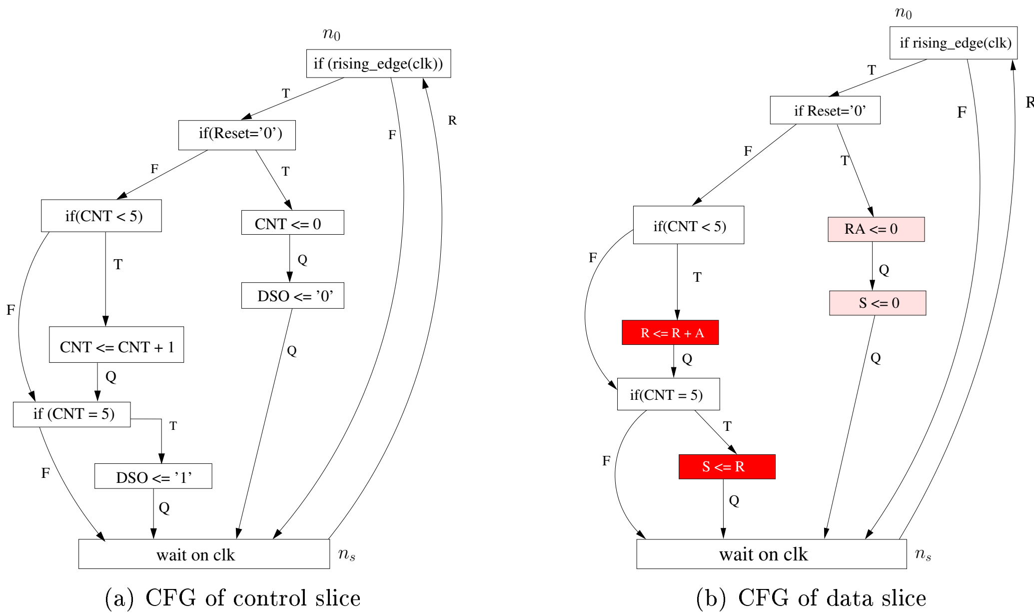
Once all the dependencies are marked by dependency analysis, we perform the partitioning step which generates two CFG of control and data slice. Figure 3.7 shows the result of the slicing. Assignment nodes marked as control are collected to form the CFG for control slice as shown in Figure 3.7(a). Nodes marked as data with accompanying control predicate nodes in GFD are collected together to form new CFG for the data slice as shown in Figure 3.7(b). The VHDL descriptions of control slice and data slice are shown in Figure 3.9(a) and 3.9(b)



(a) Control Flow Graph of MAIN process (b) CFG with flow dependencies

Figure 3.6: Control and data flows in accumulator

respectively.

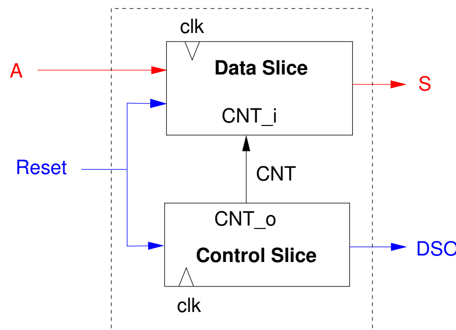


(a) CFG of control slice (b) CFG of data slice

Figure 3.7: CFG of control and data slices for accumulator

The creation of new control and data slice as VHDL entities undergoes some automatic restructuring after dependency analysis. For example while slicing accumulator, the value of

signal `CNT` signal must be communicated from control slice entity towards the data slice entity. Therefore, we need to create an additional output port named `CNT_o` at line 4 in control slice of Figure 3.9(a), and corresponding input port `CNT_i`, at line 5 in data slice entity of Figure 3.9(b). An additional combinational process (concurrent signal assignment statement) is needed in the control slice architecture to update the new port `CNT_o` at line 25 in Figure 3.9(a). This new process is created automatically in the form of a new CFG according to definition 3.4. Block diagram of the resulting control and data slices is shown in Figure 3.8(a)



(a) Control and data slice blocks

```
1.  entity accum is
2.     port (CLK, Reset:in bit;
3.           A:in integer;
4.           S: out integer;
5.           DSO: out bit);
6.  end entity;
7.  architecture RTL of accum is
8.     signal CNT: integer range 0 to 5;
9.  begin
--****************************************
10.  inst_data: entity work.accum_Data(RTL)
11.     port map(CLK, Reset, A, S, CNT);
------------------------------------------
12.  inst_cntrl: entity work.accum_Control(RTL)
13.     port map(CLK, Reset, DSO, CNT);
--****************************************
14.end architecture;
```

(b) Top level VHDL module for slices

Figure 3.8: Slicing results of accumulator

```
1.  entity accum_Control is
2.   port (CLK, Reset:in bit;
3.         DSO: out bit;
4.      CNT_o:out integer range 0 to 5);
5.   end entity;
--**********************************
6.  architecture RTL of accum_Control is
7.     signal CNT: integer range 0 to 5;
8.  begin
9.  proc_C_0: process(CLK)
10.  begin
11.    if (CLK'event and CLK='1') then
12.      if(Reset = '0') then
13.         CNT <= 0;
14.         DSO <= '0';
15.      else
16.         if (CNT < 5) then
17.            CNT <= CNT+1;
18.         end if;
19.         if(CNT = 5) then
20.            DSO <= '1';
21.         end if;
22.      end if;
23.    end if;
24.  end process;
------------------------
25.     CNT_o <= CNT;
26.  end architecture;
```

(a) Regenerated VHDL for control slice

```
1.  entity accum_Data is
2.     port (CLK, Reset:in bit;
3.           A:in integer;
4.           S: out integer;
5.       CNT_i:in integer range 0 to 5);
6.  end entity;
--**********************************
7.  architecture RTL of accum_Data is
8.    signal R: integer;
9.  begin
10.  proc_D_0: process(CLK)
11.   begin
12.    if (CLK'event and CLK='1') then
13.       if(Reset = '0') then
14.          R <= 0;
15.          S <= 0;
16.       else
17.          if (CNT_i < 5) then
18.             R <= R + A;
19.          end if;
20.          if(CNT_i = 5) then
21.             S <= R;
22.          end if;
23.       end if;
24.   end if;
25.  end process;
26.  end architecture;
```

(b) Regenerated VHDL for data slice

Figure 3.9: Accumulator slices in VHDL

A top level module is also generated automatically which instantiates, and connects both control and data slice entities with same external interface as that of original module. In

this way the simulation environment delivered for original module is reused to test the sliced model. The VHDL description of top level module for accumulator is shown in Figure 3.8(b). An intermediate signal `CNT` is declared at line 8 in the top level architecture to connect the control slice output signals with the data slice input signals.

3.5 Slicing modules with local variables

In slicing algorithm, we have only considered VHDL modules with signals. However, local variables are also frequently used in designs. VHDL variables are assigned and used only within the process in which they are declared. Unlike signals, during the execution of the process, a variable being assigned an expression immediately gets the current evaluated value of the expression.

The way in which variables are used varies from designer to designer. Mostly they are used for intermediate calculations within processes. However, in many designs variables are used in such way that sythesizers may infer memory elements (Flip-flops or Latches) from such variables. According to IEEE standard for RTL synthesis [58], a variable is inferred as an edge triggered memory element (a D flip-flop) when

- it is not being assigned in all the execution paths of the process,
- it is being assigned at one clock edge, and read at the subsequent clock edge.

Variables are inferred as combinational functions when they are being assigned in all execution paths of the process. If a variable is assigned at an execution point in the process, and this variable is used in the subsequent statements before reaching suspension statement, we say that variable is *written before read*. For this kind of variables, synthesizer infers combinational logic.

Since slicing partitions the design into two VHDL entities communicating via intermediate signals, we need to take care while splitting processes with local variables because there may exist situations, where some data assignment statement is dependent on a local variable labeled as control during slicing. In that case, we need to globalize the local variables via signals to communicate variables values between the slices.

In order to transmit the values of local variables to global entities of slices, we need to convert variables into signals. However, this conversion is not straightforward because variables and signals have different semantics in VHDL. The expressions assigned to variables immediately update the memory location reserved for variables however signals are not immediately updated but are scheduled to be assigned new expressions (waveforms) until the next suspension statement is reached: that is the next delta cycle [57] in case of synchronous modules. We will elaborate the problem of slicing with local variables by an example as propose the solution.

Accumulator with local variable

A new version of accumulator example of Figure 3.5 is implemented with `CNT` as local variable instead of a signal as shown by VHDL description of Figure 3.10. Compared to implementation of Figure 3.5(b), there is a slight modification of range of `CNT` in the original implementation due to different semantics of variables and signals in VHDL.

We can not directly apply the slicing rules in this implementation because after slicing, the value of local variable `CNT` marked as control variable in control slice has an impact on the data activities in the data slice. When statement `R <= R + A;` at line 21 is moved to data slice while `CNT` being control variable is updated in control slice, the execution of statement `R <= R + A;` in data slice depends on value of `CNT` as shown by statement `if (CNT < 5) then` at line 20. Variable `CNT` can not be converted directly into signals to convey local values to global signals.

```
1.  entity accum_v is
2.    port (CLK, Reset:in bit;
3.       A:in integer;  --DATA INPUT
4.       S:out integer;
5.       DSO:out bit);
6.  end entity;

7.  architecture RTL of accum_v is
8.    signal R: integer;

9.  begin
10. MAIN: process(CLK)
11.   variable CNT: integer range 0 to 6;
12.   begin
13.     if (CLK'event and CLK='1') then
14.       if(Reset = '0') then
15.          R   <= 0;
16.          CNT := 0;
17.          S   <= 0;
18.          DSO <= '0';
19.       else
20.         if (CNT < 5) then
21.            R <= R + A;
22.            CNT := CNT+1;
23.         else
24.            CNT := CNT+1;
25.         end if;
26.         if(CNT = 6) then
27.           DSO <= '1';
28.           S <= R;
29.         end if;
30.       end if;
31.     end if;
32.   end process;
33. end architecture;
```

Figure 3.10: VHDL RTL description of accumulator with variable

To overcome the problem of conveying values of local variables between slices via signals, we have proposed a ***VHDL transformation*** before slicing algorithm, and some enhancement afterwards. Thus Control-data slicing with local variables involves two additional steps. During first step, we transform the VHDL module internal structure. During second step, a syntax analysis called *enhancement* is applied to find the sequence of statements, where we need to convey value of a local variable between control slice and data slice entities.

3.5.1 VHDL transformation

For slicing modules with local variables, we perform a transformation before applying slicing algorithm. In this transformation, we split synchronous process with variables into a synchronous and a combinational part. Synchronous part represents only registers updates. It is a new synchronous process which only contains a set of register signal assignment statements. The combinational part represents the combinational logic of the process. All the transfer functions of registers are carried to a new combinational process sensitive to all the signals being read. Each variable inferred as register in the original process is represented by two additional signals after this transformation. In this way, we are capable of transmitting local variable value changes to the global entities via signals.

Basic idea The basic idea behind the transformation is as follows:

> *"In any sequential process with local variables, registers, either inferred from signals or variables, are only updated in a synchronous process. All the combinational logic functions assigned to such signals and variables at the rising edge of the clock are*

calculated out of this synchronous process, and are represented combinationally in separate combinational processes".

During transformation, the architecture $\mathcal{A}$ of the given module is converted into $\mathcal{A}'$. Each sequential process with local variables in $\mathcal{A}$ is splitted into a new synchronous process, a new combinational process, and a set of output processes for each output being assigned in original process. For a synchronous process $P \in \mathcal{P}$ with local variables with $\mathscr{V} \neq \emptyset$. Let

- $\mathcal{W}_P$ denotes set of signals written in process P
- $\mathcal{R}_P$ denotes set of signals being read in P
- REG_P denotes set of registers belonging to P

In synchronous process with local variables, we categorize three kind of registers as follows:

1. $REG_s \subset REG_P$ represents set of registers inferred from signals. Each $R_s \in REG_s$ is called *signal register*
2. $REG_v \subset REG_P$ represents set of registers inferred from local variables. Each $R_v \in REG_v$ is called *variable register*
3. $REG_{si} \subset REG_s$ represents set of registers inferred from interface signals. Each $R_{si} \in REG_{si}$ is called *signal interface register*

We describe the transformation algorithm as Step I, and the enhancement algorithm as Step II as follows.

3.5.2 Step I : Transformation algorithm

The pseudocode of the transformation procedure has been given in algorithm 2. It takes architecture $\mathcal{A}$ of the module's entity and returns a new architecture $\mathcal{A}'$. Operations are performed on CFGs of the synchronous processes with local variables. The new architecture contains new CFGs representing sequential, combinational and output processes generated according to the algorithm. The new architecture is then subjected to control-data slicing algorithm. Before slicing, the VHDL description of the transformed model can also be generated from the transformed CFGs for testing purpose.

The transformation algorithm does not impact the input/output ports of the module. It only restructures the internal processes in the architecture. In this way the transformed model can be tested by the same simulation environment provided with the original model.

Illustration of transformation

We will apply the proposed transformation algorithm to the accumulator of Figure 3.10. In this module, signals `R`, `DSO`, `S`, and variable `CNT` are registers. For each of these registers, new signals are declared in the architecture declarative part as shown in Figure 3.11 with corresponding VHDL comments for illustration.

Internal signal `R` is represented by new signal `R1`. Variable `CNT` is represented by two signals `CNT1` and `CNT2` as declared at line 3 in Figure 3.11. Output interfaces `DSO` and `S` are also replaced by signal pairs `DSO1`, `DSO2`; and `S1`, `S2` respectively.

The transformed body of the architecture is shown in Figure 3.12. The newly created processes in the new architecture are described below.

Algorithm 2 Pre-slicing Transformation Algorithm

Require: VHDL architecture $\mathcal{A}$
Ensure: Transformed VHDL architecture $\mathcal{A}'$
for (each synchronous process P with $\mathscr{V} \neq \emptyset$) **do**
– Identify *signal*, *variable*, and *signal interface* registers in P.
– Replace P by following:
for each signal register $R_s \in REG_s$ **do**
* create a new signal s_1
end for
for each variable register $R_v \in REG_v$ **do**
* create two new signals v_1 and v_2
end for
for each signal interface register $R_{si} \in REG_{si}$ **do**
* create two new signals si_1 and si_2
end for
– Create a new synchronous process P_s containing exactly, sequentially in any order: $\{R_s\texttt{<=}s_1; \mid R_s \in REG_s\} \cup \{v_1\texttt{<=}v_2; \mid R_v \in REG_v\} \cup \{si_1\texttt{<=}si_2; \mid R_{si} \in REG_{si}\}$
– Create a new combinational process P_c exactly sensitive to $\mathcal{R}_P \cup REG_s \cup \{v_1 \mid R_v \in REG_v\} \cup \{S_{si1} \mid R_{si} \in REG_{si}\}$ and with declared variables belonging to REG_v. Its body exactly contains following three parts sequentially in the given order:

- Sequentially in any order: $\{R_v := v_1; \mid R_v \in REG_v\} \cup \{s_1 <= R_s; \mid R_s \in REG_s\} \cup \{si_2\texttt{<=}si_1; \mid R_{si} \in REG_{si}\}$.
- The body of P where all assignment to R_s is replaced by assignment to s_1 and assignment to R_{si} is replaced by assignment to si_2.
- Sequentially in any order: $\{v_2\texttt{<=}R_v; \mid R_v \in REG_v\}$.

– Create an output process for each R_{si} sensitive to si_1 containing assignment $R_{si}\texttt{<=}si_1$;.
end for

New synchronous process: For `MAIN` process, we obtain a synchronous process labeled `Synch_p` at line 2 in Figure 3.12. It contains only register updates, and no combinational expression is assigned in the body. Register due to internal variable `CNT`, and signal interface registers `S` and `DSO` are represented by new signal registers `CNT1`, `DSO1` and `S1` respectively.

New combinational process: The new combinational process is labeled as `Comb_p` at line 12 in Figure 3.12. It has three parts indicated by line numbers as follows:

1. PART I starts from line 15 to 18, and contains assignments which read the associated registers
2. PART II starts from line 19 to line 35, and contains combinational part of original process with updated targets of signal assignments according to algorithm 2
3. PART III is at line 36 updates variable register input `CNT2`

New output processes: For ports `S` and `DSO`, the transformation algorithm creates two output processes labeled `Output_p_0` and `Output_p_1`, as shown at lines 39 and 43 respectively to update these output ports.

```
1. architecture RTL_transformed of accum_v is
2. signal R1:integer;                          --for signal R
3. signal CNT1,CNT2:integer range 0 to 6;      --for variable CNT
4. signal DSO1,DSO2:bit;                       --for signal interface DSO
5. signal S1,S2:integer;                       --for signal interface S
......
```

Figure 3.11: Accumulator architecture declarative part after transformation

```
1. begin
----------------------------------------
--****** Synchronous process *********
----------------------------------------
2.  Synch_p: process
3.  begin
4.   if ((clk = '1') and clk'EVENT) then
5.     R <= R1;         --signal
6.     S1 <= S2;   --interface signal
7.     DSO1 <= DSO2;--interface signal
8.     CNT1 <= CNT2;--variable
9.   end if;
10.  wait on clk;
11. end process;
----------------------------------------
--****** Combinational process *********
----------------------------------------
12. Comb_p: process
13.  variable CNT:natural range 0 to 10;
14. begin
      -- *** PART - I ***
15.   R1 <= R;
16.   S2 <= S1;
17.   DSO2 <= DSO1;
18.   CNT := CNT1;               -- Node n_1
      -- *** PART - II ***
19.   if(Reset = '0') then
20.      R1  <= 0;
21.      CNT := 0;
22.      S2  <= 0;
23.      DSO2 <= '0';
24    else
25.         if (CNT < 5) then       -- Node n_2
26.            R1 <= R + A;         -- Node n_3
27.            CNT := CNT + 1;
28.      else
29.            CNT := CNT + 1;
30.      end if;
31.      if(CNT = 6) then
32.           DSO2 <= '1';
33.           S2 <= R;
34.      end if;
35.   end if;
      -- *** PART - III ***
36.   CNT2 <= CNT;
37.   wait on CNT1,R,DSO1,S1,A,Reset;
38. end process;
----------------------------------------
--******* Output processes *********
----------------------------------------
39. Output_p_0: process(DSO1)
40.     begin
41.        DSO <= DSO1;
42.     end process;
        ----------
43. Output_p_1: process(S1)
44.     begin
45.        S <= S1;
46.     end process;
47. end architecture;
```

Figure 3.12: Accumulator architecture body after transformation

Four new processes obtained after this transformation are functionally equivalent to the `MAIN` process of the original module at clock cycle level. The entity declaration remains unchanged, however architecture declaration contains additional signals, and the body of architecture contains new processes after transformation.

3.5.3 Step II: Enhancement algorithm

Due to VHDL transformation proposed in Step I, we split the synchronous processes with local variables into interacting combinational and sequential parts. Applying slicing rules on splitted representation also requires some structural analysis between dependency analysis, and partitioning steps of the slicing algorithm.

According to slicing algorithm, data nodes (data assignment statements) are moved to data slice along with corresponding predicate nodes (control statements). During dependency analysis, we observe that if a node assigning local variable marked as “control” in new combinational process postdominates a node marked as “data”, we would need to convey the value of variable marked as “control” towards the node marked as “data” in the data slice. This is possible by assigning the updated value of the control variable in the combinational process of control slice to a global signal which communicates with the corresponding combinational

process in the data slice.

For this purpose, we need to determine those sequences of statements, in which a variable being written at a point in the process body is read such that a data statement is under the control of the read variable. For each sequence of statements of this form, the updated (written) value of variable is assigned to a new signal which carries the updated variable value towards the data slice. The slicing algorithm needs to be enhanced with this mechanism while dealing with variables. An enhancement algorithm is thus proposed for the restructuring which is applied after we have marked all the nodes as control and data according to dependency analysis rules, and before splitting the module by partitioning rules.

After applying dependency analysis rules of the algorithm 1, we mark the CFG which contains "control" as well as "data" nodes, as a *mixed CFG*. During enhancement procedure, we traverse all the mixed CFGs to search for an assignment node marked as "data", which is nested by a predicate node marked as "control", such that corresponding USE set contains a variable marked as "control". If such sequence is detected in the combinational process, then we add a signal assignment node as successors to the variable assignment node which is dominating the predicate node. The new assignment node copies the value of the variable to a new signal. This signal transmits the updated variable value to the other CFG of the process in the data slice by applying partitioning rules of the algorithm 1.

The pseudocode of the enhancement function is given as algorithm 3. This function is only applied to mixed CFGs. In case of insertion of a signal assignment statement in the process,

Algorithm 3 Slicing enhancement function

void enhancement() {
for each $CFG = (N, E)$ with corresponding process $P = (\mathcal{L}, \mathscr{V}, \mathscr{S})$ such that $\mathscr{V} \neq \emptyset$ **do**
  if $(\exists v \in \mathscr{V}, n_1 \in N$ such that $DEF_{n_1} = \{v\}) \wedge (\exists n_2 \in N$ such that $v \in USE_{n_2}$ and n_2 postdominates $n_1)$ **then**
    if (n_2 is control) $\wedge$ ($\exists n_3 \in N$ such that n_3 is data) **then**
      - Create a new output interface signal s_o for control slice entity $\mathcal{E}_c$
      - Insert a new signal assignment node with statement of the form, $s_o <= v$; as a successor of n_1.
      if (n_1 is nested by any $n \in N$) **then**
        * Insert a new signal assignment node of the form $s_o <= v$; as n_0 at the start of CFG
      end if
      - Create a new input interface signal s_i for data slice entity $\mathcal{E}_D$
      - Replace v in n_2 by s_i in new CFG of $\mathcal{E}_D$
      - Add s_i in the sensitivity list of the corresponding data slice process
    end if
  end if
end for
}

an additional assignment to the same signal is also inserted at the start of the process to avoid unnecessary latch inference after synthesis. The algorithm 3 involves the CFG traversals. We have used the depth first search (DFS) algorithm to detect the USE and DEF sequences of local variables mentioned in algorithm.

Illustration of enhancement

We will consider our running example of accumulator with local variable to illustrate the enhancement procedure. The input module to the slicing algorithm consists of four VHDL processes obtained after pre-slicing transformation as given in Figure 3.12.

Since algorithm is only needed to be applied for mixed combinational process, therefore in this case the process labeled `Comb_p` at line 12 fulfills this condition. By applying enhancement algorithm to this process, we find that variable `CNT` is being written at line 18, and read at line 25. We mark variable assignment at line 18 as node n_1, and that at line 25 as node n_2. Since there exists a data assignment node n_3 at line 26 is nested by n_2 due to `R` being a data register, therefore value written to `CNT` at statement n_1 must be communicated to the data slice via signal.

Thus we create a new output interface signal `CNT_1`, and schedule an insertion of assignment of variable `CNT` to signal `CNT_1` just after the assignment to variable `CNT`. This assignment is shown at line 7 in the corresponding process coded in Figure 3.13.

Correspondingly in data slice, a new input interface signal `CNT_1` is scheduled to be created in the port list of the data entity. After slicing, the statement at line 25 of Figure 3.12, which is scheduled to be moved to the data slice, the algorithm replaces variable `CNT` by the new input signal interface `CNT_1`.

Hence the statement in data slice appears, as shown at line 9 in the corresponding data slice process in Figure 3.14. This new input interface signal `CNT_1` is also added in the sensitivity list of the process in the data slice containing above statement at line 16 in Figure 3.14.

```
1.  Control_Process_2: process
2.     variable CNT : integer range 0 to 6;
3.  begin
4.      CNT_i_2 <= CNT;          -- signal interface assignment to avoid latch
5.      DSO2 <= DSO1
6.      CNT := CNT1;
7.      CNT_i_1 <= CNT;          -- newly inserted signal interface assignment
8.      if (Reset = '0') then
9.           DSO2 <= '0';
10.          CNT := 0;
11      else
12.         if (CNT < 5) then
13.          CNT := CNT + 1;
14.         else
15.          CNT := CNT + 1;
16.         end if;
17.          CNT_i_2 <= CNT; -- newly inserted signal interface assignment
18.         if (CNT = 6) then
19.           DSO2 <= '1';
20.         end if;
21.     end if;
22.     CNT2 <= CNT;
23.     wait on CNT1,DSO1, Reset;
24. end process;
```

Figure 3.13: Combinational process in control slice after enhancement

Similarly, referring to the same process `Comb_p` in Figure 3.12, variable `CNT` is also being written at line 27 and read at line 31, and there exists an underlying data operation at line 33. Therefore, a new interface signal assignment statement is inserted as shown in Figure 3.13 at line 17 after `CNT := CNT + 1;`. Correspondingly in the data slice process in Figure 3.14, a new input interface signal `CNT_2` (new port) is inserted, and variable `CNT` is replaced by signal `CNT_2` as shown at line 12. Signal `CNT_2` is also added in the sensitivity list of the corresponding process in data slice, as shown at line 16 of Figure 3.14.

```
1. Data_Process_2: process
2. BEGIN
3.     R1 <= R;
4.     S2 <= S1;
   --*****************
5.     if (Reset = '0') then
6.         R1 <=  0;
7.         S2 <=   0;
8.     else
9.         if (CNT_1 < 5) then     -- CNT is substituted with CNT_1
10.            R1 <= R + A;
11.        end if;
12.        if (CNT_2 = 6) then   -- CNT is substituted with CNT_2
13.            S2 <= R;
14.        end if;
15.        end if;
16.     wait on R,S1,Reset,A,CNT_1,CNT_2;  -- 2 additional signals in sensitivity list
                                          -- due to enhancement procedure
17. end process;
```

Figure 3.14: Combinational process in data slice after enhancement

To avoid any latch inference and preserve the synthesis semantics, the algorithm puts additional assignment to the signal `CNT_2` at the start of the combinational process as shown at line 4 of Figure 3.13.

```
1. entity accum_v_Control is
2.  PORT (CLK: IN bit;
3.      Reset: IN bit;
4.      DSO: OUT bit;
5.      CNT_1: OUT natural range 0 to 6;
6.      CNT_2: OUT natural range 0 to 6);
7. end entity;
```

(a) Control Slice after enhancement

```
1. entity accum_v_Data is
2.  PORT (A:IN integer;
3.       S:OUT integer;
4.       CLK:IN bit;
5.       Reset:IN bit;
6.       CNT_1:IN natural range 0 to 6;
7.       CNT_2:IN natural range 0 to 6);
8. end entity;
```

(b) Data slice after enhancement

Figure 3.15: Accumulator: entities of slices after enhancement

Note that variable `CNT` being written at node corresponding to statement `CNT := 0;` at line 21 of Figure 3.12, need not be communicated via signal because it is not being read by the subsequent statement in the same delta cycle during process execution.

After enhanced slicing, we obtain control and data entities in VHDL as shown Figure 3.15 with two additional ports `CNT_1` and `CNT_2` for communication between slices. The block diagram of enhanced slicing result are depicted in Figure 3.16

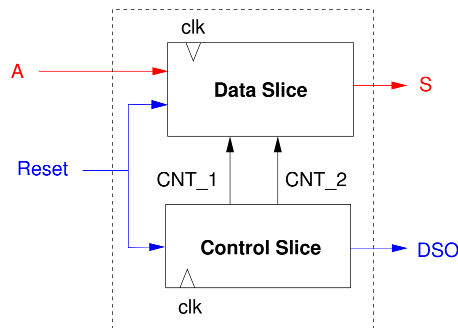


Figure 3.16: Slicing result of accumulator with variable

3.5.4 Impact of VHDL transformation

The pre-slicing transformation allows to manipulate combinational activities separated from the *clock* at delta cycle levels. In this way, we are able to globalize the local variables effect within the delta executions of combinational process. With transformation and enhancement, our control-data slicing paradigm is modified, and looks as shown in Figure 3.17. The processes with local variables are first transformed and then sliced with enhancement algorithm used where applicable.

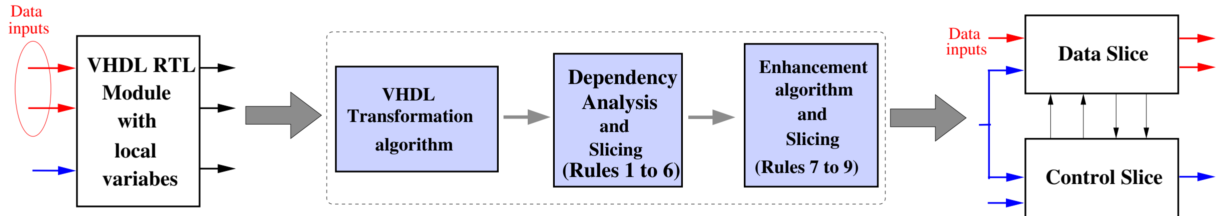


Figure 3.17: Control-data slicing with local variables

It has been tested by exhaustive simulations that proposed transformation preserves the functionality of the original module before and after synthesis. The entity declaration remains unchanged. The architecture declaration contains additional signals, and the body of architecture contains new processes after transformation.

A drawback of this transformation may be a negligible degradation in simulation speed due to additional processes and signals introduced in transformed module as compared to original one.

3.6 Implementation of control-data slicing

For implementation, we have chosen the VHDL front-end analysis tool SAVANT [94], [107] which implements internal intermediate representation (IIR) of VHDL language according to standard Advanced Intermediate Representation with Extensibility (AIRE) [5]. SAVANT contains a VHDL front end analyzer that parses, type-checks, and converts VHDL files into an object-oriented intermediate representation based on the AIRE standard. SAVANT is constructed using PCCTS (Purdue Compiler Construction Tool Set) parser generator [2] written in C++.

The IIR is a well connected abstract syntax tree (AST). The complete VHDL file after parsing is stored as IIR nodes which can be accessed through pointers. There are various application programming interface (API) functions provided for different manipulations on the IIR. These functions are readily used to determine DEF/USE sets, hierarchy traversals, VHDL code regeneration, and other operations required in the control-data slicing algorithm. In our implementation, we perform manipulations on IIR of given VHDL file, and build our data structures (CFG and GFD) for slicing algorithm and regenerate IIR using API functions. Use of SAVANT in implementing slicing algorithm not only avoided implementing VHDL frontend parser, but also simplified the implementation by reuse of existing API functions.

3.7 Applications of control-data slicing

In this section, we describe practical implications of control and data separation using slicing algorithm. We describe how sliced model can be used for model checking a class of properties with possibility of data abstraction. We exploit the sliced representation to raise the abstraction level of RTL model for faster simulations. We also mention some applications of slicing in various steps during computer-aided design (CAD) process.

3.7.1 Assisting model checking by slicing

One of our objectives to separate control and data was to model-check those parts of the IP module containing its temporal behavior which comes out to be in the form of *control slice* by the slicing algorithm. The data independent control slice is aimed at deploying for model checking. For many applications such as multimedia and digital signal processing, the control slice is much simpler than original module because it does not contain actual data processing but the timing information of data flows. Many interesting safety and liveness properties are concerned only with the control slice. Therefore, in modules with data independent control, we can only subject the control slice for model checking. In this way we achieve an abstraction of the behavior of the model.

Model checking experiments

We have conducted a few experiments to observe the impact of our control-data slicing algorithm on model checking time in IP modules. Some RTL models and their control slices have been tested for the same property. We observed improvements in overall property verification time comprising of the model elaboration time, initialization elapse, and the state space exploration time. Experiments have been conducted for four different modules: serial parallel multiplier (SPM), a serial transmitter (UART Tx), DCT input registers stage (DCT IR), and data encryption standard (DES) module. Results are shown in Table 3.1. Properties verified for these modules are described in natural language and corresponding CTL specifications as below.

P_1 = "THE DSO EVENTUALLY BECOMES ACTIVE"

$\boxed{EF(DSO = 1)}$ where DSO is an output control signal indicating presence of valid data on output.

P_2 = "IN ALL EXECUTION PATHS, IF *reset* IS HIGH IN THE CURRENT CLOCK CYCLE THEN INTERNAL BUFFER AND REGISTER ARE CLEARED IN THE NEXT CLOCK CYCLE"

$\boxed{AG(reset \Rightarrow AX(bufe \wedge rege))}$ where $reset$ is a control input. $bufe$ and $rege$ are boolean output control signals of a UART module which become high when internal buffer and register are empty.

P_3 = "AN INPUT IS ALWAYS ACKNOWLEDGED AT THE SAME TIME WHEN AN OUTPUT IS BEING REQUESTED"

$\boxed{AG(iack \wedge oreq)}$ where $iack$ and $oreq$ are the '*input acknowledge*' and '*output request*' signals respectively.

$\mathtt{P}_4$ = "UNDER A CONDITION THAT *reset* SIGNAL IS ALWAYS KEPT LOW, IF *DSI* IS HIGH THEN, *DSO* WILL ALWAYS BECOME ACTIVE AFTER 30 CLOCK CYCLES PROVIDED *DSI* IS NOT REACTIVATED THROUGHOUT THE 30 CYCLES."

Depending upon the size of the control slice, there is considerable improvement in elaboration and model checking time while verifying above properties for control slice as compared to original module given for verification. The benchmark examples are data oriented applications in which all the data processing is abstracted away resulting a compact control slice model. These experiments have been conducted with VIS model checker [3], [19].

Module: Property		**Comb. blocks**	**No. of registers**	**Verification time (sec)**	**Improvement**
SPM: $\mathtt{P}_1$	Original module	157	53	0.429	78.7 %
	Control slice	14	5	0.091	
UART Tx: $\mathtt{P}_2$	Original module	41	23	0.226	49.5 %
	Control slice	22	6	0.114	
DCT IR: $\mathtt{P}_3$	Original module	369	170	1.82	91.4 %
	Control slice	37	12	0.155	
DES: $\mathtt{P}_1$	Original module	1363	128	15.48	97.9 %
	Control slice	28	8	0.314	
SQRT: $\mathtt{P}_4$	Original module	2561	387	24.05	99.2 %
	Control slice	39	15	0.188	

Table 3.1: Model checking results with and without slicing

Properties involving primary inputs are not directly possible to be model checked because variables involved in the property specifications must be registers (states). Therefore, for verifying such properties additional states are needed for inputs which are automatically generated in the verification process by instantiating the module under test in a top level module, and injecting inputs via registers.

Use of monitors

Properties similar to $\mathtt{P}_4$ are sometimes tedious to express in temporal logic by engineers less familiar with the formal methods. Since temporal logic formalas are equivalent to state machines [42]. Therefore, $\mathtt{P}_4$ is implemented as a state machine representing property violation monitor. Monitors are similar to the 'observers' proposed by N. Halbwachs [46] and G. Berry [11]. A monitor for property $\mathtt{P}_4$ is implemented which drives an *error* signal. It becomes high whenever the property is violated. The state of the *error* signal is model-checked by CTL invariant `AG(!error=1)`.

Monitors could easily be implemented by engineers verifying hardwares by simulation test benches because they have better understandability of hardware behaviors in terms of finite state machines rather than LTL or CTL. Temporal logic specification can also be automatically converted to equivalent state machines by using algorithms proposed in [42], [40], [108]. For automating the process of writing state machine for such specifications, techniques proposed by IBM Research Inc. [6], [10] and D. Borrione et al. [16] are also useful.

We aimed at making the basic model checking easy to be used in the verification process. A model checking mechanism is thus envisaged as shown in Figure 3.18 utilizing the control data slicing for data abstraction at first place, and the existing formalisms to specify properties as FSM monitors to facilitate model checking based verification for engineers.

Control slice extraction and creation of additional states for inputs are automatic in the current implementation. Automating the implementation of monitors and integration of specification formalisms could be a future work.

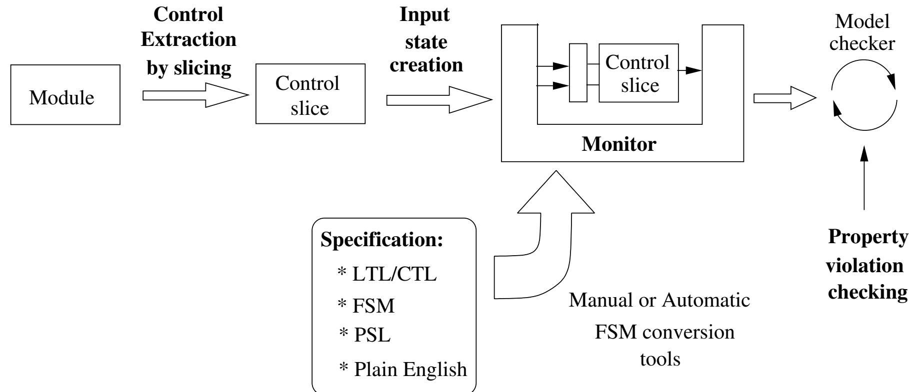


Figure 3.18: Assisting model checking by slicing and monitors

3.7.2 Assisting simulation by slicing

One of our main objectives of slicing was to increase simulation speed by abstracting the complex computations in RTL data slice while keeping the timings of the computations (isolated as control slice) intact. We will describe an improvement in simulation based validation by replacing data processing within data slice, by a *high level computation function* which calculates the results faster as compared to concrete data processing in original model.

The function is assumed to be provided by the designer which may be a high level functional VHDL model or may come from the functional specifications in C/C++ language. The concrete data processing is replaced by such function to reduce the simulation time. The approach is tested with a few examples and results have been shown according to classification of different hardware modules.

Basic idea

We intend to replace the RTL description of the data slice by a high level faster simulation model with same behavior at external interfaces. Such models are also called bus cycle accurate (BCA) models [91], [39]. Our envisaged data abstraction from RTL to high level is aimed at a class of modules which take input samples from the environment, process them by consuming some clock cycles and write the output results to the environment. Such data processing may be represented by a data flow shown in Figure 3.19 in which data slice is partitioned into following three phases.

1. Data acquisition

2. Data processing

3. Output phases

During "acquisition phase", data values are read from the environment via data inputs, and are stored in internal registers. In original model, this phase may take several clock cycles depending upon the specific design of the module. The acquisition phase is followed by the 'data processing phase' which implements the core functionality of the module. It is time consuming phase which may take several clock cycles for a processing. Once all data is processed, results are written to the data output where environment can read them. This is called 'output phase' which may take place in a single cycle, or may take more than one clock cycles depending upon the specifications of the design. All these phases could be controlled by the primary control inputs from the environment and/or by the control signals coming from the control slice as shown in Figure 3.19.

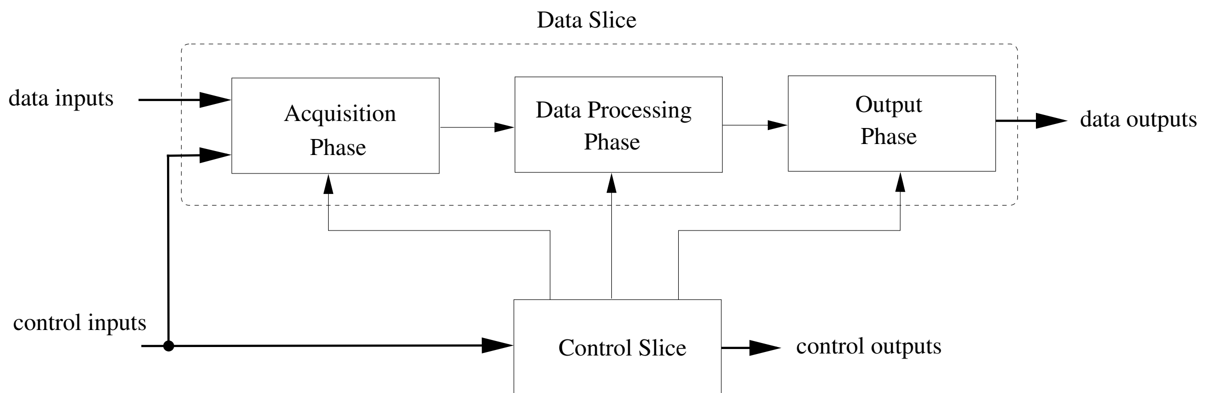


Figure 3.19: Data processing phases in data slice

By representing the data slice in this manner, we can replace the data processing phase by an equivalent but faster computation function which calculates the results in a functional way instead of cycle by cycle computation with RTL operators. This function takes the inputs via parameters, performs the data computations in zero time, and returns the result. This would lead us to abstractions analogous to BCA models [91] which simulate faster as compared to RTL models.

The replacement of data processing by an abstract data computation function also leads us to a model in which some internal control signals coming from control slice driving the data path are no more needed. Hence the control states producing control signals could be abstracted away. In this way control state machine could also be potentially minimized, and for certain applications where control state space is not explorable would become explorable. Thus we can model-check control state machine with less risk of state explosion.

Practical examples for which functional data slice model can be applied, are the data intensive IP components in communications and multimedia processing applications such as fast Fourier transform (FFT), discrete cosine transform (DCT), data encryption standard (DES), digital modulators/demodulator, and other arithmetic cores. The abstraction of data slice by high level function does not deal with modules with data dependent control such as variable length decoder (VLD), greatest common divisor, and UART receiver because the timing of the data operations in these modules depends on the data values.

Implementation aspects

For realization of functional data slice abstraction, one way is to propose a sophisticated syntactic and semantic analysis algorithm to extract the high level computation function automatically from the VHDL representation of data slice. Without design hints, this job would be non-trivial because the algorithm would heavily depend on VHDL syntax and require rich static semantic analysis.

Another alternative is to simulate the given model with test patterns and observe the output to guess different computation functions implemented in the data slice, and replace them with equivalent faster implementations at higher level. Such replacement requires the timing information of the data inputs and outputs at the interface, and the latency of the processing to be independent of the replacement so that behavior at the input and output buses would conform to the original model. This also requires pattern recognition routines in the RTL descriptions to detect the boundaries of data processing phase to be replaced with abstract function without loss of timing information.

An easy implementation could be possible by allowing some designer interventions. If the designer provides the un-timed functional model of the IP component, we can identify the effective data processing in data slice with relevant inputs/outputs interfaces, and replace it with the provided high level functional model. For such an implementation, we define some rules to identify the three phases in the data slice. This requires additional information from the designer about the primary control inputs. For instance, an input is needed to easily discover the data acquisition phase. Similarly, if the output strobing signal is given, we can discover the output phase of the data slice. The boundaries of remaining data processing phase could be identified by defining some syntactic pattern matching rules.

In our implementation, we suppose that following information is taken from the designer

- The *system reset* input
- A "*data strobe input*" signal to indicate when the valid data inputs are sampled. Similarly, a signal named "*data strobe output*" to indicate when the valid data output is available
- For each module, a single *data computation function* is provided with compatible input and output interfaces

Since we consider all data computations being carried out in data processing phase, the acquisition phase contains only a set of assignments from data input ports to internal registers. The results are written to data outputs after a zero time calculation and become available immediately at outputs, but they are considered valid when indicated by the *data strobe output* signal.

The structure of data slice shown in Figure 3.19, can be represented by a VHDL process and named as *functional data slice template*. A functional data slice template for single data input and single data output is shown in Figure 3.20. The three phases are indicated by the VHDL comments. The interface signal `DSI` at line 3 is the data strobe input signal. `D_IN` and `D_OUT` are the set of primary data inputs and primary data outputs respectively. `Reset` is the system reset signal. Architecture of abstract data slice contains single process sensitive to the clock signal.

Variables are declared to hold the data input samples and the output results. A variable `REG` at line 12 holds the input samples to be used in the computation. The acquisition phase is

```
1. entity Funct_Data_Slice is
2.     port ( clk: in bit;
3.            DSI: in bit;
4.            Reset: in bit;
5.            D_IN : in bit_vector;
6.            D_OUT: out bit_vector);
7. end entity;

8. architecture abstract_arc of Funct_Data_Slice is
9      signal RESULT;
10. begin
11.   process (clk)
12.       variable REG ;
13.   begin
14.     if (rising_edge(clk)) then
15.         if (Reset is active) then
16.             REG := 0;                  -- ***** Reset state *****
17.         elsif (DSI is active) then
18.             REG := D_IN;                -- ***** Acquisition phase *****
19.         else
20.             RESULT <= f(REG);       -- ***** Data processing phase *****
21.         end if;
22.     end if;
23.   end process;
24    D_OUT  <= RESULT;    -- ***** Output phase *****
25. end architecture;
```

Figure 3.20: Functional data slice template

shown at lines 17 and 18. The `Reset` signal clears the internal registers as given at lines 15 and 16. The data processing phase is simply a call to function `f` as shown at line 20. This function may be a VHDL function or it may be a foreign function in an other language like C/C++. It takes `REG` as inputs and calculates the result in zero time, and returns output to `RESULT`. In case of a function in foreign language, a VHDL procedure with `foreign` attribute specification is to be provided as mentioned in VHDL foreign language interface (FLI) specifications [80]. This function or procedure is used to communicate between VHDL and the foreign subprogram. In output phase, the calculated result is assigned to output ports as shown at line 21.

The template of Figure 3.20 can be automatically generated from the information provided by the user without performing syntactic analysis on the data slice of the module, and it is purely dependent on the information provided by the designer. The number of data inputs and outputs of the template could vary according to the modules specifications. For example in a multiplication circuit, the template can be generated with two data inputs labeled as the `multiplier` and the `multiplicand`.

Illustration of abstraction by example

We illustrate the idea of functional data slice modeling by an example. Consider an implementation of an 8-bit serial parallel multiplier (SPM) as shown on left side of Figure 3.21. Equivalent VHDL description is also given in Figure B.1 in Appendix B. The circuit samples two 8-bit data words at inputs `A` and `B` when `Load` input is high, performs multiplication in 8 clock cycles and gives the result at output `S` at 9th clock cycle. Output `DSO` indicates that valid result of multiplication is ready at 16-bit output `S`. Input vectors `A` and `B` are labeled as data inputs by the designer. `Load` and `Reset` are control inputs. Automatic slicing of SPM model results the control and data slice entities connected via intermediate signals as shown in block diagram of Figure 3.21. Here `cnt1` and `cnt2` are the intermediate signals carrying timing information for data operations inside the data slice.

We suppose that the designer gives us the supplementary information, that `Load` is a data strobe input signal, and `Reset` is the reset signal. Similarly, if the designer identifies `DSO` as

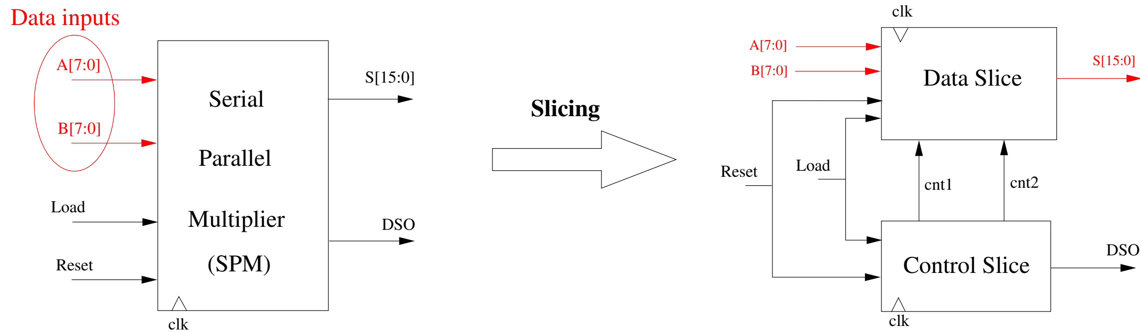


Figure 3.21: SPM after slicing

data strobe output, we can identify the "output phase". The "data processing phase" starts from line 18 to 23 in Figure B.2 which is a cycle by cycle operations (shift and add) to calculate the multiplication.

We have a zero time multiplication function provided in C language from functional specifications of the model. We replace the RTL shift, and add operations (data processing) from line 18 to 23 in SPM description of Figure B.2 by a single call to this multiplication function. VHDL simulators such as *ModelSim*® provides Foreign Language Interface (FLI) [80] to call foreign language routines in VHDL. For instance, to call a foreign C subprogram in ModelSim, we write a VHDL subprogram declaration and use `FOREIGN` attribute of VHDL.

The idea of integrating foreign C function calls in VHDL is interesting for faster simulations. As in most of the design specifications, we have un-timed functional models written in high level languages such as C or C++ earlier in the design cycle. Therefore, those models could be reused for such simulation, which are faster than the concrete RTL descriptions.

For SPM example, the C routine to calculate faster multiplication is given in appendix B on Figure B.5. The subprogram required to call C routines from the VHDL is given in Figure B.3. Each C function to be called in VHDL, is declared in a global package with `FOREIGN` attribute specified as on line 3 of Figure B.3. This declaration of subprogram is called in VHDL description, where we replace the data processing with a call to C routine, and return the result without consuming clock cycles. In Figure B.4 we have shown the replacement of original data computation by a call to zero time multiplication function `fast_spm` at line 16. Type casting is required between VHDL and C data types according to rules mentioned in FLI reference manual [80]. Note that internal registers `RA`, `RB`, and `RR` are no more needed in computation.

It is also notable that internal control signal `cnt1` coming from the control slice in the original model is no more used in the abstract model. Therefore, we can remove this signal and its corresponding states in the control slice. In this way, we can achieve a minimization of states in control slice. For large control and data slices where state space might be very large and unexplorable, would become explorable by this abstraction. The abstracted IP model in the form of block diagram looks as shown in Figure 3.22 after abstraction.

We have described the informal steps in transforming RTL description of a model into an abstract model based on the information provided by the designer about the timing signals at inputs and outputs. Some syntax analysis rules could be defined in the future research to make this transformation automatic.

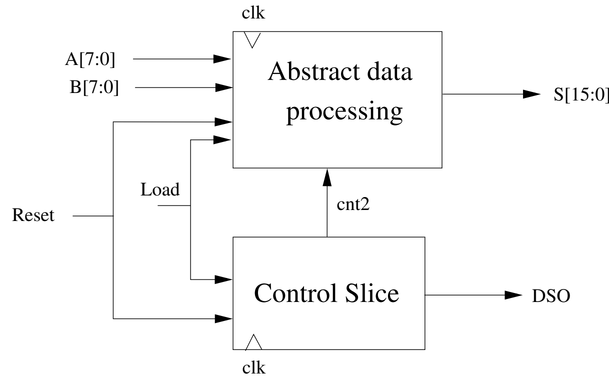


Figure 3.22: SPM after data abstraction

	Model	Simulation time (sec)	Improvement
SPM	Concrete	37.180	38.1 %
	Abstracted	23.004	
DIST	Concrete	23.638	75.6 %
	Abstracted	5.752	
DES	Concrete	119.077	13 %
	Abstracted	103.510	
SQRT	Concrete	11.85	34.6 %
	Abstracted	7.74	
FDIV	Concrete	55.69	29.1 %
	Abstracted	39.53	

Table 3.2: Simulation Results

Experimental results

We have tested the SPM example by simulation, and compared the time consumed by the simulation run in concrete and abstracted models. Various tests were observed on different examples such as serial parallel multiplication (SPM), distance calculation between two complex numbers (DIST), data encryption standard (DES), integer square root (SQRT), and floating point division (FDIV) modules The results of tests are provided in Table 3.2. With same number of random test vectors provided for concrete and abstract models under same conditions, we observe that the abstract model saves around one third of the simulation time for data computation intensive applications.

Classification of IP modules

The simplified functional abstraction of the data slice discussed uptil now, is applicable to a class of designs which take input samples from the primary inputs once, and internally process them by consuming some cycles. During processing, no new inputs are taken into account from the environment. Once all the data processing is completed, results are written to the primary outputs in single cycle. Such designs are called as *single phase designs* as indicated by Fummi et al. in [14].

There exist more general class of modules named *multi phase designs*. In multiphase

designs, data processing is performed in several steps. After each step partial inputs might be read from the environment. Similarly, after each data processing step, partial results might also be available at the outputs. This classification of hardware designs is intuitive, and has also been considered by Fummi et al. [14] while raising the abstraction level of RTL designs to TLM [1]. We have tested only single phase designs for demonstration of effectiveness of the approach. However, the technique can of course be enriched in the future to deal with multi phase designs as well.

3.7.3 Miscellaneous applications of Control Data separation

Besides assisting model checking and simulation based verification, our control-data slicing could also be helpful in computer-aided design (CAD) applications. As control and data have significantly different properties, many CAD tools tend to deal with them differently, as revealed in power estimation research by D.I. Cheng et al. [24]. If a separation of control and datapath is available, power estimation can be automated.

The proposed control-data slicing is also helpful for faster and more accurate area estimation earlier in the design cycle as indicated by V.J. Lam et al. [68]. Since control and data have distinct characteristics, therefore use of different tools is considered during various design processes. Logic synthesis, and automatic place and route tools are used to attain high quality implementation of the control slice while regular nature of data slice makes them better suited to be implemented manually or by datapath compilers.

FSM optimization techniques [62] are widely used in logic synthesis tools. If the control state machines in HDL codes are separated, optimization can be performed automatically, giving greater convenience and better synthesis results.

A control-data separation has also recently been found in design-reuse applications. The computation element (CE) abstraction proposed by L. Shannon et al. [97], decouples the datapath and system-level communication from the application-specific control to promote design reuse by localizing controller redesign of an IP module for new applications.

The control and data flow extraction techniques have been used in *design for testability* in order to reduce the complexity of test generation in large hardware designs [43], [88]. In modeling formalism such as in Petri nets, the control-data slicing could be used to allow different tools to operate on distinct parts of the model [38].

Furthermore, to enhanced the understandability in hardware debugging tools, our control-data slicing techniques could be helpful to convert HDL descriptions into graphical state machines for control parts and data flow networks for datapaths so that users can understand design functionality more quickly as proposed by J. Kim et al. [63].

3.8 Conclusions

We have proposed an algorithm based on VHDL slicing to separate *control* and *data* in synchronous descriptions of IP modules described in VHDL. The proposed slicing is intended for IP modules with mixed control and data activities according to the intuition of "Control" highlighted in chapter 2. The VHDL description is provided with labeled data inputs to the slicing algorithm. It performs a control and data flow analysis based on the knowledge of data inputs, and results two separate entities as *control slice* and *data slice* communicating with each other. For slicing modules with local variables we have proposed a solution in terms of VHDL transformation and restructuring to enhance the slicing.

Control-data slicing results strongly depend on the labeling of the data inputs of the IP module. Right choice of the data inputs is necessary for a reasonable separation between control and data. With proposed slicing algorithm, we aim to obtain data independent Control. Additional slicing rules are integrated to obtain a separation with Control depending on data according to a different notion of the control depending on the designer's anticipation.

In case of modules with data independent control, we can isolate the control slice to study only the timing related properties of the IP module without considering data processing. This would allow to reduce the state space of the whole module, and save the overall model checking time. Similarly, data slice having no impact on the control slice in such modules can be replaced by a high level fast implementation which consumes less number of cycles during simulation as compared to original module. The data slice isolated from control slice is also intended for abstract consideration of data to manipulate formal static analysis of data flows.

Possible future work in control-data slicing might be the soundness proof of the slicing and transformation algorithms. It would be interesting to prove that data registers in the data slice are flow dependent on data inputs where as control registers do not have any flow dependence on data inputs after slicing. VHDL syntax can also be extended to consider non-synthesizable and behavioral VHDL models.

Next chapter presents another idea of control and data consideration based on boolean data dependencies which might lead us to improve the slicing in low level modules. On the other hand, the proposed slicing algorithm is reused in the next chapters, where we talk about the formal verification of data flows in modules. The slicing assists us to investigate the data dependencies only within the data slice. An application of slicing reduces the complexity of processing in the formal verification approach as discussed in chapter 5.

Chapter 4

Data dependency analysis using Significance

4.1 Introduction

Formal verification of datapath properties of IP modules is a tedious task due to the presence of large number of registers and complex functions. Many useful properties of datapath avoid actual data values in specifications and take into account only the intentional presence or absence of data at specified clock instants. With such consideration, we can verify properties concerning the *intentionality* of data at specific time without considering actual concrete values.

We introduced the notion of "*Significance*" in chapter 2 to identify automatically the 'Control' in IP modules. In this chapter, we will characterize significance as a mean to represent intentionality of data in IP modules taking into account the impact of boolean data dependencies among variables. The concept of significance in this chapter, is aimed at assisting static formal verification of a class of data related properties of modules.

We will first give an informal illustration of significance and its relationship with the boolean data dependencies among variables in this section. The state of the art related to the idea of verification using Significance is described in the next section. We present a theoretical description of boolean data dependence and significance functions for low level modules in section 4.3, and provide a straightforward realization in section 4.4. Refinements in significance semantics have been proposed in section 4.5. We will portray the use of significance for static formal analysis of the datapath properties in section 4.6.

4.1.1 Significance and intentionality of data

A synchronous module receives data from environment via inputs and transmits results calculated from these inputs to the environment via outputs at each clock cycle while making transitions of its internal states. By *environment*, we mean the surrounding IP modules connected and synchronously interacting with the module.

The environment provides *intentional* data to module to obtain an intentional result. However, there exist some clock instants when a specific input is not being used in the computation of a certain output and hence can acquire an *unintentional* value which is irrelevant to the computation.

The ***significance*** is used as a medium to represent the *intentionality* of data for the hardware module being used in an environment. The presence of an intentional (*significant*) value on input has an impact on the outputs. On the other hand, when this value may not have any impact on the results then it is considered as *non-significant*.

In chapter 2, the significance was considered as a boolean attribute associated with data values which indicates that during a specific operation of the datapath, a *significant* data value at specific time instant is sampled and used to compute a significant output result whereas a *non-significant* value propagating into the module results an irrelevant output.

As discussed in precedent chapters, hardware modules consist of the datapaths with interacting control state machines. Significance only matters about signals carrying data values in the datapath whereas control signals containing critical timing references of data activities are considered as significant. Therefore to avoid the unnecessary complexity, we use the slicing algorithm mentioned in chapter 3 to split a module as *data slice* and *control slice*. We then apply the significance issues only to the data slice.

Illustration of significance and intentionality We will give an illustration of the notion of significance with respect to the intentionality of designer. A D flip-flop has been shown in Figure 4.1 as RTL circuit diagram with equivalent VHDL description. Notice that in comparison with the illustration of Figure 2.4 in chapter 2, we have already been provided here, a separation of control and data inputs by the designer. The CLK is also considered as an implicit input here.

```
architecture RTL of DFFE is
begin
  process(CLK)
  begin
    if (CLK = '1' and CLK'EVENT) then
       if (E = '1') then
          Q <= D;
       end if;
    end if;
  end process;
end RTL;
```

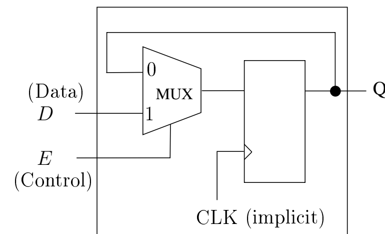


Figure 4.1: D flip-flop with Enable

We suppose that D is the data input and E is the control input. Suppose that data input D is extended with a significance attribute expressing whether its value is intentional (significant) or accidental (non-significant). E being a control input is always significant. Q is initialized as non-significant. If E is high, then D is written to the flip-flop at rising edge of CLK and Q becomes D. Thus when E is high, Q becomes significant at next clock cycle if and only if D is significant at present clock cycle. When E is low, then value of Q remains stable and its significance too. At that time, value as well as significance of D is not used. Thus additional attribute associated with signals possesses information that how intentional value of Q depends on intentional value of D and E.

The significance information provided at inputs at a given clock cycle travels through the module and appears at outputs at some later clock cycle. We can establish relationships among the input and output significance to statically analyze the flow of significant and formally prove the concerned properties.

4.1.2 Boolean data dependencies

The significance information propagates through internal registers and combinational logic. The rules which propagate the significance through the module are called *significance semantic rules* which play important role in establishing the relationship between input and output significance. These rules are based on the notion of *boolean data dependence* among different variables. We say that a boolean output is *data dependent* on a boolean input, if changing the value of this input causes a change in value on the output. We will describe the concept of data dependence for boolean formulas and hardware modules. Consider a two input AND function with inputs labeled as A, B and output as Q. We can intuitively see following observations

1. Value of Q depends on A and B because both can cause an impact on value of Q.

2. If we consider that A is fixed to logic false then output Q does not depend on value of B. Symmetrically, if B is fixed to logic false then Q does not depend on A.

These observations reveal two different notions of dependence in hardware modules. In first case, Q *statically* depends on A and B. We call this kind of behavior as ***static data dependence*** because no input is assigned any value to execute and determine the output. In second case, we say that *dynamically*, Q does not depend on A if B is logic false. We assign values to certain inputs to check the impact of other inputs. This requires semantic execution (simulation) of the model. Therefore, we call this kind of behavior as ***dynamic data dependence***. The concept of static and dynamic dependence is illustrated with details in section 4.3.

In the next section we will describe state of the art techniques for static verification similar to our approach.

4.2 State of the art

Our notion of significance permits us to formally investigate and validate the flow of data in IP modules. The proposed idea of functional verification of data flow using significance is similar to dynamic taint analysis [32], [74], [78] which consists of marking and tracking the information flow within a software program. In dynamic taint analysis, program variables are associated with taint marks and flow of tainted values is investigated by control and data flow analysis in the program. The taint analysis is aimed at detecting and preventing attacks on software programs for testing and debugging purpose.

Dynamic taint analysis targets only software programs, whereas we are using a similar notion for testing and debugging hardware designs. The analysis of J. Clause et al. [32] is done on low level binary representation of programs. In comparison, we are working on low level representation of hardware modules in terms of boolean equations and registers (gate level). Taint propagation policies implemented in taint analysis are conceptually similar to static and dynamic significance semantics in our methodology. Taint analysis is based on compiler techniques to determine control and data flow dependencies, and there are no formal basis in the proposed approaches. We have instead a formal description of data dependencies and significance in hardware modules.

T. Tolstrup et al. [99] have presented information flow analysis of a subset of VHDL. The result of the analysis is a non-transitive directed graph. The graph has a node for each variable and signal used in the program and a directed edge between nodes if information *might* flow from one node to another. The information flow obtained by such analysis is helpful for a

class of security applications. The complete information flow graph obtained by this approach can be very huge in large designs. The analysis is based on high level VHDL source code which needs tedious compiler techniques whereas for accuracy and ease of implementation purposes, our analysis targets low level representation which can readily obtained from RTL VHDL source codes by synthesis.

Analysis of behavioral VHDL code based on data flows has been discussed in [96] and [13], aimed at identifying properties of digital circuits from synthesis and testability point of view. The analysis presents an informal way of discovering potential deadlocks in VHDL behavioral descriptions raised from information flow among processes via signals. Flow analysis at behavioral level are useful to debug highlevel specifications of the module. Our analysis is formal with bit level accuracy to investigate many low level issues in hardware modules which are not captured by high level analysis techniques.

4.3 Theoretical description of Significances

In this section we will describe the theoretical grounds for the notion of *significance* and the *boolean data dependencies* in low level hardware modules. We will first describe the dynamics of module and portray the concept of static and dynamic dependencies as follows.

4.3.1 Dynamics of the module

In chapter 2, we have provided a definition of hardware module as 3-tuple $M = (I, O, f)$ with respect to its inputs I, outputs O, and the internal semantics f. The semantics f can be described in terms of a set of boolean formulas denoted as Φ, and the set of registers denoted as R. For each output or internal signal we have a boolean formula denoted as $\varphi \in \Phi$.

We consider synchronous IP modules at abstraction level of clock cycles as shown in Figure 4.2, where clock is considered as an implicit input. This is equivalent to the classical *mealy state machine model* of the hardware [87]. A value on a wire is measured to be valid at regular time intervals corresponding to the rising edges of the clock. Such module consists of registers (memory elements), combinational logic function, inputs and outputs. In Figure 4.2, symbols i, o and r represent input signal, output signal and register respectively. ${r_j}'$ represent input signal to the register r_j.

The index j symbolizes the multiplicity of these boolean signals. $\mathbf{c}$ collectively represents the combinational computation of the module. We consider synthesizable modules without combinational loops in $\mathbf{c}$. A combinational computation $\mathbf{c_x}$ can be viewed as equivalent boolean formulas φ_x in which the variables are registers r_j and inputs i_j of the module.

Module execution

The cyclic execution of the module is based on the fact that values on inputs and registers, determine the values of outputs and the registers in subsequent clock cycles. We denote the current timing instant (rising edge) as t and next instant as $t + 1$ where t, $t + 1$ are natural numbers. We will designate the value of a signal x at instant t by x^t.

The succession of events caused by a rising clock edge at time instant t, provokes an update of registers until edge $t + 1$ when r_j receive r'_j. The outputs o_j, and inputs of registers r'_j are updated between events t and $t + 1$, which are combinationally dependent on registers r_j and inputs i_j. Considering the overall system with environment, the outputs o_j of a module M

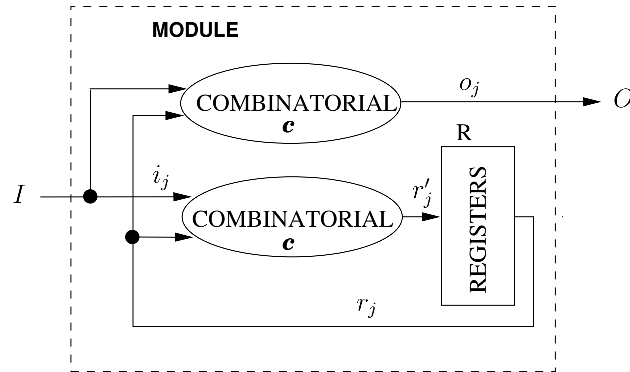


Figure 4.2: Module M

are the input i_j of an other module M', therefore a change in the value of the outputs of M, leads to an update of *values* of M' which combinationally depend on the output of M. These *values* are the registers r_j and the outputs o_j of the module M'.

Hence this causes a certain number of iterations of value updates. This iterative process terminates because we suppose that there are no combinational loops in the module as well as in overall system. We have therefore $o_j^{t+1} = \varphi_{o_j}(I^{t+1}, R^t)$ and $r_j^{t+1} = r_j'^t = \varphi_{r_j}(I^t, R^t)$. In this way, we characterize the evolution of states of a module as a function of sequence of its inputs with passing time.

State of a module at a particular time instant corresponds to the value of its registers at that instant. The next states are determined as a function of inputs and current states. The outputs are also determined by the current states and the new inputs. A state can depend on all the inputs since the initial state. For instance, a module can emit the sum of all the inputs received since initial state (modulo 2^n where n is the number bits of the output). Also to capture the *dependence* between inputs and outputs and expressing their concerned properties, we represent the sequences since initial states.

4.3.2 Boolean data dependencies in modules

By *dependence* of an output 'o' on an input 'i', we mean that value on 'i' has an impact on the value of 'o'. If we consider the "***static data dependence***" for example, this can lead us to a trivial definition. It tells that there exists a dependence if we exhibit a context in which changing the value of 'i' causes a change in value of 'o'. So in this way, the output of an AND function logically depends on all its inputs, the output of a register depends on its input at precedent clock cycle, the output of a multiplier circuit depends on both multiplier as well as multiplicand. Hence the static dependence signifies that there is a data dependence, if there is a possibility of the impact of an input on the output. Static notion of dependence represents an absolute dependence of an output on all the inputs which might involve while executing the module.

When we talk about the "***dynamic data dependence***", the concept is much less evident. We do not talk about the "possibility" of an impact of inputs because the context which represents the dynamic aspect of dependence is fixed. With dynamic dependence, we address the questions such as "*in a given context, a particular input has impact on the output*". For

multiplication circuit, if multiplier is zero then result does not depend in multiplicand. As an example, for AND gate with three inputs following analysis seems to be promising:

- The only context we can consider for an input is the value of other two inputs
- If all the inputs are `1`, all the inputs have impact on the output because changing the value of any input changes the output result
- If single input is `0` then only that input has an impact on the output because changing the value of it will change the output

However, following observation does not satisfy the notion of data dependence:

- If all three inputs are `0` then no input has impact on the output because changing the value of a single input keeping rest of the inputs unchanged does not change the output!

We see from the analysis with dynamic notion, that an individual input can not represent the combined intervention of the inputs on an output. Based on this fact, we can not really say about the *context* for a specific input in a satisfiable way. This is why while treating the case of dynamic dependence we drift from the concept of dependence towards an other concept of natural language which is similar to what we actually want to represent and is described by the fact that *a part of the inputs determines an effective computation* while rest of the inputs do not propagate the impact. The idea is that, all the valuations associating same boolean values to that part of the inputs, while associating any values to the rest of the inputs, brings same result on the output.

4.3.3 Approximations of static and dynamic dependencies

We describe the approximate realizations of the data dependencies in terms of *static and dynamic significances*. S. Coudert [35] has provided a formal demonstration of significance and sound semantic rules in boolean formulas and hardware modules. These semantics lead to the formal proof of an important property of the modules which informally states that with respect to sound semantic rules,

> "If an output of a module is significant at a specific clock instant then all the inputs on which that output depends, must also be significant".

This property is concerned about the *dependency relationship* between the significance of inputs and outputs of the hardware module which can be represented in terms of basic boolean functions. Boolean functions have been considered in the form of generic propositional logic to mathematically characterize the behavior of the hardware modules. For understanding mathematical basis of the proposed semantics, we encourage the reader to read the reference article [35]. Here we will only give the recursive definitions of the approximate semantics for static and dynamic significances and relate them with our implementation through the correctness proofs.

S. Coudert has proposed the notion of ***significance function*** for boolean formulas and proved the canonical extension to the hardware modules consisting of boolean formulas and registers. A significance function for a module determines, which outputs are significant with respect to significant inputs. In general, the significance function takes into account the

significance as well as values of the inputs to determine significance of the output. This permits an approach aimed at representing the *static* and *dynamic* data dependence as described in preceding subsection. The significance function requires tagging each input with a boolean value representing significance character. It calculates output significance based on the history of the input values of the module such that a significant output at clock instant 't' is a function of past values as well as past significances before 't'.

The static and dynamic significance functions are represented as $\mathcal{S}_{stat}(\varphi, v, s)$ and $\mathcal{S}_{dyn}(\varphi, v, s)$ respectively. In this representation, 'φ' denotes a *boolean formula*, v denotes the *valuation function* of the boolean formula which results the effective boolean value of the formula as `0` or `1`, and 's' denotes a valuation function similar to 'v' but results the obtained significance in terms of `0` or `1`. The definitions are provided in the form of properties for single variable, as well as for basic boolean operations $\wedge$, $\vee$ and $\neg$.

Static significance

The static significance function $\mathcal{S}_{stat}(\varphi, v, s)$ is defined by induction as follows:

$$\mathcal{S}_{stat}(a, v, s) = s(a), \; for\ a\ boolean\ variable\ a \tag{4.1a}$$

$$\mathcal{S}_{stat}(\varphi \wedge \psi, v, s) = \mathcal{S}_{stat}(\varphi, v, s) \wedge \mathcal{S}_{stat}(\psi, v, s) \tag{4.1b}$$

$$\mathcal{S}_{stat}(\varphi \vee \psi, v, s) = \mathcal{S}_{stat}(\varphi, v, s) \wedge \mathcal{S}_{stat}(\psi, v, s) \tag{4.1c}$$

$$\mathcal{S}_{stat}(\neg\varphi, v, s) = \mathcal{S}_{stat}(\varphi, v, s) \tag{4.1d}$$

These properties show that significance of boolean operations $\wedge$ and $\vee$ depends only on the significance of its operands.

Dynamic significance

The approximation of the dynamic data dependence called *dynamic significance* $\mathcal{S}_{dyn}(\varphi, v, s)$ is characterized as follows.

$$\mathcal{S}_{dyn}(a, v, s) = s(a), \; for\ a\ boolean\ variable\ a. \tag{4.2a}$$

$$\begin{aligned}\mathcal{S}_{dyn}(\varphi \wedge \psi, v, s) = (&\mathcal{S}_{dyn}(\varphi, v, s) \wedge \mathcal{S}_{dyn}(\psi, v, s)) \\ &\vee(\mathcal{S}_{dyn}(\varphi, v, s) \wedge \neg v(\varphi)) \\ &\vee(\mathcal{S}_{dyn}(\psi, v, s) \wedge \neg v(\psi))\end{aligned} \tag{4.2b}$$

$$\begin{aligned}\mathcal{S}_{dyn}(\varphi \vee \psi, v, s) = (&\mathcal{S}_{dyn}(\varphi, v, s) \wedge \mathcal{S}_{dyn}(\psi, v, s)) \\ &\vee(\mathcal{S}_{dyn}(\varphi, v, s) \wedge v(\varphi)) \\ &\vee(\mathcal{S}_{dyn}(\psi, v, s) \wedge v(\psi))\end{aligned} \tag{4.2c}$$

$$\mathcal{S}_{dyn}(\neg\varphi, v, s) = \mathcal{S}_{dyn}(\varphi, v, s) \tag{4.2d}$$

These properties show that significance of boolean operations $\wedge$ and $\vee$ depends on the significance as well as value of its operands. We can note that the 'significant logic zero' value

on any input of an $\wedge$ operation forces the output to be *significant*. Similarly, the 'significant logic one' value on any input of $\vee$ operation forces the output to be *significant*.

It is notable that static significance $\mathcal{S}_{stat}$ as proposed by [35], is similar to the semantics $\mathcal{S}_0$ informally discussed in chapter 2 for separation of Control and Data. Similarly, the dynamic significance $\mathcal{S}_{dyn}$ is also same as the additional rules mentioned as semantics $\mathcal{S}_1$ in chapter 2 in an informal way.

It has been formally proved that approximate significance semantics $\mathcal{S}_{stat}$ and $\mathcal{S}_{dyn}$ are sound but not complete. Therefore, we will not always be able to obtain the same results for two semantically equivalent modules with same external behavior observed with respect to values on the input and outputs. In particular, if we want to define sound significances, we can not be optimistic for complete significances. Despite of these limitations, these type of significances offer an advantage of being easily implementable with quite reasonable complexity.

4.4 Realization of significances

The notion of significance and corresponding semantics, could be realized in real hardware modules at gate level or RTL in various fashions. At gate level, we can easily extend the data values and define the propagation rules for basic logic gates. At RTL, we need to extend each data value with significance and define the propagation rules for each RTL operator in HDL. We discuss the realization issues of significance extension at both levels as follows.

4.4.1 Significance realization at low level

We have implemented the approximate static and dynamic semantics at gate level. As described earlier, the concept of significance applies only to *data* therefore we consider a separation of *control* and *data* by slicing algorithm proposed in chapter 3, and we synthesize the RTL *data slice* of the module to perform significance computation while keeping control slice intact at RTL.

For gate-level models, we will describe the control and data signals as follows:

1. $\mathtt{D} = \{d_1, d_2, ..., d_n\}$ is set of *data signals* such that each $d_i \in \mathtt{D}$ is a 4-valued signal taking its value from a set of symbolic values $\{\mathtt{F},\mathtt{T},\mathtt{f},\mathtt{t}\}$, where `F` and `T` represent *significant*, and `f` and `t` represent *non-significant* boolean values

2. $\mathtt{C} = \{c_1, c_2, ..., c_m\}$ is set of *control signals* such that each $c_i \in \mathtt{C}$ is a 2-valued signal taking its value from a set of symbolic values $\{\mathtt{F},\mathtt{T}\}$

The control and data signals are distinct in the module: that is $\mathtt{C} \cap \mathtt{D} = \emptyset$

For boolean constants, we decide that their values are *significant*. There is intuitive reasoning for considering constants as *significant*. Indeed, a boolean constant is very likely to be directly or indirectly decided by the designer of the hardware module. One could imagine that designer specifies: *a given boolean value is constant but unknown*. However, this case is pointless because any logic synthesizer would ultimately have to assign a default value to this constant. Consequently, in the significance computation we would analyze the boolean formulas and modules by considering known constants.

For single signal, we define the *significance function* as applicable to all kind of significances as follows:

Definition 4.1. *The image of significance function for a single signal is equal to the significance of that signal.*

Above definition corresponds mathematically to Eq. 4.1a and 4.2a.

For n-signals, where $n > 1$, the approximations to static and dynamic notions of data dependencies are realized in the form of truth tables.

Realization of static significance

We give an implementation of approximate static data dependency rules for basic logic operators among two arbitrary data signals $d_i, d_j \in \mathtt{D}$ in Table 4.1. These truth tables show that a non-significant value dominates over a significant value. We observe that significance of any

$d_i \wedge d_j$	**T**	**F**	**f**	**t**
T	T	F	f	t
F	F	F	f	f
f	f	f	f	f
t	t	f	f	t

$d_i \vee d_j$	**T**	**F**	**f**	**t**
T	T	T	t	t
F	T	F	f	t
f	t	f	f	t
t	t	t	t	t

$\neg d_i$	
T	F
F	T
f	t
t	f

Table 4.1: Truth tables for data signals (Static notion)

input is propagated towards the output no matter it is being used under the corresponding context or not, indicating that the input may have an impact on the output.

Propagation of control signals As described in precedent chapters, we consider a distinction of Control and Data in hardware modules. Therefore, the interaction between control and data signals is realized by restriction on static significance semantics where control signals are always considered *significant*.

Control signals operated with data signals are given in Table 4.2 where $c_i \in \mathtt{C}$ and $d_j \in \mathtt{D}$. These truth tables show that the value of control input to a binary operator being always significant is considered dominant over value and significance of data.

$c_i \wedge d_j$	**T**	**F**	**f**	**t**
T	T	F	f	t
F	F	F	F	F

$c_i \vee d_j$	**T**	**F**	**f**	**t**
T	T	T	T	T
F	T	F	f	t

Table 4.2: Truth tables for control and data signals

A control signal c_i at the input of $\neg$ operation, gives the control signal at the output. Similarly, a data signal at the input, gives a data output. The control signals $c_i, c_j \in \mathtt{C}$ operated among each other such as $c_i \wedge c_j$ and $c_i \vee c_j$, are evaluated by classical semantics of $\wedge$ and $\vee$.

Realization of dynamic significance

Dynamic significance is realized in Table 4.3 which reflects the context relative dependence. Notice that when a significant input determines the result, then the significance of other inputs does not matter. Compared to static semantics, significance calculation function constructed

$d_i \wedge d_j$	**T**	**F**	**f**	**t**
T	T	F	f	t
F	F	F	F	F
f	f	F	f	f
t	t	F	f	t

$d_i \vee d_j$	**T**	**F**	**f**	**t**
T	T	T	T	T
F	T	F	f	t
f	T	f	f	t
t	T	t	t	t

$\neg d_i$	
T	F
F	T
f	t
t	f

Table 4.3: Truth tables for data signals (Dynamic notion)

from dynamic semantics rules will be more complex than that from static semantics. The dynamic dependence of an output on an input is more sensitive as compared to static dependence, and considers run-time valuations of accompanying inputs to evaluate the impact. Therefore degree of optimization in case of dynamic semantics is less. They are suitable for observing the data dependencies in hardware modules with specified constraints.

We will show the correctness of these realizations with respect to the formal definitions given in [35].

4.4.2 Correctness of the Realizations

In realization of static and dynamic significances as truth tables, we have used four symbolic values to associate the additional significance attribute with existing boolean character of the signals. These symbolic values can be represented by a pair of boolean attributes (sig, val) where val stands for the objective value of the signal and sig represents the significance. The symbolic values can be mapped with 2 bits in classical boolean values. Different mapping are available, however we have chosen the one shown in table 4.4, because it is more intuitive.

	sig	val	**Intuition**
F	1	0	Significant false
T	1	1	Significant true
f	0	0	Non-significant false
t	0	1	Non-significant true

Table 4.4: Mapping table for value and significance

In general, the *significance function* $\mathcal{S}$ for $\wedge$ and $\vee$ between two data signals d_i and d_j is a function of values and significances of d_i and d_j. Mathematically, if we denote f_{sig} as a

function with its range as the set of significance (from sig), then we have

$$\mathcal{S}(d_i \wedge d_j) = f_{sig}(sig(d_i), val(d_i), sig(d_j), val(d_j)) \tag{4.3a}$$
$$\mathcal{S}(d_i \vee d_j) = f_{sig}(sig(d_i), val(d_i), sig(d_j), val(d_j)) \tag{4.3b}$$

where $sig(d_i)$ and $val(d_i)$ are boolean variables representing significance and value of d_i respectively, and $sig(d_j)$ and $val(d_j)$ represent significance and value of d_j respectively. The significance of $\neg$ operation is the function of value and significance of the input given as

$$\mathcal{S}(\neg d_i) = f_{sig}(sig(d_i), val(d_i)) \tag{4.3c}$$

Similarly, for $\wedge$, $\vee$ and $\neg$, we have general *value functions* given as follows:

$$\mathcal{V}(d_i \wedge d_j) = f_{val}(val(d_i), val(d_j)) \tag{4.4a}$$
$$\mathcal{V}(d_i \vee d_j) = f_{val}(val(d_i), val(d_j)) \tag{4.4b}$$
$$\mathcal{V}(\neg d_i) = f_{val}(val(d_i)) \tag{4.4c}$$

where f_{val} is a notation used to represent a function with its range as the set of values (from val). The value functions are intuitively independent of significances $sig(d_i)$ and $sig(d_j)$ which would also be evident by the following correctness proofs.

Correctness of static significance

AND function With the help of mapping table 4.4, the static significance of $\wedge$ operation given in table 4.1 can be represented by Karnaugh maps for significance and values as shown in table 4.5.

f_{sig}:

$sig(d_i)\ val(d_i)$ \ $sig(d_j)\ val(d_j)$	11	10	00	01
11	1	1	0	0
10	1	1	0	0
00	0	0	0	0
01	0	0	0	0

f_{val}:

$sig(d_i)\ val(d_i)$ \ $sig(d_j)\ val(d_j)$	11	10	00	01
11	1	0	0	1
10	0	0	0	0
00	0	0	0	0
01	1	0	0	1

Table 4.5: Mapped truth tables for $\wedge$ operation (static notion)

From the significance map f_{sig} of table 4.5, we obtain following significance function for $\wedge$ by K-map simplification.

$$\mathcal{S}(d_i \wedge d_j) = sig(d_i) \wedge sig(d_j) \tag{4.5}$$

Since, d_i and d_j are signals, therefore according to definition 4.1, we can write Eq. 4.5 as

$$\mathcal{S}(d_i \wedge d_j) = \mathcal{S}(d_i) \wedge \mathcal{S}(d_j) \tag{4.6}$$

which corresponds to Eq. 4.1b by substituting $\varphi = d_i$ and $\psi = d_j$ and symbolizing $\mathcal{S}$ as $\mathcal{S}_{stat}$. Therefore, we can say that semantics of $\wedge$ for data signals in table 4.1 are same as given by Eq. 4.1b.

By simplifications of the value map f_{val} in table 4.5, we obtain following value function.

$$\mathcal{V}(d_i \wedge d_j) = val(d_i) \wedge val(d_j)$$

This shows that the value of $\wedge$ function is independent of the significance of its inputs, and is coherent with the intuitive definition of Eq. 4.4a.

OR function The static significance of $\vee$ operation given in table 4.1 can be represented by Karnaugh maps as shown in table 4.6.

f_{sig}:

$sig(d_i)\ val(d_i)$ \ $sig(d_j)\ val(d_j)$	11	10	00	01
11	1	1	0	0
10	1	1	0	0
00	0	0	0	0
01	0	0	0	0

f_{val}:

$sig(d_i)\ val(d_i)$ \ $sig(d_j)\ val(d_j)$	11	10	00	01
11	1	1	1	1
10	1	0	0	1
00	1	0	0	1
01	1	1	1	1

Table 4.6: Mapped truth tables for $\vee$ operation

From the significance map f_{sig}, we obtain following significance function by K-map simplification.

$$\mathcal{S}(d_i \wedge d_j) = sig(d_i) \wedge sig(d_j) \tag{4.7}$$

According to definition 4.1, we can write it as

$$\mathcal{S}(d_i \wedge d_j) = \mathcal{S}(d_i) \wedge \mathcal{S}(d_j) \tag{4.8}$$

which corresponds to Eq. 4.1c by substituting $\varphi = d_i$ and $\psi = d_j$, and symbolizing $\mathcal{S}$ as $\mathcal{S}_{stat}$. Therefore, we can say that semantics of $\vee$ for data signals in table 4.1 are same as given by Eq. 4.1b.

By simplifications of the value map f_{val} in table 4.6, we obtain following:

$$\mathcal{V}(d_i \wedge d_j) = val(d_i) \vee val(d_j)$$

, which shows that the value of $\vee$ function is independent of the significance of its inputs. This is coherent with the intuitive definition of Eq. 4.4b.

NOT function The $\neg$ operation given in table 4.1 can be represented by Karnaugh maps as shown in table 4.7. From the significance map f_{sig}, we obtain following significance function after simplification.

$$\mathcal{S}(\neg d_i) = sig(d_i) \tag{4.9}$$

$sig(d_i)\ val(d_i)$	f_{sig}
11	1
10	1
00	0
01	0

$sig(d_i)\ val(d_i)$	f_{val}
11	0
10	1
00	1
01	0

Table 4.7: Mapped truth tables for ¬ operation

Correctness of dynamic significance

AND function With the help of mapping table 4.4, the dynamic significance of ∧ operation given in table 4.3 can be represented by Karnaugh maps for significance and values as shown in table 4.8.

f_{sig}:

$sig(d_i)\ val(d_i)$ \ $sig(d_j)\ val(d_j)$	11	10	00	01
11	1	1	0	0
10	1	1	1	1
00	0	1	0	0
01	0	1	0	0

f_{val}:

$sig(d_i)\ val(d_i)$ \ $sig(d_j)val(d_j)$	11	10	00	01
11	1	0	0	1
10	0	0	0	0
00	0	0	0	0
01	1	0	0	1

Table 4.8: Mapped truth tables for ∧ operation (dynamic notion)

From the significance map f_{sig} of table 4.8, we obtain the 3 terms in significance function for ∧ by K-map simplification given as follows.

$$\mathcal{S}(d_i \wedge d_j) = (sig(d_i) \wedge sig(d_j)) \vee (sig(d_i) \wedge \neg val(d_i)) \vee (sig(d_j) \wedge \neg val(d_j)) \tag{4.10}$$

Since, d_i and d_j are signals, therefore according to definition 4.1, we can write Eq. 4.10 as

$$\mathcal{S}(d_i \wedge d_j) = (\mathcal{S}(d_i) \wedge \mathcal{S}(d_j)) \vee (\mathcal{S}(d_i) \wedge \neg val(d_i)) \vee (\mathcal{S}(d_j) \wedge \neg val(d_j)) \tag{4.11}$$

which corresponds to Eq. 4.2b by substituting $\varphi = d_i$ and $\psi = d_j$ and, symbolizing $\mathcal{S}$ as $\mathcal{S}_{dyn}$, and val as valuation function v. Therefore, we can say that semantics of ∧ for data signals in table 4.3 are same as given by Eq. 4.2b.

By simplifications of the value map f_{val} in table 4.8, we obtain

$$\mathcal{V}(d_i \wedge d_j) = val(d_i) \wedge val(d_j)$$

, which shows that the value of ∧ function is independent of the significance of its inputs, and is coherent with the intuitive definition of Eq. 4.4a.

OR function The dynamic significance of $\vee$ given in table 4.3 is splitted into Karnaugh maps as shown in table 4.8. From the significance map f_{sig} of table 4.9, we obtain the 3 terms

f_{sig}:

$sig(d_i)\ val(d_i)$ \ $sig(d_j)\ val(d_j)$	11	10	00	01
11	1	1	1	1
10	1	1	0	0
00	1	0	0	0
01	1	0	0	0

f_{val}:

$sig(d_i)\ val(d_i)$ \ $sig(d_j)\ val(d_j)$	11	10	00	01
11	1	1	1	1
10	1	0	0	1
00	1	0	0	1
01	1	1	1	1

Table 4.9: Mapped truth tables for $\vee$ operation (dynamic notion)

in significance function for $\vee$ by K-map simplification given as follows.

$$\mathcal{S}(d_i \wedge d_j) = (sig(d_i) \wedge sig(d_j)) \vee (sig(d_i) \wedge val(d_i)) \vee (sig(d_j) \wedge val(d_j)) \tag{4.12}$$

Since, d_i and d_j are signals, therefore according to definition 4.1, we can write Eq. 4.12 as

$$\mathcal{S}(d_i \wedge d_j) = (\mathcal{S}(d_i) \wedge \mathcal{S}(d_j)) \vee (\mathcal{S}(d_i) \wedge val(d_i)) \vee (\mathcal{S}(d_j) \wedge val(d_j)) \tag{4.13}$$

which corresponds to Eq. 4.2c by substituting $\varphi = d_i$ and $\psi = d_j$, and symbolizing $\mathcal{S}$ as $\mathcal{S}_{dyn}$, and val as valuation function v. Therefore, we can say that semantics of $\vee$ for data signals in table 4.3 are same as given by Eq. 4.2c.

By simplifications of the value map f_{val} in table 4.9, we obtain following

$$\mathcal{V}(d_i \wedge d_j) = val(d_i) \vee val(d_j)$$

which shows that the value of $\vee$ function is independent of the significance of its inputs, and is coherent with the intuitive definition of Eq. 4.4b.

NOT function Since truth tables for $\neg$ function is same in case of static and dynamics semantics of tables 4.1 and 4.3, therefore we have the same mappings for significance and value for $\neg$ function as shown in table 4.7.

We also show the mapping of semantics with Control and Data considerations as follows.

Mapping Control/Data restriction

AND function The significance of $\wedge$ operation between control signal c_i, and data signal d_j of table 4.2 is given as Karnaugh maps of significance and values in table 4.10.

Here, we obtain 2 terms in significance function for $\wedge$ by K-map simplification given as follows.

$$\mathcal{S}(c_i \wedge d_j) = sig(d_j) \vee \neg val(c_i)$$

which describes that output function will be significant when value of control signal c_i is 0. Otherwise the significance of function is equal to the significance of d_j.

f_{sig}:

$val(c_i)$ \ $sig(d_j)val(d_j)$	11	10	00	01
1	1	1	0	0
0	1	1	1	1

f_{val}:

$val(c_i)$ \ $sig(d_j)val(d_j)$	11	10	00	01
1	1	0	0	1
0	0	0	0	0

Table 4.10: Mapped truth tables for $\wedge$ operation (Control/Data)

OR function In the same way, significance of $\vee$ operation between control signal c_i, and data signal d_j of table 4.2 is given in table 4.11.

f_{sig}:

$val(c_j)$ \ $sig(d_i)$ $val(d_i)$	11	10	00	01
1	1	1	1	1
0	1	1	0	0

f_{val}:

$val(c_j)$ \ $sig(d_i)$ $val(d_i)$	11	10	00	01
1	1	1	1	1
0	1	0	0	1

Table 4.11: Mapped truth tables for $\vee$ operation (Control/Data)

Here, we obtain 2 terms in significance function for $\vee$ by K-map simplification given as follows.

$$\mathcal{S}(c_i \vee d_j) = sig(d_j) \vee val(c_i)$$

which describes that output function will be significant when value of control signal c_i is `1`. Otherwise the significance of function is equal to the significance of d_j.

Again the $\neg$ function in this case has same semantics as given in truth tables 4.1 and 4.3, and the K-map as table 4.7.

4.4.3 Limitation of approximate significances

Application of static and dynamic semantics to different syntactic forms of the same boolean function may produce different evaluations of significance at output. For illustration purpose let us consider following two boolean formulas of same boolean function:

$$\varphi_1 = (a_1 \wedge \neg a_3) \vee (\neg a_1 \wedge a_2) \vee (\neg a_2 \wedge a_3)$$
$$\varphi_2 = (\neg a_1 \wedge a_3) \vee (a1 \wedge \neg a_2) \vee (a_2 \wedge \neg a_3)$$

It can easily be proved by truth tables that $\varphi_1 \equiv \varphi_2$. If we consider a case of significance evaluations in which a_1 and a_2 are significant but a_3 is non-significant then according to table 4.3, both formulas do not produce same results. For instance if $a_1 = \texttt{F}$, $a_2 = \texttt{T}$, and $a_3 = \texttt{f}$ then by substitution we obtain $\varphi_1 = \texttt{T}$ but $\varphi_2 = \texttt{t}$. Similarly, if we consider a_1 and a_2 as

```
1.    if (B = '0') then
2.        R <= R + A;
3.    else
4.        R <= '1';
5.    end if;
```

Figure 4.3: $if - else$ structure in VHDL RTL

control signals and a_3 as data signal, and apply semantics of table 4.2, we observe different significance evaluations for φ_1 and φ_2.

The limitation comes from the *syntax dependent* calculus of the static and dynamic significances. These approximations are sound but not complete. This spurious behavior can cause a loss of useful information during the verification process. Such kind of indications are called as ***false alarms*** in verification. The approximate significances can cause an indication of a significant value on an output as non-significant in a module. False alarms in the property verification process can be avoided by using significance semantic rules independent of syntax of boolean functions. However, such semantics costs more from complexity of computation viewpoint.

4.4.4 Significance realization at high level

Introducing significance at RTL is tedious as compared to that at gate level which requires redefinition of various high level HDL semantic constructs. We have high level operations such as addition, subtraction, and multiplication of integers and bit vectors at RTL. By directly implementing significance at RTL for such operators in a straight forward way, the degree of accuracy of the analysis might be reduced as remarked by following discussion.

High level data type conversion The RTL description contains complex data type conversions. For example, while taking into account a type conversion from an n-bit vector to an integer value, a significant bit vector can be converted into an integer however a bit vector containing non-significant bits is ambiguous to convert into concrete integer value. We need to enhance language constructs which take into account the significant and non-significant integers.

Structural issues Some of the problems while using significance semantics directly for RTL code are the structural issues. RTL code consists of high level language constructs like `if-else` clauses, `case` statements for which significance oriented semantics is needed to be defined.

Consider a piece of VHDL RTL description shown in Figure 4.3. In the original RTL code with 2-valued logic, the description has its defined meanings. However, if we have `A`, `B`, and `R` as 4-valued data signals being able to carry value and significance then significance of `B` is not being taken into account because in the original model the comparison of `B` at line 1 is based on 2-valued logic. While `B` being non-significant (i.e. either `'t'` or `'f'`) `R` will always be significance if same descriptions used for significance extended signals.

Therefore, either we need to define significance semantics of `if-else` and `case`, or we need to perform syntax transformation which takes into account the significance of the signals. For

```
1.   if (B = 'F') then
2.      R <= R + A;
3.   elsif (B = 'T') then
4.      R <= 'T';
5.   else
6.      R <= 'f';       -- or 't' (some non-significant value)
7.   end if;
```

Figure 4.4: Syntax transformation required at RTL

the given example of Figure 4.3, we have to rewrite the code as shown in Figure 4.4 after syntax analysis which could be tedious in complex cases.

Vector assignments While applying significance to hardware modules,it would be interesting to consider an abstraction level which takes into account the whole bit vector carrying either a significant or non-significant value. Implementing such mechanism is difficult to achieve at RTL. Consider a bit vector being assigned in following fashion:

```
s_p1:  V(15 downto 8) <=  A xor B;           -- a statement in process p1
s_p2:  V((7 downto 0) <= (other => '0');     -- a statement in process p2
```

where `p1` and `p2` are VHDL processes. If we would consider significance of individual bits then above structure of code would also be suitable for significance computation. However, to avoid complexity if we consider a bundled significance computations by merging the significance bits of the vector `V` to obtain a single significance bit, we have to make additional significance computation for bit vector `V` as follows:

```
p3: V_sig <= A_sig and B_sig;     -- additional process p3
```

where `A_sig` and `B_sig` are new signals indicating the significance of the vectors `A` and `B` respectively. Detection of assignments to bit vectors in HDL code and correspondingly deducing significance in the form of a bundled assignment requires a rich syntax and semantics analysis.

These are the main issues which we investigated during experimentation at RTL. There are more issues such as `for` loops, process activations on significance changes which could make the significance realization hard at RTL.

4.4.5 Simplification of module

The extension of the module with significance apparently increases the size of the module. However, depending upon the nature of the specific semantics, there are potential minimizations possible in the module. Particularly, in case of straightforward static significance $\mathcal{S}_{stat}$, the significance computation function obtained from modules could be quite compact due to the presence of redundant boolean variables irrelevant to the output significance.

The deletion of unused and redundant variables can practically be achieved by logic optimization algorithms. Logic optimization can remove the combinational and sequential logic corresponding to unused output values in the module resulting a simplified *significance computation model* for static verification. The degree of simplification depends on the property being verified, input constraints, and the nature of significance propagation semantics.

4.5 Refinements in significance computation

In this section, we will present some ideas of improving the approximate semantics of static and dynamic data dependencies. We wish to approach a natural computation of significance which is closer to the ideal data dependencies, and which is based on the semantic analysis of the boolean formulas rather than their syntax. We have worked around a few semantics to derive a significance computation function from a given boolean formula considering the basic properties of boolean algebra.

4.5.1 Refined significance : $\mathcal{S}_{ref}$

A drawback of static and dynamic semantics proposed by [35] is that they do not take into account the properties of *tautology* and *contradiction* in the definition. In classical logic with boolean variable x, we write these properties with follows boolean formulas

- $\phi_1(x) = x \wedge \neg x = 0$ (Contradiction)
- $\phi_2(x) = x \vee \neg x = 1$ (Tautology)

Intuitively, the significance of above formulas must be a constant, and should not depend on the significance or value of x because the resultant value is constant in both cases. However, if we evaluate the significance of ϕ_1 and ϕ_2 either with static or dynamic semantics, we obtain following significance functions.

$$\begin{aligned} \mathcal{S}_{stat}(\phi_1) &= \mathcal{S}_{dyn}(\phi_1) = sig(x) \\ \mathcal{S}_{stat}(\phi_2) &= \mathcal{S}_{dyn}(\phi_2) = sig(x) \end{aligned}$$

These equations show that, the significance of the contradiction formula ϕ_1, and the tautology formula ϕ_2 depends on the significance of the input variable x which is not true according to the intuition.

We have, therefore introduced some refinements in the dynamic semantics. The *refined significance semantics* is denoted as $\mathcal{S}_{ref}$ which takes into account the significance evaluation for tautologies and contradictions in boolean formulas.

Semantics

For two data signals, d_i and d_j, we define the ***equality test*** between their values as follows

$$\mathcal{E}qual(val(d_i), val(d_j)) = \neg(val(d_i) \oplus val(d_j)) \quad (4.14)$$

where $\oplus$ is the exclusive OR (XOR) function such that

$$val(d_i) \oplus val(d_j) = (val(d_i) \wedge \neg val(d_j)) \vee (\neg val(d_i) \wedge val(d_j))$$

The semantics of equality function $\mathcal{E}qual$ are given in table 4.12. The result of the equality test is also a boolean value representing `1` for equality, and `0` for inequality.

With the help of equality test function of Eq. 4.14, we define new significance semantics for $\wedge$ and $\vee$ with data signals d_i and d_j as follows.

$val(d_i)$	$val(d_j)$	$\mathit{Equal}(val(d_i), val(d_j))$
0	0	1
0	1	0
1	0	0
1	1	1

Table 4.12: Semantics of *equality test*

Definition 4.2 (Refined significance).

$$\mathcal{S}_{ref}(d_i \wedge d_j) = \mathcal{S}_{dyn}(\mathcal{S}_{dyn}(d_i \wedge d_j) \wedge \neg \mathit{Equal}(val(d_i), \neg val(dj))) \quad (4.15)$$
$$\mathcal{S}_{ref}(d_i \vee d_j) = \mathcal{S}_{dyn}(\mathcal{S}_{dyn}(d_i \vee d_j) \vee \mathit{Equal}(val(d_i), \neg val(dj))) \quad (4.16)$$

The semantics for single variable and the $\neg$ operation is unchanged. i.e.

$$\mathcal{S}_{ref}(d_i) = sig(d_i)$$
$$\mathcal{S}_{ref}(\neg d_i) = sig(d_i)$$

With definition 4.2, we can see that, in case of tautology or contradiction where $d_j = \neg d_i$, we can obtain following, by substitution

$$\mathcal{S}_{ref}(d_i \wedge \neg d_i) = \mathcal{S}_{dyn}(\mathcal{S}_{dyn}(d_i \wedge \neg d_i) \wedge \neg(\mathit{Equal}(val(d_i), \neg val(\neg d_i))))$$
$$\mathcal{S}_{ref}(d_i \vee \neg d_i) = \mathcal{S}_{dyn}(\mathcal{S}_{dyn}(d_i \vee \neg d_i) \vee (\mathit{Equal}(val(d_i), \neg val(\neg d_i))))$$

Since $\mathit{Equal}(val(d_i), \neg val(\neg d_i)) = 1$ according to Eq. 4.14, and $\mathcal{S}_{dyn}(d_i \wedge \neg d_i)$ results $sig(d_i)$ according to Eq. 4.10 and 4.12, therefore we obtain following

$$\mathcal{S}_{ref}(d_i \wedge \neg d_i) = \mathcal{S}_{dyn}(sig(d_i) \wedge \neg(1))$$
$$\mathcal{S}_{ref}(d_i \vee \neg d_i) = \mathcal{S}_{dyn}(sig(d_i) \vee 1)$$

by simplifying above equations, we have

$$\mathcal{S}_{ref}(d_i \wedge \neg d_i) = \mathcal{S}_{dyn}(sig(d_i) \wedge 0)$$
$$\mathcal{S}_{ref}(d_i \vee \neg d_i) = \mathcal{S}_{dyn}(sig(d_i) \vee 1)$$

, which results according to Eq. 4.10, 4.12, and table 4.4, the following significances.

$$\mathcal{S}_{ref}(d_i \wedge \neg d_i) = \mathtt{F} \equiv (1, 0)$$
$$\mathcal{S}_{ref}(d_i \vee \neg d_i) = \mathtt{T} \equiv (1, 1)$$

We see that with refined significance $\mathcal{S}_{ref}$, we obtain a constant significant values for tautology and contradiction formulas. Thus definitions 4.15 and 4.16 are more natural as compared to definition 4.2b and 4.2c.

So for as the realization of $\mathcal{S}_{ref}$ is concerned, the definition could easily be integrated with static and dynamic realizations by defining functions which first checks the equality of the two operands and then addresses the table 4.3 of $\mathcal{S}_{dyn}$ for significance evaluation.

4.5.2 Semantic computation of Significance

The refined significance $\mathcal{S}_{ref}$ could further be enhanced by considering a semantic calculus of significance. A new approach of significance computation is under consideration of the LabSoC team for optimizations. With this approach, the significance computation is not based on the rules respecting the original structure of the boolean formula. It rather computes a new formula from a given boolean formula by applying induction principle on its variables.

The semantic calculus proposed in the inductive approach associates to each n-variable boolean formulas $\varphi(x_1, \ldots, x_i, \ldots, x_n)$, a new $2 \times n$-variable formula

$$\mathcal{S}_{\varphi}((x_1, \ldots, x_i, \ldots, x_n)(s_1, \ldots, s_i, \ldots, s_n))$$

computing the significance of $\varphi(x_1, \ldots, x_i, \ldots, x_n)$. Here $(s_1, \ldots, s_i, \ldots, s_n)$ are significances corresponding to the variables $(x_1, \ldots, x_i, \ldots, x_n)$. The semantic calculus captures the significance in the sense that

$$\mathcal{S}_{\varphi}((x_1, \ldots, x_i, \ldots, x_n)(s_1, \ldots, s_i, \ldots, s_n)) = \mathtt{1}\ (\mathit{true})$$

if and only if, $\varphi(x_1, \ldots, x_i, \ldots, x_n)$ only *depends* on significant values of each x_i.

The formula $\mathcal{S}_{\varphi}$ is quite different from φ and could potentially be complicated depending upon the nature of the given boolean function.

4.6 Applications of Significance

Significance can serve for verifying properties concerning the valid presence or absence of data. With this concept it would be intresting to extract the significance computation model from the given hardware description of the module according to a given set of rules for the propagation of significance among boolean operators, and statically investigate the boolean data dependencies between inputs and outputs.

Significance can be used to statically detect errors in modules where some invalid data values might be provided at the inputs which propagate to the outputs producing invalid results. Similarly, some invalid (non-significant) initialization of the internal registers of the module would cause an erroneous output which might be detected by this approach.

Since concrete data value computations are ignored in case of static semantics, we are able to achieve optimizations in data processing. Moreover for properties concerning only significance, output values are ignored. Therefore, combinational and sequential logic elements taking part in output value calculation could be removed to obtain a simplified significance calculation model.

Common bugs addressed by Significance

The concept of significance can be used to perform static formal analysis for bug detection in pipeline of datapaths. We can characterize different bugs in relation to significance showing how the concept can serve to specify and validate bugs in complex systems.

- Data that is not expected to be lost (significant) in the pipeline is lost (becomes non-significant) in the datapath. By analyzing such bugs at gate level, we can observe that they can arise for instance in case, where a `reset` signal is mistakenly asserted and hence destroying the useful significant values of data registers. Selection signal of a multiplexer might be generated incorrectly such that a non-significant data is chosen at the output.

- Data that was not expected to be destroyed (made non-significant) mistakenly survived in the datapath registers (remained significant)

- Some wrong (non-significant) source of data has been used for some computation

- A bug in the control logic of an array implementing a FIFO (first in first out) queue can cause a problem. For instance, it would be required to verify that if we send a significant data sample followed by a non-significant sample as input to FIFO then it is never possible that we see a non-significant data sample followed by a significant sample at the output.

- The control slice containing critical timing information of the module should not be affected by non-significant data values. For example in serial output of the UART module, we can statically verify that the start, stop and parity control bits (timing information) in the output packet are never non-significant in any possible state of the module.

- Erroneous cases where some invalid (non-significant) initialization of the internal registers of the module would cause an invalid output might be formally detected by significance. Such errors could also be detected by `'U'` (uninitialized value) propagation during reset phase in VHDL simulations with IEEE multivalued logic system [4]. However, these simulations are neither formal nor exhaustive. The *Significance* might serve here for static formal verification.

The subsequent chapter describes the utilization of the *Significance* in formal verification.

4.7 Conclusions

In this chapter, we have proposed the notion of *Significance* to represent intentional data movements in IP modules. We have presented a theoretical description of the significance, and related it with more basic concept of static and dynamic data dependence among boolean variables in low level IP modules. Approximations to the ideal static and dynamic data dependencies have been proposed for homogeneous significance calculations.

The approximate static and dynamic semantics being sound but not complete, have an advantage of easy implementation. Although due to syntax dependent computations, there are possibilities of *false alarms* during verification process, yet they come up with reasonable complexity of implementation. Static semantics provide a way of model simplification for static verification of a class of datapath properties. Dynamic semantics being more powerful but complex than static semantics can be used to formally study the critical data dependencies in low level hardware modules.

A realization of significance at RTL and gate level is discussed, and possibility of simplification of modules is described. Application of significance at gate level description avoids subtle syntax and semantic analysis for RTL operators. Significance can also be defined directly for the RTL operations to avoid the first synthesis step, and to allow working at RTL instead of gate level. This would need to define significance propagation rules for high level operators such as addition and multiplication and complex syntactic analysis to deal with high level control statements.

We have proposed refinements in approximate significance computations taking into account the interpretation of tautologies and contradictions. A semantic computation of significance could be provided in the future which would lead to an analysis independent of the syntax of the boolean formulas.

It would also be interesting to prove the correctness of refined semantics. That is, for any n-variables boolean formula, the $\mathcal{S}_{ref}$ function leads to a unique $2 \times n$-variables boolean formula for its *significance*. A compact realization, and integration of refinements with existing significance implementation could also be a future contribution.

Chapter 5

Verification based on Significance

5.1 Introduction

The *Significance* attribute presented in precedent chapter was aimed at statically investigating the data flow, and verifying functional correctness of datapath operations in hardware module. This chapter is aimed at utilizing the concept, in a prototype verification framework to assist a validation engineer in verifying important datapath properties. In particular, we are interested in formally verifying properties of the module which state that *desired outputs arrive well at the time when they are expected to be arrived.* Such properties talk about the intentional *presence* or *absence* of data on the inputs and outputs of the module.

The presented verification technique is based on model checking [29], [31] in which we first automatically transform the module to enable it for sound significance calculations, and then provide a way to specify properties concerning significance with an ease of implementation for an engineer. The verification approach, being static and formal, provides an easy and powerful way of debugging data flows in hardware modules as compared to simulations.

We will first describe an overview of the verification process by relating the behavior of the module capable of performing sound significance calculations with the designer's anticipation about the expected behavior. In the next section, we will give the details how a given hardware module undergoes different steps in the proposed verification process to finally verify the specified property. An example is used to illustrate this idea in section 5.3. A prototype implementation of the framework is presented in section 5.4, and experimental results are provided to show the effectiveness of the approach in section 5.5.

5.1.1 Significance and timing

Conceptually, a synchronous hardware module being used in an environment samples significant data inputs at a specific time instant (rising/falling edge) to produce significant result at an other time instant. "*When a data input or output should be significant*", is the ***timing information*** of that input or output. As described in precedent chapters this timing information is intuitively present in the *control slice* of the module. Timing information of data inputs and outputs is needed to specify properties concerning significance of inputs and outputs of hardware module in the verification process.

The timing information might be available in different ways. One way is to obtain it from the control slice of the module. For example in data computation intensive hardware designs, a primary control input is used in such a way that it enables the data input loading at a specific

clock instant. Hence this input could be used as an indication that some data input should be significant at that clock instant for an intentional execution of the module. Similarly, on the output interface, a *control output* is sometimes generated to provide an information to the outside world that valid result is available at the output.

It is also possible that timing information is implicitly known. For example, the designer knows that a particular output of a module will be available (significant) after 8 clock cycles of the injection of the inputs.

For the sake of illustration here, let us suppose a generic hardware module with given input and output interfaces as shown in Figure 5.1. Here $Data_{in}$ is the data input of the module which is supposed to be injected the significant or non-significant data values from the environment at each clock cycle of clock input clk. An input named "data strobe input" (shorted as DSI) as shown in Figure 5.1, is used to decide about the significance of the data input $Data_{in}$. That is, when DSI is asserted, a valid data value at $Data_{in}$ is loaded to module for an intentional computation.

On the output interface, signal $Data_{out}$ carries the result, and a special control output "data strobe output" (shorted as DSO) as shown in Figure 5.1 indicates that the significant result is available at $Data_{out}$.

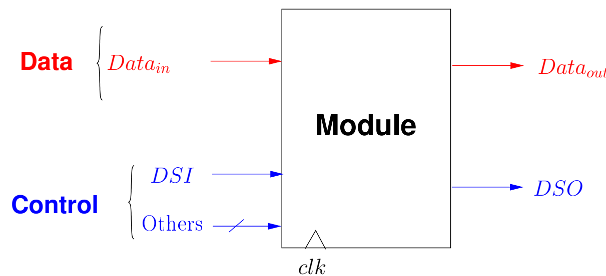


Figure 5.1: A generic module under verification

Hence the DSI and DSO reflect the timing information of a class of hardware module of Figure 5.1 which is used as reference during verification process. The generic module of Figure 5.1 represents a major class of hardware modules which are used as IP modules in a large scale design. Therefore, we have chosen it for illustration. If the timing information is implicit within the module, and not present in the form of DSI and DSO signals then it can explicitly be derived from the specifications of the relevant design, and used as timing reference while specifying property.

5.1.2 Module's behavior and designer's anticipation

Figure 5.2 depicts a conceptual view of the verification approach. There are two distinct notions: the *module's behavior*, which is subjected to perform significance calculations, and the *designer's anticipation* about the desired output at their right timing instants.

In generic module of Figure 5.1, DSI and DSO serve to establish the *designer's anticipation* about the significance. Based on this information, the designer provides significant inputs to the module when DSI is active, and he/she believes that the output has been computed by the module uniquely from those inputs which were tagged as significant, therefore there will be significant output expected to be arrived at the moment when DSO is active. Utilizing the

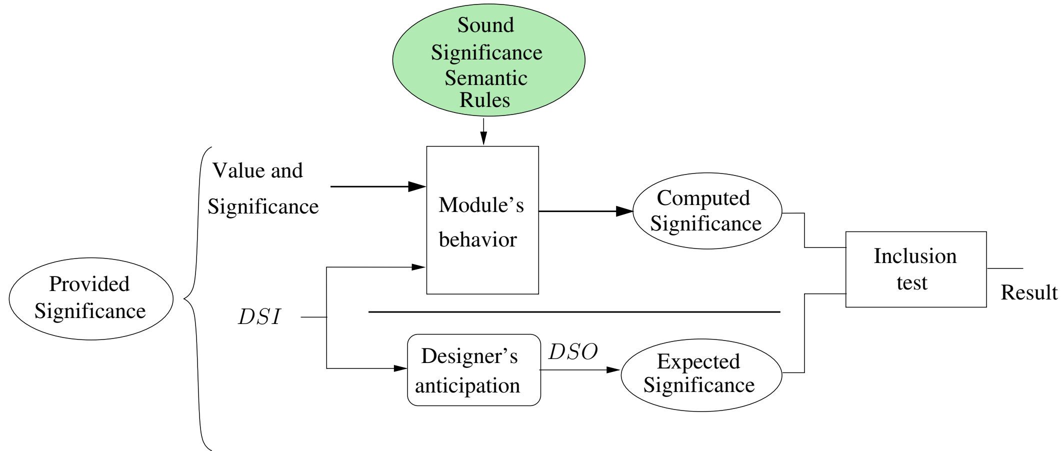


Figure 5.2: A conceptual description of verification approach

significance semantic rules, the input significance in the module is propagated to computed the significance at the output. The significance *computed* by the module is checked for inclusion with the *expected significance* according to the anticipation of the designer to obtain the result of the verification. Based on sound but incomplete significance semantics, we can justify the result in follows ways.

a) Perfect correspondence If the computed significance result by the module conforms to the expected significance result by the module then there is a perfect correspondence between the module's behavior and the designer's anticipation. The module is functioning correctly according to the specified property.

b) Error indication If the designer provides a non-significant input to the module it is possible that an output is significant due to the fact that given non-significant input was not propagated and not used in the computation of the output, whereas the designer may anticipate that the result produced by the non-significant inputs should be non-significant. This is possible due to some design error in the data flow of module which does not properly transmit a non-significant input value towards output. This category of malfunctioning is named as ***design errors*** and are detectable in the proposed framework by using sound semantic rules of significance propagation.

If the module does not contain any design bug, and the designer knows which inputs are being used in the computation of the data output, and he intentionally sets those inputs as significant at the time of their loading; then by sound semantic rules of significance propagation, the module should produce significant result at output. However, it is very difficult for the designer to know that a particular input has been used in the computation of an output in a certain execution. Because in different executions of the module, there is a different context of that input which implies whether that input has been propagated and used in the computation of the output or not. This limitation originates from the static/dynamic data dependent behavior of the hardware descriptions as explored in precedent chapter.

c) False alarm indication Sound significance semantic rules have limitations and cause the propagation of a non-significant input to the output even if it is not really used in the computation of the output. This is possible due the fact that semantics used for the significance computations are syntax dependent and incomplete. They are sound but not complete, therefore they can indicate an error in the module even if there does not exist any real error. Such type of malfunctioning is categorized as ***false alarms***. They can cause a loss of useful information, because they do not indicate the availability of significant data at the output when it was expected to be significant.

The degree of false alarms during the verification depends on the strength of the significance semantics. Stronger is the significance semantics, less is the possibility of false alarms. However, a strong significance semantics costs more from complexity of computation point of view. We illustrate the false alarms by example as follows:

Illustration of false alarms Let us suppose a module M, which multiplies two numbers provided simultaneously at the data inputs A and B, and gives the result on its output Q 8 cycles later. The timing reference model (control slice) for the designer's anticipation specifies that "when inputs A and B are loaded at time instant t, output Q will be available at $t+8$".

Under normal condition, if module M does not contain any design bug, and both A and B are supplied significant inputs at t then Q must be significant at $t+8$. However, if we suppose a special case of multiplication in which A is permanently set to significant 0, and the designer anticipates that output Q should also be significant 0. Then by using static semantics described in precedent chapter, if B was set non-significant then output Q will be evaluated to be non-significant, which is an indication of false alarm. In this way the calculated result being non-significant does not match with the designer's expected result which was significant 0. The reason is that static semantics is weak, and propagates the non-significant value of B to the output even if it is not really used to evaluate Q when A is set to significant 0.

If we use dynamic semantics the result will be different. The dynamic semantics being stronger than static semantics makes the output Q significant 0 even if B was non-significant. Dynamic semantics do not propagate the non-significant value of B to the output hence the calculated result is correct as expected by the designer, and there is no false alarm in this particular case.

It would be interesting to detect the malfunctioning in a module concerning significance of the inputs and outputs by static verification techniques such as model checking. We propose the technique based on basic model checking to study these behaviors statically in a verification framework. We give a prototype implementation of the framework. During verification process, given module undergoes some transformations which finally result a model suitable for verifying properties concerning significance by model checking. Our prototype verification framework enables a validation engineer to specify semi-automatically an environment according to intended property. This environment implements the property monitor in the form of state machine as exploited in [16], [82]. If the generated results do not match with the expected results then violations of properties may be indicated.

The major operations in the verification process have been developed in the next section with detail.

5.2 Description of verification technique

Figure 5.3 depicts the verification process in the proposed technique. We will describe each operation during verification process is a sequential order.

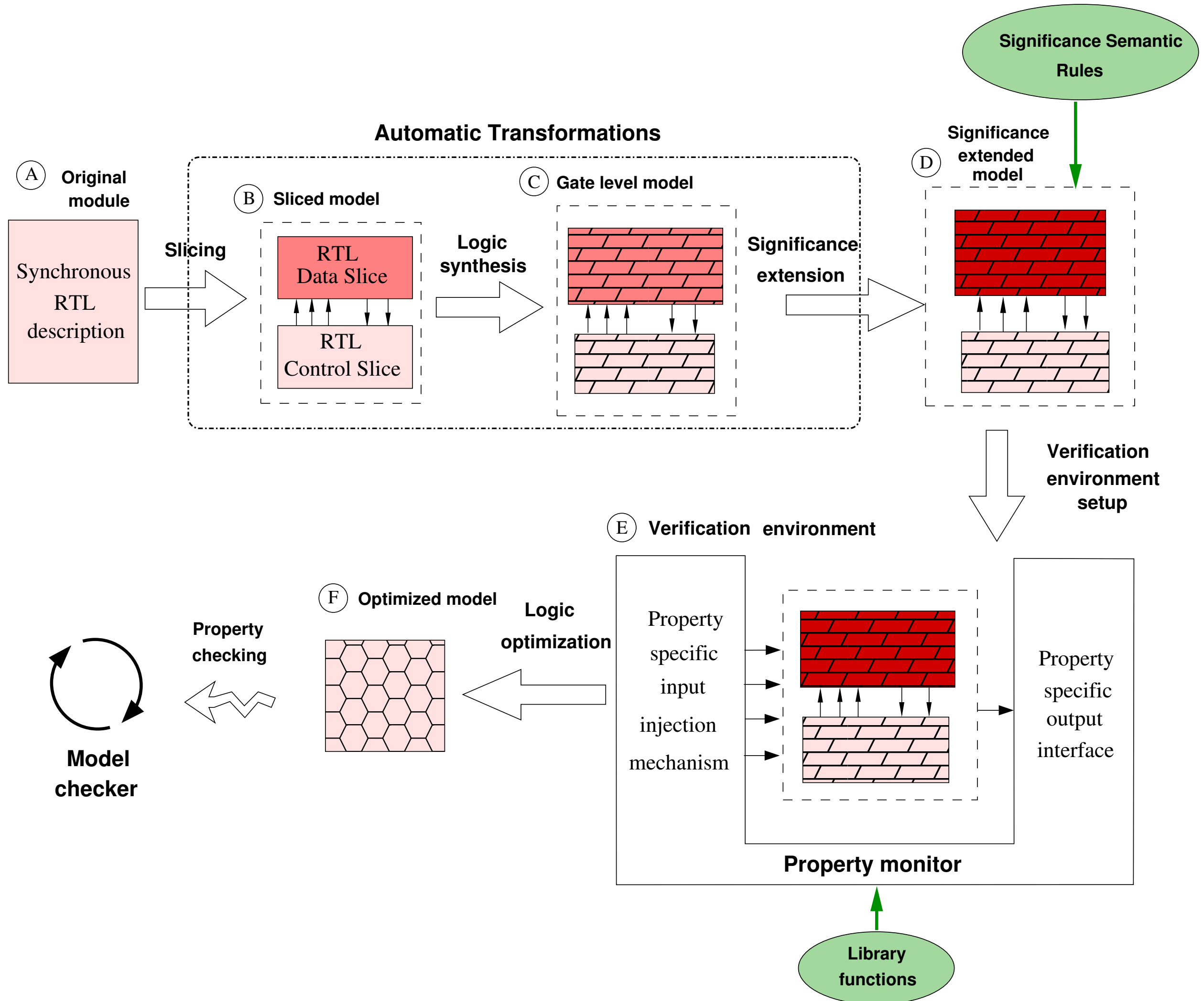


Figure 5.3: Processing steps during verification

We are considering synchronous RTL design described in VHDL for verification in the proposed framework, as marked by block Ⓐ in Figure 5.3. An isolated generic module has been shown in Figure 5.1. From the I/O interface view point, distinction is considered between *control* and *data* inputs. The control inputs are considered as always significant during a data computation, whereas data inputs may be significant or non-significant at different clock cycles.

The distinction between control and data inputs should also be maintained to internal signals and outputs of the module, so that the significance can only be propagated with data. For this purpose, the module is subjected to some necessary transformations as described below.

5.2.1 Automatic transformations

The *transformations* enable the original module to carry out the significance computations. The transformation process involves following three steps in a sequence:

1. Control-data slicing
2. Logic synthesis of data slice
3. Significance extension by typesetting

The original module marked as block Ⓐ in Figure 5.3 is to be subjected for significance calculations. However, to avoid the complexity of extending VHDL semantics at RTL and keep the accuracy of the analysis, we synthesize the given RTL model to boolean equations and registers.

a) Slicing and logic synthesis:

For significance extension at gate level, we need to keep the separation between 'control' and 'data' signals. This is achieved by declaring all the signals with customized types named as `control` for control signals and `data` for data signals. The signal with `control` type can attain one of 2 boolean values {`F,T`} and a signal with `data` type can attain one of 4 possible values {`F,T,f,t`}. A customized VHDL package defines the two data types as VHDL *enumerated types*.

The gate level model obtained after logic synthesis is a generic flat description with some default data type (For instance `bit` or `std_logic` in VHDL) for all signals. We need to perform recursive traversals through gate level description to assign distinct `control` type and `data` type for all boolean signals. These traversals are based on similar algorithms used in control-data slicing but for gate level models the complexity of the traversals could be high due to huge size of the gate level models.

In order to minimize the traversals through the gate level model, we reuse our control-data slicing algorithm. Before logic synthesis and significance extension, we can split it into control and data slice, and apply synthesis and significance extension to individual slices. In gate level data slice, consisting of large number of data signals and few control signals, we first set all the signals as `data`, and then perform dependency traversals to set `control` type for those signals which do not depend on data inputs directly or transitively.

Similarly in control slice, types for all signals are intially set as `control`, and later by dependency traversals; we set the type as `data`, for signals depending on the data signals coming from data slice. In case of data independent control slice, we do not need to perform the logic synthesis of control slice because it would not contain any data signal to be extended with significance. Therefore, RTL control slice is kept intact, and logic synthesis and traversals are needed only in data slice.

In this way, slicing avoids large number of signals to typeset in the gate level description. Withing individual gate level slices only a few number of signals are needed to be set for `control` and `data` types. Thus during the transformation, original RTL model marked as block Ⓐ in Figure 5.3 is first sliced to obtain a *sliced model* marked as block Ⓑ. The data slice within the sliced model is then subjected to the logic synthesis to obtain the *gate level model* as marked by block Ⓒ in Figure 5.3.

The transformation process is automatic in the verification framework thanks to control-data slicing algorithm, logic synthesis algorithm, static dependency analysis [8], and pattern matching routines for setting `control` and `data` types in the model. The use of slicing during transformation is illustrated by following example.

Illustration of transformation We consider a trivial module foo as shown in Figure 5.4 to illustrate the transformation steps. As shown in circuit diagram on left side of the figure, an input `DSI` selects either to load a new value `D` or to feed back the previous value in internal register `Q`. Input `RST` initializes the register. `Q` is the data output of the module. An output `DSO` becomes high whenever a new data is loaded to the data register `Q` in the previous clock cycle. `RST` and `DSI` are labeled as control inputs whereas `D` is data input.

A distinction between control and data is also depicted in the RTL circuit diagram by a dotted line in Figure 5.4. This distinction must be preserved while applying significance computation. However, when we directly synthesize and regenerate the gate level description, this distinction is lost.

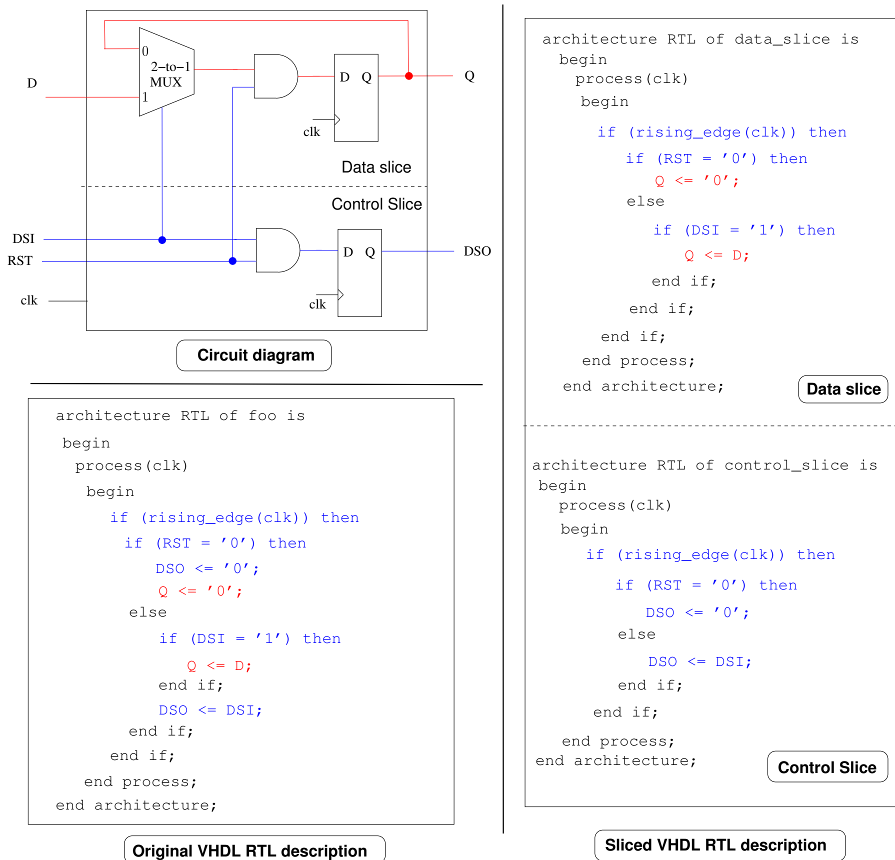


Figure 5.4: Module foo: Circuit diagram, original and sliced RTL descriptions

Synthesis of RTL description of Figure 5.4 results a gate level description given in Figure 5.5 as VHDL architecture. This description is a flat network of interconnected gates and flip-flops with intermediate signals of a default type `bit`, as shown at line 2 in Figure 5.5. We keep

the distinction between the control and data signals by setting different types as `control` and `data`. Therefore, a traversal from the inputs towards output through the network of gates is required. In this example, `RST` and `DSI` being control signal constitute signal `r2` at line 9 to be typed as `control`. Similarly, `n_0` at line 6 should also be of type `control`, which is being evaluated from pure control signal. Rest of the signals should be typed as `data`. We see that for module *foo*, we need to traverse the description for 2 control signals. The detection of such control signals, and setting their types becomes complex in large gate level models with large number of signals.

```
1. architecture gate_level of foo is
2.  signal m1,m2,n_0,k,r1,r2 : bit;
3. begin
4.   m1 <= D and DSI;
5.   m2 <= Q and n_0;
6.   n_0 <= not (DSI);
7.   k <= m1 or m2;
8.   r1 <= k and reset;
9.   r2 <= DSI and reset;
10.  inst: entity DFF(RTL) port map(r1,Q);
11.  inst: entity DFF(RTL) port map(r2,DSO);
12.end architecture;
```

Figure 5.5: Gate-level description for module *foo*

The complexity of this traversal can be reduced if we synthesize only the RTL data slice. The sliced version of the original *foo* description is shown in Figure 5.4. We keep the control slice intact and subject the data slice of *foo* for synthesis. The synthesized data slice description is shown in Figure 5.6 in which we can see that control signal `r2` is no more present in the data slice because it has been retained in control slice by the slicing algorithm. Therefore, traversal for typesetting signal `r2` is avoided in this case. We only need to typeset one signal i.e. `n_0` as `control` in the synthesized data slice of Figure 5.6 instead of two signal as compared to that in original synthesized module *foo*.

```
1. architecture gate_level of Data_slice is
2.  signal m1,m2,n_0,k,r1 : bit;
3. begin
4.   m1 <= D and DSI;
5.   m2 <= Q and n_0;
6.   n_0 <= not (DSI);
7.   k <= m1 or m2;
8.   r1 <= k and reset;
9.  inst: entity DFF(RTL) port map(r1,Q);
10.end architecture;
```

Figure 5.6: Gate-level description for Data slice of module *foo*

In this way the complexity of traversing the gate level source code of the module for setting distinct types for *control* and *data* signals has been reduced by the assistance of slicing algorithm.

b) Significance extension by typesetting

By setting different type for control and data signals, we implicitly extend the model with significance. The gate level model with distinct data and control signals is capable of carrying significance information according to the propagation rules defined in significance semantics. The significance extended model as marked by block Ⓓ of Figure 5.3 preserves the existing behavior of the original module. However, it also contains additional logic to perform significance calculations. Significance semantic rules defined for basic boolean functions (AND, OR and NOT) in the form of truth tables are used to determine the propagation of significance from inputs towards outputs. Static and dynamic semantic rules mentioned in precedent chapter are currently integrated in the verification framework.

A generic significance extended model A generic significance extended model obtained from original module of Figure 5.1 is shown in Figure 5.7. Here $Data'_{in}$ and $Data'_{out}$ are the significance extended inputs and outputs that carry concrete value and significance. The size of these interfaces is twice than those of given model of Figure 5.1. For instance a 4-bit data input of original module will be represented by 8-bit input in significance extended model where each additional bit represents significance.

Size of the data registers is also implicitly twice, and each data register in significance extended model is represented by 2 bits. One bit keeps the original value of the register, and other bit keeps the significance information of that value. Size of the corresponding combinational logic also increases which carry the 4-state signals in significance extended model instead of 2 state signals in the original module. Since control inputs are always considered significant, therefore they do not need significance extension, and are used as they are in original module. Thus DSI and DSO preserve the same role as they have in original module.

The significance extension increases the size of the design if we investigate the effect of individual extended bits. However, logic optimizations [81] are later possible due to unused outputs in the model which do not involve in the property being verified. Similarly, constraints on input values also add simplifications in the resulting model for verification.

5.2.2 Specification of significance property

Once the module is transformed to carry out significance computations, it can be used for verification. For static formal verification, the significance extended model can be directly subjected to model checker and properties concerning significance could be specified in various formalisms such as temporal logic (CTL or LTL) or finite state machines. However, depending upon the significance propagation rules and the property being verified, there are potential logic optimizations possible, which would be useful to reduce the model checking time. Moreover specifying properties concerning significance are tedious to express in temporal logic for engineers who have less knowledge about formal methods as already discussed in chapter 3.

Thus in our prototype implementation, we allow the specifications of significance properties by state machines monitors (observers). State machines are sometimes an easy-to-use way for validation engineers, to specify formal properties instead of CTL or LTL formalisms.

An environment is built semi-automatically which implements *property monitor* [46], [16] to feed data and control inputs to the significance extended model according to the property.

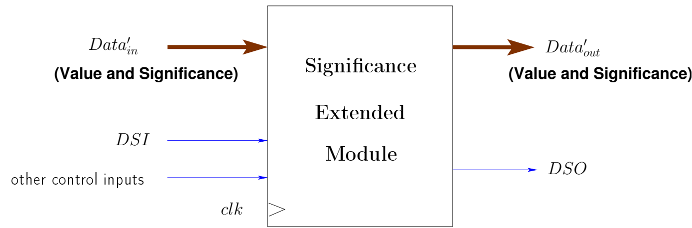


Figure 5.7: Generic significance extended model

The structure of a generic monitor has been shown in Figure 5.8. The block shown in the middle of Figure 5.8 is an instance of the significance extended model. An *input interface* supplies data and control inputs to the module according to the desired property being verified. An *output interface* is designed to observe the output significance, and is implemented according to the specifications given in the property.

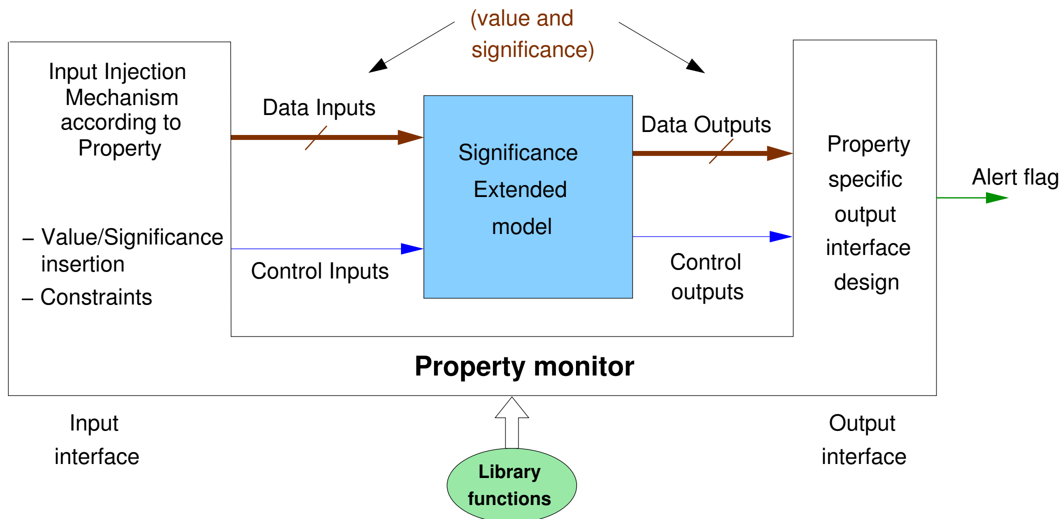


Figure 5.8: Generic property monitor

Structure of a property monitor The monitor feeds value as well as significance to the significance extended model as shown in Figure 5.8 according to the property being verified. The design of input injection mechanism varies according to the properties. The main function of input injection is to convert each input value to a combination of significance and value. Constraints on the inputs are also specified in the input injection mechanism. These constraints can be applied to observe the special cases of the module's behavior. For instance, we can define within the environment that a particular input is always less than a specified threshold value or an input 'x' is always less than an input 'y' or an input is always set to constant. Based on the specified constraints in the environment, the functionality of the module under verification can be made limited which reflects the possibility of simplification of the final module to be verified.

The complexity can be reduced by an abstract consideration of data values. In most of the data computation intensive designs, *bit vectors* are used to represent samples. Instead of considering the significance character of each individual bits in the sample, we consider the entire bit vector is either carries significant value or non-significant value. Therefore, input interface of the monitor is designed in such a way that data input vectors are constrainted to carry either all significant bits or all non-significant bits. This merging mechanism simplifies the significance calculation logic by excluding all those cases in which individual bits have different significance attribute in a bit vector. With this constraint, there are possibilities of redundant paths in the significance calculation logic which can be optimized to obtain a compact model.

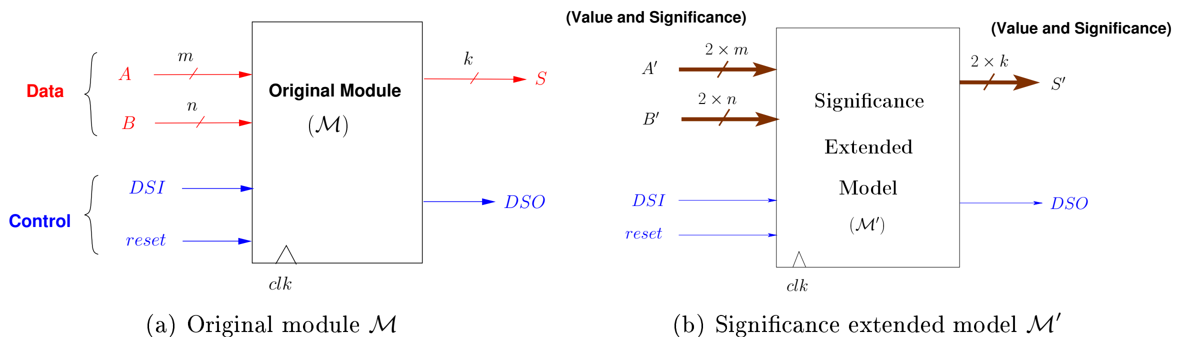
The monitor calculates the output significance as a merged representation for each output data bit vector. Timing information for each data output is also necessary to specify the properties of interest. Depending upon a specific property to be verified there may be different possible ways of implementation of the output interface. For safety properties, the output calculation block can be implemented in such a way that it produces alert flags when safety conditions are violated.

Specifications of input and output interface designs in the monitors is facilitated by some ***library functions*** written in VHDL in our framework. These functions implement significance manipulations on bits and bit vectors at RTL which can readily be called while specifying state machines for monitors. For instance extending a bit (or bit vector) with significance attribute, conversion of a bit (or a bit vector) into significance or non-significant value, merging the significance of a bit vector to a single bit, and getting significance information of a bit vector are the examples of various operations implemented as VHDL library functions in the framework to facilitate the monitor implementation.

We will give the illustration of a verification environment setup using monitor by an example property.

A property monitor

Figure 5.9(a) shows a module given for verification. We suppose that A and B are n-bit, and m-bit data inputs of the module respectively. Let DSI be the control input such that when DSI is high then inputs A and B are sampled by the module for computation. A control input $reset$ is provided to clear the internal registers. Let S be a k-bit data output of the given module, and DSO indicates the moment when valid result is available at S.



(a) Original module $\mathcal{M}$ (b) Significance extended model $\mathcal{M}'$

Figure 5.9: Module $\mathcal{M}$ under verification

After slicing and pre-synthesis of module $\mathcal{M}$ of Figure 5.9(a), we typeset the signals, and apply significance propagation rules to obtain the significance extended model $\mathcal{M}'$ as shown in Figure 5.9(b). Here A' and B' are significance extended data inputs, and S' is significance extended data output. Since A is n-bit, and B is m-bit vector in the original module, therefore in significance extended module the size of input vector grows to $2 \times n$ and $2 \times m$ respectively.

We want to verify following property of this module:

Property 5.1.
"IF 'A' AND 'B' ARE SIGNIFICANT WHEN 'DSI' IS ACTIVE THEN OUTPUT 'S' IS SIGNIFICANT WHEN 'DSO' IS ACTIVE."

This property practically addresses the notion of dependence of a significant output on the significant inputs as theoretically discussed in previous chapter. The property is also important to check the functional correctness of a module because it specifies that a significant output should not be calculated from the non-significant inputs when they are being used in the computation.

A monitor for model $\mathcal{M}'$ is shown in Figure 5.10 for verification of the property 5.1. It contains an instance of the significance extended module $\mathcal{M}'$ which is being fed with data values along with significances. Outputs are being calculated according to the specified property. The input and output interfaces designed according to property 5.1, are explained as below.

Input interface design for $\mathcal{M}'$: The monitor takes normal data values from the primary inputs A and B. Size of these inputs is same as that of corresponding inputs A and B in original module $\mathcal{M}$. The monitor uses two library functions: $to_sig()$ and $to_nonsig()$. Both functions perform a conversion in which n-bit value representation is encoded into $2 \times n$-bit representation which carries significance information along with the data value.

Library function $to_sig()$ converts a given bit vector into significant value whereas $to_nonsig()$ converts the given vector into non-significant value. Both functions implicitly implement the constraint on the inputs in which all the bits in bit vectors A and B are either set as significant or non-significant.

DSI is the control signal which indicates when should the inputs be loaded to the module. In this implementation it has been ensured that only significant inputs are loaded to the module (via $to_sig()$ library function) when DSI is active. When DSI is not active then only non-significant inputs are loaded (via $to_nonsig()$ function) with the help of 2-to-1 multiplexer. In this way the monitor of Figure 5.10 implements an aggressive behavior of the environment to verify property 5.1.

We have only shown the input injection mechanism for data input A' in figure. Exactly similar mechanism is implemented for input B' as well.

Output interface design for $\mathcal{M}'$: Since property 5.1 is only concerned about the significance of the output S', therefore value of S' is not used. The $2 \times k$-bit data output vector S' is merged to a single bit representing either significant or non-significant result according to following rule.

> "*If any bit in the bit vector is non-significant then whole bit vector is considered as non-significant*".

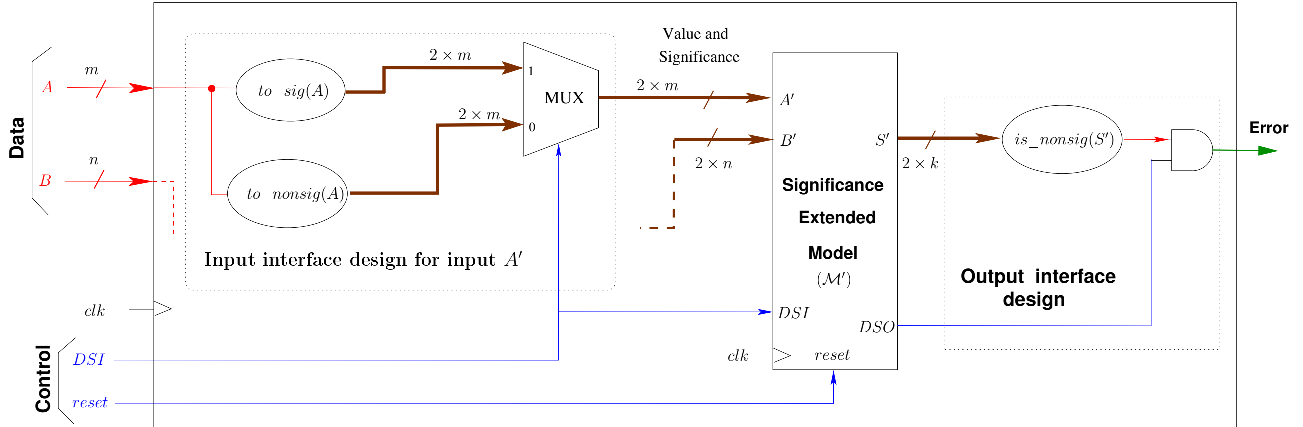


Figure 5.10: Monitor: property 5.1 for model $\mathcal{M}'$

This mechanism is achieved by a library function *is_nonsig*() in Figure 5.10. This function takes a significance extended bit vector as input, and results a boolean signal telling whether the given vector is significant or non-significant. The boolean output of this function becomes high whenever any bit in the data output vector S' is found non-significant.

The monitor generates a boolean output signal '*Error*' as shown in Figure 5.10 which is calculated from *DSO* and output of *is_nonsig*() library function in following way: "*If the DSO signal becomes active, and at the same time, data output S is non-significant then Error becomes active*". This is represented by an AND gate. Thus *Error* signal is an indication of the violation of the property 5.1. To check whether there is any occurrence of this violation in any possible execution of the given model, we can use model checking to verify that *Error* signal never becomes active in this environment.

The specification of property monitors is currently semi-automatic in the verification framework. We have provided some useful library functions in VHDL which facilitate engineers to specify significance properties.

5.2.3 Logic optimization

The significance extended model contains output value calculation logic which is not concerned with the significance oriented property being verified. Moreover, there are possible redundant paths due to constraints on the inputs in the verification environment. Thus before subjecting the final model checking step we remove the unused logic in the module so that the module can be simplified.

In our framework, the automatic simplification of the whole model with the specified monitor, is achieved via logic optimization algorithms. The unused logic is minimized, and undriven registers are deleted thanks to logic optimization algorithms implemented in logic synthesizers. Once the verification environment has been set up for the module according to the claimed property, it is subjected to optimizations which detects sequential and combinational logic elements in the model not being used in the calculation of output significance. This results in an *optimized model* as shown by block Ⓕ in Figure 5.3.

5.2.4 Property-checking for optimized model

The optimized model would be compact because it would only contain the significance information of the data outputs. For model checking, the optimized model is fed to the model checker as shown by the last step in Figure 5.3. To verify whether property 5.1 holds in all states of the system along all possible executions. We can model-check the error signal $Error$ via CTL invariant $AG(Error = 0)$ which specifies that $Error$ never becomes active.

The assertion of $Error$ signal will be a true error indication in case of static semantics. If A and B were significant at the time of DSI and property is violated, then the module is possibly faulty and induces junk (non-significant) values to the result. However, if the non-significant value is not expected to be appeared at the output due to some special operation (such as multiplication by zero) the static semantics could assert the $Error$ signal which would be a false alarm. In case of dynamic semantics however, such false alarm will not occur because dynamic semantics are more powerful than static semantics.

5.3 Illustration by example: Equivalence function

In this section we will illustrate the property verification process in the proposed framework by a small example. We have considered a trivial example of 2-bit equivalence function (XNOR function) to provide an illustration of the property monitor implementation , and elimination of unused module elements according to the specified property being model-checked.

5.3.1 Module description

The circuit diagram of equivalence function module named $\mathcal{M}_e$ is shown in Figure 5.11. It consists of 2 single bit data inputs A and B, and a single bit data output S. The data processing takes two clock cycles. A control input DSI indicates that A and B are loaded into the module. Control output DSO is asserted 2 cycles after DSI is asserted to indicate that outputs S is ready. Control slice and data slice are separately shown in the circuit diagram of Figure 5.11 by a dotted line. There are 5 registers in this module.

5.3.2 Significance extension and verification

We want to verify property 5.1 for module $\mathcal{M}_e$ of Figure 5.11. For this purpose, we transform it via synthesis and typesetting, to obtain the significance extended model as shown in Figure 5.12. Inputs A' and B' are 2-bit inputs with one bit carrying value, and other bit carrying significance. The data registers RA, RB and RS of original module are implicitly replaced by 2-bit registers to carry 4-valued signals by setting the type as `data`. Registers $RA0$, $RB0$ and $RS0$ keep the existing value whereas registers $RA1$, $RB1$ and $RS1$ shown in dark color keep the significance information of the values in $RA0$, $RB0$, and $RS0$ respectively.

Let static semantics is applied for the significance computation. According to static semantics the significance for 2-bit XOR function can be represented by an AND function as it states that "*output will be significant if both inputs are significant*". Therefore, $RA1$ and $RB1$ are operated by an AND gate as shown in Figure 5.12 to produce the significance of output S. The significance of S goes to the additional registers $RS1$. The output S' of the significance extended module is the 2-bit packet carrying contents of registers $RS0$ and $RS1$, the value as well as significance.

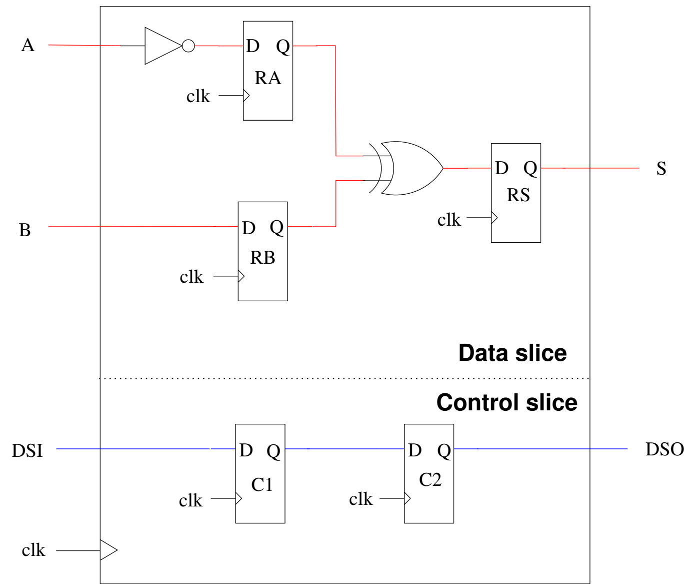


Figure 5.11: Equivalence function model: $\mathcal{M}_e$

The control slice remains intact in $\mathcal{M}'_e$. Total number of registers in significance extended model becomes 8.

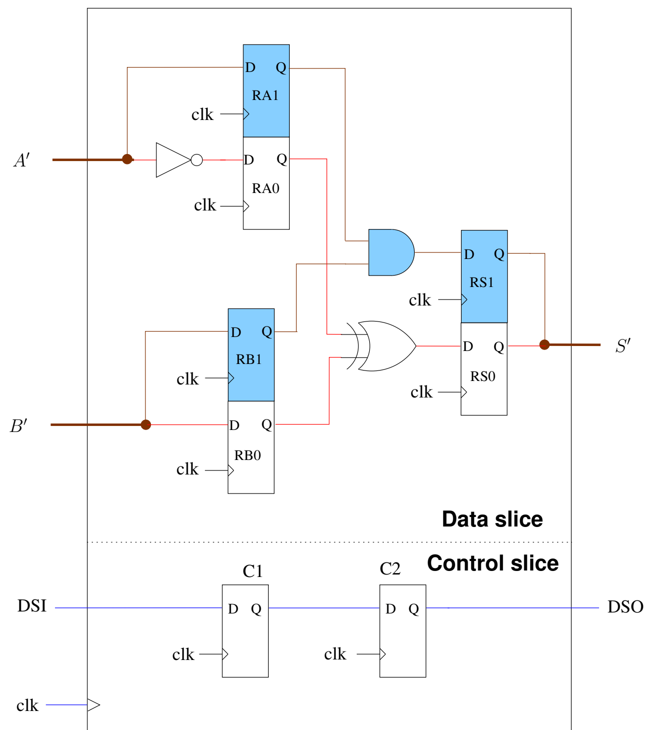


Figure 5.12: Significance extended module $\mathcal{M}'_e$ of Equivalence function

The extension of registers and corresponding combinational logic in the significance extended module as shown in Figure 5.12 is automatic. The user does not need to add additional significance logic. Instead, this is implicitly done by the synthesizer which derives the significance computation logic from the semantics as given in the form of truth tables in precedent chapter.

The monitor implementing the property 5.1 for module $\mathcal{M}_e$ is depicted in Figure 5.13. Significance extended model $\mathcal{M}'_e$ of Figure 5.12 is instantiated in the environment which implements the monitor. The input mechanism is designed using library functions *to_sig*() and *to_nonsig*(). Both data inputs A and B are converted to A' and B' by these functions. The 2-bit significance extended inputs A' and B' are fed to module $\mathcal{M}'_e$ in such a way that only significant values are ensured on both inputs when DSI is high, otherwise only non-significant values are fed to both inputs. This is implemented by 2-to-1 multiplexers.

On the output side, we are only interested in significance, therefore we consider output from register $RS1$ only, which carries the significance information of the output S. A primary output error signal e becomes high if the output $RS1$ is low (i.e. S is non-significant) and DSI is high as specified in property 5.1. Additional state for output signal e is created by an output register. This is done to make the error signal e a state variable for model checking purpose.

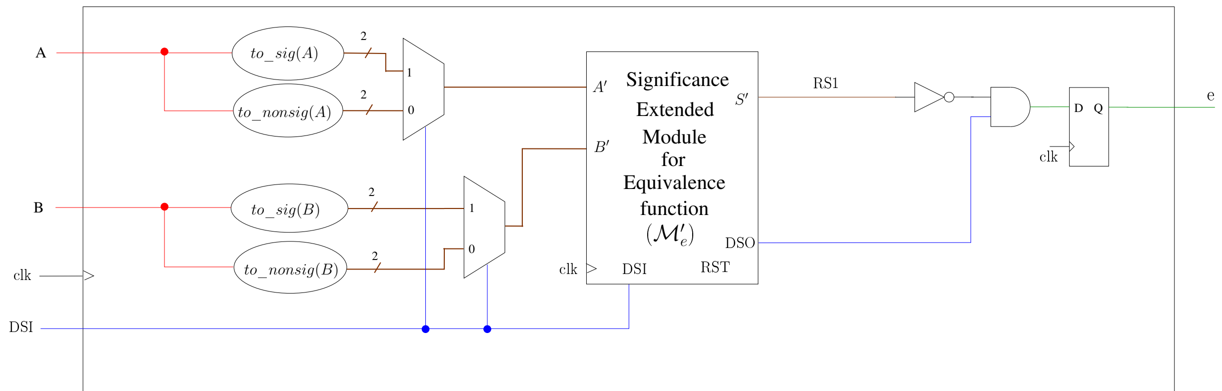


Figure 5.13: Monitor for equivalence function with property 5.1

The verification environment of Figure 5.13 is subjected to logic optimization to remove registers, $RA0$, $RB0$ and $RS0$; and their corresponding combinational logic because they are not being used in the calculation of the output significance and do not drive any primary output of the module. The resulting compact module obtained after logic optimization has been shown in Figure 5.14. In optimized model, the symbolic values `'f'` and `'t'` to represent non-significant boolean character are transformed into some equivalent binary representation. Therefore, logic optimizer simplifies the module in such a way that primary inputs A and B of the environment are not used. Instead the DSI serves to indicate significant or non-significant character of the data signals as shown in circuit of Figure 5.14. DSI is used as *significance input* to the registers $RA1$ and $RB1$ of the data path in such a way that DSI = `'1'` represents a significant value injection to those registers, and DSO = `'0'` represents a non-significant value injection to the registers.

In this way only a significant value (i.e. when DSI = `'1'`) may propagate to the output

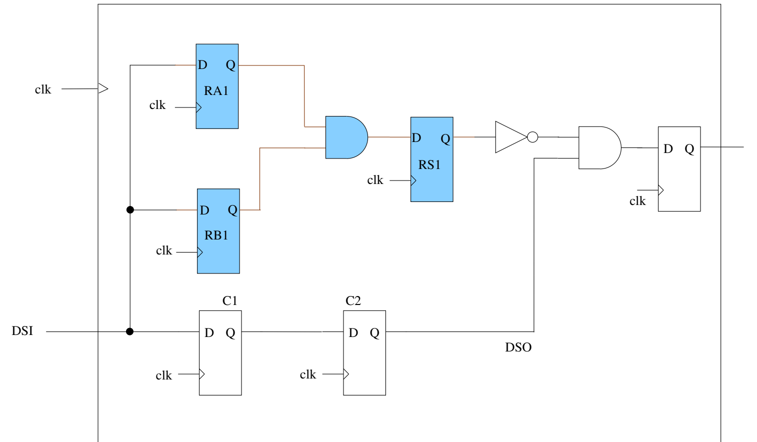


Figure 5.14: Optimized model of equivalence function for property 5.1

significance register $RS1$ after 2 clock cycles when DSO also becomes high at the same time. When DSI is `'0'` then output of register $RS1$ becomes `'0'` after 2 clock cycles indicating that a non-significant value is propagated but at that time, the DSO remains `'0'`. Thus the primary output error signal e never becomes high, and the specified property passes. The simplification process depends on the logic optimization algorithm. The optimization shown in Figure 5.14 was obtained specifically with logic synthesizer we were using. It could be possible that some other logic optimizer would optimize the circuit in a different way.

The optimized model contains 5 registers and less number of logic gates as compared to those in original module. The XOR function would be consisting of AND and OR gates in the original module $\mathcal{M}_e$ which is replaced by a single AND function for significance calculation after minimization. The NOT gate in the original module $\mathcal{M}_e$ is also removed. A few gates are added to specify the property in the environment. However, for larger industrial modules, this does not count as additional surcharge because of a lot of minimization possible in such designs. The optimized model of Figure 5.14 is subjected to model checker with CTL specification: $AG(e = 0)$.

The example of equivalence function considered for illustration of the framework, is trivial and used here to illustrate only different processing steps during the verification process. A considerable simplification and performance can be observed in larger examples as illustrated in experimental results in section 5.5.

5.4 Prototype implementation

We will briefly describe the implementation aspects in the prototype verification framework. Currently we treat synchronous RTL models described in VHDL. VHDL allows rapid prototyping by typesetting distinct *control/data* signals with ease of defining significance propagation rules for boolean operators.

The VHDL slicing tool is used for control and data separation at RTL in the framework which is already presented in chapter 3. We define two distinct data types named `control`, and `data` in a global VHDL package. The control signals typed with a VHDL enumerated type: `control` carry classical symbolic values from the set `{'F','T'}`. The data signals typed with

enumerated type: `data` are able to carry symbolic values from the set `{'F','T','f','t'}` such that `'F'` and `'T'` represent significant values, and `'f'` and `'t'` represent non-significant values. A customized VHDL package defines the two data types as VHDL *enumerated types*. The significance semantics for basic logic operators such as `and`, `or`, `not` and `xor` are also defined for signals of type `control` and `data` in the form of truth tables within custom package.

Significance extension and vector merging is provided in our framework by library functions named *to_sig*() and *to_nonsig*(). They are defined in globally in a VHDL package. Both functions are designed to take a vector of synthesizer generated default data type `bit` as input, and result a vector of type `data`. The property monitor is built in a top level VHDL module which instantiates the significance extended model with input injection mechanism implemented according to specified property.

The commercial synthesis tool *Encounter*® RTL compiler by Cadence Inc.[56] was available to us. It was used to achieve prior synthesis as well as logic optimizations after property specification. The logic optimization algorithms implemented with this synthesis tool perform resource sharing, speculation, mux optimization, carrysave arithmetic (CSA) optimizations, and redundancy removals [22].

There are many commercial (FormalCheck, RuleBase, FomalPro), and academic (NuSMV, PRISM, SPIN, VIS) model checking tools. For experimentation, we relied on academical tools which are freely available. SPIN and PRISM both have their own modeling language to represent the system. NuSMV is aimed for software as well as hardware model checking but it also requires the model to be written in its own language. For rapid prototyping, we needed a model checker which requires no effort of transforming and rewriting the model to be checked.

VIS comes with a frontend to support a subset of Verilog. We have VHDL as an input language, and for our case VIS is considered to be a better option because VHDL to Verilog conversion is supported by the logic synthesis via *Encounter*® RTL compiler. This avoided the time consuming implementation of a tool to transform VHDL into a model checker specific language.

A Verilog HDL front end '*vl2mv*' is provided to compile a subset of Verilog into an intermediate format BLIF-MV [101] [65]. BLIF-MV is a low level hardware description language designed for describing hierarchical hardware circuits. The final optimized gate level Verilog model generated from logic synthesizer is directly fed to VIS for model checking. Specification to check violation of the property is given in CTL or LTL language in a separate file.

5.5 Experimental results

We have tested few examples by verifying property 5.1 mentioned in section 5.2. Results are provided in Table 5.1 for serial parallel multiplier (SPM), greatest common divisor (GCD), input register stage for discrete cosine transform (DCT IR), data encryption standard (DES), and FIR filter with static significance semantics. For each module, the generic property 5.1 is specified according to module's inputs and outputs. For instance in case of GCD with two operands `op1`, `op2`, a control input `load`; and outputs `res` and `done` the property 5.1 takes the form:

> IF '*op*1' AND '*op*2' ARE SIGNIFICANT WHEN '*load*' IS ACTIVE THEN OUTPUT '*res*' IS SIGNIFICANT WHEN '*done*' IS ACTIVE.

Similarly, for other modules the property is represented accordingly. The simplification is obtained in terms of number of logic gates and number of registers. The degree of simplification depends on the nature of the module. For instance in case of SPM, original module contained 146 combinational blocks (Comb. blocks), and 53 registers which are reduced to 62 blocks, and 38 registers after simplification. However, in case of GCD, number of registers are not reduced due to dependencies of output significance on input data values.

IP Modules	**Type**	**Comb. blocks**	**No. of registers**	**Model checking time (sec)**
SPM	Original	146	53	0.383
	Resulting	62	38	
GCD	Original	484	32	0.423
	Resulting	222	64	
DCT IR	Original	286	172	0.332
	Resulting	42	11	
DES	Original	1363	128	1.258
	Resulting	237	128	
FIR	Original	307	208	2.288
	Resulting	219	173	

Table 5.1: Model checking results with static semantics

Results of the modules with same properties using dynamic semantics are shown in Table 5.2. We see that degree of simplifications is less in this case as compared to that in static case. Stars '*' marked with some of the results indicate constraint considerations on inputs while verifying property. For instance, in case of DES, key schedule is fixed to significant. Similarly for FIR filter, coefficients are taken as significant constants. These constraints assist to reduce module's state space by constant propagation during logic optimization. These results are obtained on Intel(R) Pentium(R) 4 CPU at 2.66GHz with 1GB of memory.

IP Modules	**Type**	**Comb. blocks**	**No. of registers**	**Model checking time (sec)**
SPM	Original	146	53	2.15
	Resulting	165	63	
GCD	Original	484	32	0.92
	Resulting	439	64	
DCT IR	Original	286	172	5.974
	Resulting	417	178	
DES*	Original	1363	128	23.52
	Resulting	3527	249	
FIR*	Original	307	208	0.176
	Resulting	63	53	

Table 5.2: Model checking results with dynamic semantics

5.6 Conclusions

We have proposed a formal verification framework for the validation engineers to verify properties which talk about the right data movements at their right timing instants in a hardware module using the concept of *significance*. The framework considers a straightforward implementation with sound significance semantics for rapid prototyping. The incompleteness of significance semantics may cause loss of useful information due to false alarms in the verification process which could be avoided using syntax independent significance computations as discussed in precedent chapter.

Current implementation is a prototype with many improvement to be made yet. Although some library functions are provided to facilitate property specification via monitors, the mechanism is not yet fully automated. Automatic conversion of high level formal specifications such as property specifications language (PSL) [59] or temporal logic specifications into state machines [17], [42], [82] would be integrated in the future.

We are investigating the efficient implementation of *refined significance* and *semantic significance calculus* in the form of reduced ordered binary decision diagrams (ROBDD) [18] which would be integrated within the verification framework in the future.

Chapter 6

Conclusion et perspectives

Nous avons présenté plusieurs techniques pour faciliter la vérification et l'abstraction de modèles d'IP et plus particulièrement des méthodes basées sur le model-checking et la simulation. L'utilisation de techniques d'abstraction de données et d'analyse statique formelle de flots de données a été considérée pour une classe de modules matériels de types chemins de données (datapaths) et contrôle (machines d'etat finis). Nous avons concentré d'abord nos efforts pour définir des techniques de séparation des données du contrôle au cœur des modules matériels de telle sorte que ces techniques soient applicables aux calculs complexes sur les données, tout en laissant le contrôle constant. L'idée fut de garder les comportements temporels critiques dans le modèle abstrait.

Contrairement aux techniques existantes pour séparer le contrôle des données, fondées sur des jugements intuitifs, nous avons essayé de proposer une definition séemantique des notions de contrôle et de données dans les descriptions matérielles. En considérant de manière intuitive que *le contrôle contient l'information sur les instants où les données sont traitées*, nous avons tenté de définir la notion d'entrées de données et de contrôle. Puis nous nous sommes intéressés à l'analyse statique sur les blocs élémentaires dans le but d'identifier automatiquement les entrées de contrôle, sans connaissance a priori du système et sans information complémentaire fournie par l'utilisateur. Cette analyse est indépendante de la représentation syntaxique des modèles.

Cependant, la définition d'entrées de contrôle n'est pas suffisante pour satisfaire la condition d'indépendance syntaxique. Aussi, nous avons conclu qu'une séparation unique du contrôle et des données était illusoire dans le cas général. Une telle séparation dépend trop de l'idée subjective que le concepteur s'est fait du système et également de l'environnement dans lequel le module sera plongé. De ce fait, nous considérons des techniques d'analyse basées sur la syntaxe pour effectuer la séparation contrôle-données, avec pour conséquence de restreindre les applications sur lesquelles ces techniques peuvent être effectivement appliquées.

Nous avons utilisé des techniques de «slicing» pour réaliser cette séparation contrôle-données basées sur l'hypothèse que l'utilisateur identifie les entrées de données du module. Cette information est utilisée comme un critère pour effectuer le slicing. Avec l'algorithme de slicing proposé, nous pouvons séparer la partie contrôle sous la forme d'un «*control slice*» et la partie traitement sous la forme d'un «*data slice*» et des signaux d'interconnexion. Le slicing de modèles comportant des variables locales, implique des transformations supplémentaires pour garder le comportement du module après slicing. La représentation structurelle obtenue avec slicing nous permet d'appliquer des techniques d'abstraction de données sur le «*control*

slice» avec pour résultat des améliorations considérables dans les temps de vérification.

Pour une classe de modèles d'IP avec un contrôle indépendant des données, nous avons proposé une abstraction fonctionnelle des données du «*data slice*» décrit au niveau RTL en le remplaçant par un modèle de calcul fonctionnel. L'abstraction est semi-automatique, basée sur des techniques de reconnaissance de modèle pour détecter le «timing» des entrées/sorties des données. De cette façon, nous obtenons un modèle plus abstrait qui permet une simulation plus rapide que le modèle RTL original. Avec l'assistance à l'abstraction et à la vérification, la séparation contrôles/données peut être aussi utile dans la conception physique, par exemple pour l'estimation de la taille et de la puissance dissipée d'un circuit intégré.

Une notion de flots de données *intentionnels* dans les modèles, nommée *significativité* a été introduite. La significativité, vu comme un attribut supplémentaire associée aux données, reflète les dépendances de données booléennes entre les variables des modèles matériels de bas niveaux. Une analyse statique formelle de flots de données est possible en définissant une sémantique pour la propagation de la significativité dans les équations booléennes des modèles au niveau porte. En considérant seulement la présence ou l'absence de données aux instants spécifié, on peut vérifier formellement grâce à la significativité des propriétés intéressantes des chemins de données concernant l'intention du concepteur. Des règles de propagation de significativité homogènes mais incomplètes ont été proposés avec les descriptions formelles associées. Une implementation directe de ces rêgles a été réalisée. Une description théorique de la propagation de significativité indépendante de la syntaxe est aussi présente dans ce manuscrit. Elle est basée sur le calcul booléen inductif.

Avec la sémantique de la significativité, nous avons réalisé une technique de vérification basée sur le model-checking pour détecter statiquement des erreurs dans les chemins de données. Pour faciliter le travail de validation des ingénieurs, nous utilisons la technique des observateurs pour spécifier des propriétés relatives à la significativité sous forme de machines d'états. L'implémentation des observateurs est facilitée par l'utilisation de fonctions intégrées pour spécifier les propriétés de significativité.

Ce domaine de recherche reste ouvert: Il apparait régulièrement de nouveaux aspects qu'il faut explorer pour identifier de nouvelles catégories de propriétés. De la même manière, notre travail contient plusieurs aspects qu'il faudra étudier dans le futur. Une définition sémantique du *contrôle* est toujours une question ouverte: nous avons essayé de relier le comportement temporel du modèle avec le contrôle; cette notion pourrait-elle être abordée différemment ? L'implémentation d'algorithmes de slicing doit être amélioré avec des règles plus générales pour considérer un ensemble plus large de VHDL et incorporer des modèles RTL de Verilog.

La vérification basée actuellement sur une significativité statique ou dynamique peut entraîner des fausses alertes du fait d'une sémantique de comportement dépendant de la syntaxe. Nous avons proposé une amélioration de la significativité dynamique en introduisant un test de dépendance de données. Mais un environnement de vérification reste à concevoir en utilisant cette sémantique, indépendante de la syntaxe. De même, une représentation efficace et compacte d'équations booléennes de type ROBDD pourrait être utilisée avantageusement pour implémenter cette sémantique.

Nous nous sommes appuyés sur la notion de significance pour représenter les deux possibilités de *presence* et d'*absence* d'une donnée valide sur les signaux. Ce concept peut être étendu pour définir des attributs de signaux suivant de multiples classes. Par exemple dans un module UART, le bus de données pourrait être enrichi en transportant des informations de contrôle en plus des données, en particulier le contenu des registres internes. Une représentation abstraite de ce scénario pourrait être de définir trois attributs associés aux données

d'entrées de types *donnée*, *statut* et *commande*. Ceci pourrait être utile pour représenter les états des signaux en vue de la vérification statique des flots de données; par exemple dans le cas d'études de propriétés relatives à la sécurité du matériel.

Les techniques de vérification et d'abstraction basées sur le slicing, avec la notion de significativité ont été réalisées dans des prototypes semi-automatiques de vérification. Nous pensons qu'ils pourraient être facilement automatisés de façon plus importante pour apporter une aide plus efficace aux ingénieurs dans leur travail de validation. Actuellement nous avons seulement intégrer les modèles VHDL. Le passage aux modèle Verilog ou systemC ne devrait pas poser de problème. Les techniques sous-jacentes pourraient être étendues à tous modèles de description de matériels existants ou avenirs.

Appendix A

A.1 Extended boolean operators

Functions *not*, *and*, *or* and *mux* are the operators on extended boolean of type 'extendedBool'. Extended boolean are 2 bits integers in which LSB holds the standard boolean value (true = 1, false = 0) and the MSB holds the significance (significant = 1, non significant = 0). The implementation of these functions is given below.

```
extendedBool not(extendedBool a) {
  BOOL v, s;

  v = !val(a);
  s = sig(a);
  return toExtendedBool(v, s);
}

extendedBool or(extendedBool a, extendedBool b) {
  BOOL v, s;

  v = (val(a) || val(b));
  s = ttty.or[a][b];
  return toExtendedBool(v, s);
}

extendedBool and(extendedBool a, extendedBool b) {
  BOOL v, s;

  v = (val(a) && val(b));
  s = ttty.and[a][b];
  return toExtendedBool(v, s);
}

extendedBool mux(extendedBool i0, extendedBool i1, extendedBool c) {
  BOOL v, s;

  v = (val(c) && val(i1)) || ((!val(c)) && val(i0));
  s = ttty.mux[i0][i1][c];
  return toExtendedBool(v, s);
}
/*returns the value part of an extendedBool*/
BOOL val(extendedBool a) {
  return a & 0x1;
}
/*returns the significance part of an extendedBool*/
BOOL sig(extendedBool a) {
  return (a >> 1) & 0x1;
}

extendedBool f(extendedBool i0, extendedBool i1, extendedBool c) {
  return or(and(i0, not(c)), and(i1, c));
}
```

The function setSemantics(int operator, int semantics) initializes the truth table of an operator. Operators are passed as macros (AND, OR or MUX). Semantics are integers whose meanings are different for different operators. For AND (and correspondingly for OR), the semantics is a 10 bits integer defining the significance of the formula $AND(a, b)$ for the 10 different valuations of a, b where a and b are 'extendedBool' (in 0, 1, 2, 3) and b >= a (the AND operator commutes, so $AND(a, b) = AND(b, a)$. For MUX the semantics is an 8 bits integer, defining (from LSB to MSB) the significance of MUX(i0, i1, c) for the valuations is shown in Table A.1 where '0n' represents non-significant 0 value corresponding to `f`, and '0s' represents significant 0 value corresponding to `F`.

MUX(0n, 0n, 0n)
MUX(0n, 0n, 0s)
MUX(0n, 0s, 0n)
MUX(0n, 0s, 0s)
MUX(0s, 0n, 0s)
MUX(0s, 0n, 0n)
MUX(0s, 0s, 0s)
MUX(0s, 0s, 0n)

Table A.1: MUX valuations

The other valuations are computed from this table and from elementary properties of MUX:

- output significance does not depend on i0 and i1 value
- $MUX(i0, i1, c) = MUX(i1, i0, NOT(c))$

The implementation of function is given as follows:

```
truthTables ttty;

void setSemantics(int op, int semantics) {
  int n, i, j;

  switch(op) {
    case AND:     /* Construct turth table for AND */
                  n = semantics;
                  for(i = 0; i < 4; i++) {
                    for(j = i; j < 4; j++) {
                      ttty.and[i][j] = n & 0x1;
                      ttty.and[j][i] = n & 0x1;
                      n >>= 1;
                    }
                  }
                  break;
    case OR: /* Construct turth table for OR */
                 n = semantics;
                 for(i = 0; i < 4; i++) {
                   for(j = i; j < 4; j++) {
                     ttty.or[i][j] = n & 0x1;
                     ttty.or[j][i] = n & 0x1;
                     n >>= 1;
                   }
                 }
                 break;
```

```
    case MUX: /* Construct turth table for MUX */
              n = semantics;
              for(i = 0; i < 2; i++)
                for(j = 0; j < 2; j++) {
                  ttty.mux[i][j][0] = n & 0x1;
                  ttty.mux[j][i][1] = n & 0x1;
                }
              n >>= 1;
              for(i = 0; i < 2; i++)
                for(j = 0; j < 2; j++) {
                  ttty.mux[i][j][2] = n & 0x1;
                  ttty.mux[j][i][3] = n & 0x1;
                }
              n >>= 1;
              for(i = 0; i < 2; i++)
                for(j = 0; j < 2; j++) {
                  ttty.mux[i][j+2][0] = n & 0x1;
                  ttty.mux[j+2][i][1] = n & 0x1;
                }
              n >>= 1;
              for(i = 0; i < 2; i++)
                for(j = 0; j < 2; j++) {
                  ttty.mux[i][j+2][2] = n & 0x1;
                  ttty.mux[j+2][i][3] = n & 0x1;
                }
              n >>= 1;
              for(i = 0; i < 2; i++)
                for(j = 0; j < 2; j++) {
                  ttty.mux[i+2][j][0] = n & 0x1;
                  ttty.mux[j][i+2][1] = n & 0x1;
                }
              n >>= 1;
              for(i = 0; i < 2; i++)
                for(j = 0; j < 2; j++) {
                  ttty.mux[i+2][j][2] = n & 0x1;
                  ttty.mux[j][i+2][3] = n & 0x1;
                }
              n >>= 1;
              for(i = 0; i < 2; i++)
                for(j = 0; j < 2; j++) {
                  ttty.mux[i+2][j+2][0] = n & 0x1;
                  ttty.mux[j+2][i+2][0] = n & 0x1;
                }
              n >>= 1;
              for(i = 0; i < 2; i++)
                for(j = 0; j < 2; j++) {
                  ttty.mux[i+2][j+2][2] = n & 0x1;
                  ttty.mux[j+2][i+2][3] = n & 0x1;
                }
              break;
    default: printf("setSemantics error: undefined operator\n");
             exit(-1);
             break;
  }
}
```

A.2 Constraint system for MUX

The seach_mux_semantics() is the utility function which exhaustively searches semantics of $MUX(i0, i1, c)$ such that the following constraints are met:

- There must be a pair $(i0, i1)$ such that the value of c changes the significance of $MUX(i0, i1, c)$.

- An i0 or i1 value change shall not change the output significance.
- $MUX(i0, i1, c) = MUX(i1, i0, NOT(c))$.
- If all inputs are significant, output must be significant.
- If all inputs are non significant, output must be non significant.
- Changing the significance of any input cannot changes the output significance in the opposite way.

For each of the found semantics, the function seach_mux_semantics() prints the truth table of MUX.

```
int seach_mux_semantics() {
  int m, cnt, ok;
  extendedBool i0, i1, c;

  for(m = 0; m < 256; m++) {
    setSemantics(MUX, m);
    ok = 0;
    for(cnt = 0; cnt < 64; cnt++) {
      i0 = (cnt >> 4) & 0x3; /* input value for i0 */
      i1 = (cnt >> 2) & 0x3; /* input value for i1 */
      c = cnt & 0x3;         /* input value for c */

      /* There must be a pair (i0, i1) such that the value of c changes the
       * significance of mux(i0, i1, c) */
      if(sig(mux(i0, i1, c)) != sig(mux(i0, i1, not(c))))
        ok = 1;
      /* An i0 or i1 value change shall not change the output significance */
      if((sig(mux(i0, i1, c)) != sig(mux(not(i0), i1, c))) ||
           (sig(mux(i0, i1, c)) != sig(mux(i0, not(i1), c))))
        break;
      /* mux(i0, i1, c) = mux(i1, i0, not(c)) */
      if(mux(i0, i1, c) != mux(i1, i0, not(c)))
        break;
      /* If all inputs are significant, output must be significant */
      if(sig(i0) && sig(i1) && sig(c) && (!sig(mux(i0, i1, c))))
        break;
      /* If all inputs are non significant, output must be non significant */
      if((!sig(i0)) && (!sig(i1)) && (!sig(c)) && sig(mux(i0, i1, c)))
        break;
      /* Changing the significance of any input cannot change the output
       * significance in the opposite way */
      if(sig(mux(i0, i1, c)) && ((!sig(mux(sig1(i0), i1, c))) ||
              (!sig(mux(i0, sig1(i1), c))) || (!sig(mux(i0, i1, sig1(c))))))
        break;
      if((!sig(mux(i0, i1, c))) && (sig(mux(sig0(i0), i1, c)) ||
              sig(mux(i0, sig0(i1), c)) || sig(mux(i0, i1, sig0(c)))))
        break;
    }
    if((cnt != 64) || !ok)
      continue;
    printf("MUX #%d\n", m);     /* Truth table printing */
    printMUXTruthTable();
    }
  return 0;
}
```

A.3 Constraint system for AND

Utility function search_AND() exhaustively searches semantics of AND such that the following constraints are met:

- MUX has the semantics # 160 (setSemantics(MUX, 160)) stating that *output is significant if and only if the selector and the selected input are significant.*
- $OR(a, b) = NOT(AND(NOT(a), NOT(b)))$.
- $MUX(i0, i1, c) = f(i0, i1, c)$ where f is the function defined in section A.1.

The search explores all the semantics of AND and computes the distance between MUX and f, that is the number of valuations of i0, i1, c (64 cases) for which $MUX(i0, i1, c)\,! = \; f(i0, i1, c)$. It prints the minimum distance and all the semantics with minimum distance. printTruthTable can then be used to print the truth table of the found semantics for AND.

```
struct recorder;

struct recorder {
  int semantics;
  struct recorder * next;
  struct recorder * previous;
};

int search_AND() {
  int a, o, d, min, cnt;
  extendedBool i0, i1, c;
  struct recorder rec, * p;

  setSemantics(MUX, 160);
  min = 64;
  p = &rec;
  p->next = (struct recorder *)(NULL);
  p->previous = (struct recorder *)(NULL);
  for(a = 0; a < 1024; a++) {
    setSemantics(AND, a);
          /* Value loading for OR function */
    o = ((a >> 4) & 0x1) | (((a >> 1) & 0x1) << 1) | (((a >> 6) & 0x1) << 2) |
      (((a >> 5) & 0x1) << 3) | ((a & 0x1) << 4) | (((a >> 3) & 0x1) << 5) |
      (((a >> 2) & 0x1) << 6) | (((a >> 9) & 0x1) << 7) | (((a >> 8) & 0x1) << 8)
      | (((a >> 7) & 0x1) << 9);
    setSemantics(OR, o);
    d = 0;
    for(cnt = 0; cnt < 64; cnt++) {
      i0 = (cnt >> 4) & 0x3;   /* input value loading for i0 */
      i1 = (cnt >> 2) & 0x3;   /* input value loading for i1 */
      c = cnt & 0x3;           /* input value loading for c */

      if(f(i0, i1, c) != mux(i0, i1, c)) {
        d += 1;
        if(d > min)
          break;
      }
    }
    if(d < min) {
      min = d;
      while(p->previous != NULL) {
        p = p->previous;
        free(p->next);
      }
      p->semantics = a;
      p->next = NULL;
```

```
    }
    else if(d == min) {
      p->next = (struct recorder *)(calloc(1, sizeof(struct recorder)));
      if(p->next == NULL) {
        printf("calloc error\n");
        exit(-1);
      }
      p->next->previous = p;
      p = p->next;
      p->semantics = a;
      p->next = NULL;
    }
  }
  for(; p != NULL;) {
    printf("AND #%d: %d\n", p->semantics, min);
    p = p->previous;
    if(p != NULL)
      free(p->next);
  }
  return 0;
}
```

The search functions for MUX and AND can easily be customized to explore other constaints.

Appendix B

B.1 VHDL descriptions for Serial parallel multiplier (SPM)

```
1. library IEEE;
2. use IEEE.NUMERIC_BIT.all;

3. entity SPM is
4.  port(CLK, Reset: in Bit;
5.       A, B: in Unsigned(7 downto 0);
6.       S: out Unsigned(15 downto 0);

7.       Load: in Bit;
8.       DSO: out Bit);
9. end entity SPM;

10. architecture RTL of SPM is
11. begin

12.  MAIN: process
13.    variable RA, RB: Unsigned(7 downto 0);
14.    variable RR: Unsigned(15 downto 0);
15.    variable CNT: Natural range 0 to 9;
16.  begin
17.   if (CLK = '1' and CLK'event) then
18.    if (Reset = '0') then
19.      s <= (others => '0');
20.      DSO <= '0';
21.      CNT := 0;
22.    else
23.      if (Load = '1') then
24.        RA := A;
25.        RB := B;
26.        RR := (others => '0');
27.        CNT := 1;
28.        DSO <= '0';
29.      else
30.      if (CNT > 0 and CNT < 9) then
31.        RR := '0' & RR(15 downto 1);
32.        if (RB(0) = '1') then
33.          RR(15 downto 7) :=
               RR(15 downto 7) + ('0' & RA);
34.        end if;
35.        RB := '0' & RB(7 downto 1);
36.         CNT := CNT + 1;
37.  end if;
38.       if (CNT = 9) then
39.          DSO <= '1';
40.          S <= RR;
41.      end if;
42.     end if;
43.    end if;
44.   end if;
45.   wait on CLK;
46.  end process MAIN;
47. end architecture RTL;
```

Figure B.1: Original SPM implementation (before slicing)

```
1. Data_Process_2: process
2.   variable RA : unsigned(7 downto 0);
3.   variable RB : unsigned(7 downto 0);
4.   variable RR : unsigned(15 downto 0);
5. begin
6.    S_2 <= s_1;
7.    RA := RA_1;
8.    RB := RB_1;
9.    RR := RR_1;
10.   if (Reset = '0') then
11.       S_2 <= ( others => '0' ) ;
12.   else
------------------------------------------------------------------------
13.       if (Load = '1') then                --<----------- Data acquisition phase
14.          RA := A;
15.          RB := B;
16.          RR := ( others => '0' ) ;
------------------------------------------------------------------------
17.       else
18.          if ((cnt1 > 0) and (cnt1 < 9)) then   --<----Data processing starts
19.             RR := '0' & RR ( 15 downto 1 ) ;
20.             if (RB(0) = '1') then
21.               RR ( 15 downto 7 ) := "+"(RR ( 15 downto 7 ) , '0' & RA) ;
22.             end if;
23.             RB := '0' & RB ( 7 downto 1 ) ;    --<------ Data procesing ends
24.          end if;
------------------------------------------------------------------------
25.           if (cnt2 = 9) then
26.             S_2 <= RR;                        --<------ Output phase
27.          end if;
28.       end if;
29.    end if;
30.    RA_2 <= RA;
31.    RB_2 <= RB;
32.    RR_2 <= RR;
33.    wait on S_1,RA_1,RB_1,RR_1,Reset,Load,A,B,cnt1,cnt2;
34. end process;
```

Figure B.2: VHDL process containing data processing of SPM (regenerated after slicing)

```
1. package spm_pkg is
2. function fast_spm( A : IN integer; B:IN integer ) return integer;
3.    attribute foreign of fast_spm : function is "fast_spm␣./spm.sl";
4. end;

5. package body spm_pkg is
6.   function fast_spm( A : IN integer; B:IN integer ) return integer is
7.   begin
8.       assert false report "ERROR:␣foreign␣subprogram␣not␣called" severity note;
9.      return 0;
10.   end;
11.end;
```

Figure B.3: VHDL Subprogram declaration to interface with foreign function

```
1. Abstract_data_Process: process(clk)
2.   variable ra,rb : unsigned(7 downto 0);
3. begin
4.   if(rising_edge(clk) then
5.     if (reset = '0') then    --<---------- "Reset state"
6.        s <= (others =>'0');
7.        ra := (others =>'0');
8.        rb := (others =>'0');
------------------------------------------------------------------
9.     elsif (load = '1') then    --<------------ "Data acquisition" phase
10.       ra := a;
11.       rb := b;
------------------------------------------------------------------
12.    else
13.       if (cnt = 9) then    --<--- "untimed functional computation" and "Output" phases
14.         s <= To_Unsigned(fast_spm(to_integer(ra),to_integer(rb)),16);
15.       end if;
------------------------------------------------------------------
16.    end if;
17.  end if;
18.end process;
```

Figure B.4: Functional data computation in data slice

```
1. #include <stdio.h>
2. #include "mti.h"
3. int fast_spm(int a,int b)
4. {
5.    return a * b;
6. }
```

Figure B.5: Fast implementation of SPM

Bibliography

[1] Open SystemC initiative (OSCI) website: `http://www.systemc.org`.

[2] PCCTS Resources : `http://www.polhode.com/pccts.html`.

[3] The VIS Home Page : `http://vlsi.colorado.edu/~vis/`.

[4] *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*, 1993.

[5] *AIRE/CE, Advanced Intermediate Representation with Extensibility/Common Environment*, 1999. Internal Intermediate Representation (IIR) Specification Version 4.6 Including Digital VHDL and VHDL AMS support.

[6] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. "FoCs: Automatic Generation of Simulation Checkers from Formal Specifications". In *CAV'2000: Proceedings of 12th International Conference on Computer Aided Verification*, pages 538–542, 2000.

[7] V. Agrawal and K. Cheng. "Finite State Machine Synthesis with Embedded Test Function". *Journal of Electronic Testing*, 1(3):221–228, 1990.

[8] A. Aho, R. Sethi, and J. Ullman. *"Compilers: Principles, Techniques and Tools, Code Optimization"*. Addison-Wesley Publishing Company, 1986.

[9] Zaher S. Andraus and Karem A. Sakallah. "Automatic Abstraction and Verification of Verilog Models". In *DAC'04: Proceedings of the 41st annual conference on design automation*, pages 218–223. ACM Press, 2004.

[10] S. Ben-David, C. Einser, D. Geist, and Y. Wolfsthal. "Model Checking at IBM". In *"Formal Verification and Testing Technologies in System Design"*. Kluwer Academic Publisher, 2003.

[11] G. Berry. "Hardware and Software Synthesis, Optimization, and Verification from Esterel Programs". In *TACAS '97: Proceedings of the 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 1–3, London, UK, 1997. Springer-Verlag.

[12] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. *Tools and Algorithms for the Construction and Analysis of Systems*, chapter "Symbolic Model Checking without BDDs", pages 193–207. Lecture Notes in Computer Science. Jan. 1999.

[13] C. Bolchini and L. Baresi. "Software Methodologies in VHDL Code Analysis". *EUROMICRO Journal of System Architectures*, 44(1):3–21, 1997.

[14] N. Bombieri, F. Fummi, and G. Pravadelli. "A Methodology for Abstracting RTL Designs into TL Descriptions". In *MEMOCODE'06: Proceedings of 4th ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 103–112, 2006.

[15] D. Borrione, J. Dushina, and L. Pierre. "Formalization of Finite State Machines with Data Path for the Verification of High-level Synthesis". In *Proceedings of 11th Brazilian Symposium on Integrated Circuit Design*, pages 99–102, 1998.

[16] D. Borrione, L. Miao, K. Morin-Allory, P. Ostier, and L. Fesquet. "On-line Assertion-based Verification with Proven Correct Monitors". In *ITI 3rd International Conference on Information and Communications Technology, Enabling Technologies for the New Knowledge Society*, pages 125–143, 2005.

[17] M. Boulé and Z. Zilic. "Automata-based Assertion-checker Synthesis of PSL Properties". *ACM Transactions on Design Automation of Electronic Systems*, 13(1):1–21, 2008.

[18] K.S. Brace, R.L. Rudell, and R.E. Bryant. "Efficient Implementation of a BDD Package". In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 40–45. ACM, 1990.

[19] R.K. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. "VIS: a System for Verification and Synthesis". In *CAV'96: Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102, pages 428–432, New Brunswick, NJ, USA, 1996. Springer Verlag.

[20] R.E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[21] R.E. Bryant, S. Lahiri, and S. Seshia. "Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 78–92. Springer-Verlag, 2002.

[22] Cadence Design Systems Inc. *Using Encounter® RTL Compiler*, 2008.

[23] G. Canfora, A. Cimitile, and A. De Lucia. "Conditioned Program Slicing". *Information and Software Technology, Special issues on Program Slicing*, 40(11-12):595–607, 1998.

[24] D.I. Cheng, K. Cheng, D.C. Wang, and M. Marek-Sadowska. "A New Hybrid Methodology for Power Estimation". In *DAC'96: Proceedings of the 33rd annual conference on Design automation*, pages 439–444, New York, NY, USA, June 1996. ACM Press.

[25] E.M. Clarke, M.C. Browne, E.A. Emerson, and A.P. Sistla. "Using Temporal Logic for Automatic Verification of Finite State Systems". pages 3–26, 1985.

[26] E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[27] E.M. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. "Program Slicing for Design Automation: An Automatic Technique for Speeding-up Hardware Design, Simulation, Testing, and Verification". In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[28] E.M. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. "Program Slicing of Hardware Description Languages". In *CHARME'99: Proceedings of Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[29] E.M. Clarke, O. Grumberg, and D. Long. "Model Checking and Abstraction". *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[30] E.M. Clarke, O. Grumberg, M. Talupur, and D. Wang. "Making Predicate Abstraction Efficient: Eliminating Redundant Predicates". In *CAV '03 : Proceedings of 15th International Conference on Computer Aided Verification*, pages 355–367, 2003.

[31] E.M. Clarke, O. Grumburg, and D. Peled. *"Model Checking"*. The MIT Press, 1999.

[32] J. Clause, W. Li, and A. Orso. "Dytan: A Generic Dynamic Taint Analysis Framework". In *ISSTA '07: Proceedings of the international symposium on software testing and analysis*, pages 196–206. ACM Press, 2007.

[33] F. Corno, M.S. Reorda, G. Squillero, A. Manzone, and A. Pincetti. "Automatic Test Bench Generation for Validation of RT-level Descriptions: An Industrial Experience". In *DATE '00: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 385–389, New York, NY, USA, 2000. ACM.

[34] D. Corvino, I. Epicoco, F. Ferrandi, F. Fummi, and D. Sciuto. "Automatic VHDL Restructuring for RTL Synthesis Optimization and Testability Improvement". In *ICCD'98: Proceedings of Internatinional Conference on Computer Design*, pages 587–596, 1998.

[35] S. Coudert. "Une abstraction pour la vérification de composants matériels: La significativité de leurs entrées et sorties". Technical report, TELECOM ParisTech, Oct 2008.

[36] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[37] P. Cousot and R. Cousot. "Systematic Design of Program Transformation Frameworks by Abstract Interpretation". In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190. ACM Press, New York, Jan. 2002.

[38] B. Farwer and M. Varea. "Separation of Control and Data Flow in High-Level Petri Nets: Transforming Dual Flow Nets into Object Petri Nets". *Fundamenta Informaticae*, 72(1-3):123–137, Jan. 2006.

[39] A. Fraboulet, T. Risset, and A. Scherrer. "Cycle Accurate Simulation Model Generation for SoC Prototyping". In *SAMOS'04: International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 453–462. Springer-Verlag, 2004.

[40] C. Fritz. "Constructing Büchi Automata from Linear Temporal Logic using Simulation Relations for Alternating Büchi Automata". In *Implementation and Application of Automata. 8th International Conference (CIAA)*, pages 35–48. Springer-Verlag, 2003.

[41] D. Gajski and L. Ramachandran. "Introduction to High-Level Synthesis". *IEEE Design and Test*, 11(4):44–54, 1994.

[42] P. Gastin and D. Oddoux. "Fast LTL to Büchi Automata Translation". In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 53–65, London, UK, 2001. Springer-Verlag.

[43] I. Ghosh, A. Raghunathan, and N. Jha. "A Design for Testability Technique for RTL Circuits Using Control/Data Flow Extraction". In *ICCAD '96: Proceedings of the IEEE/ACM International Conference on Computer-aided Design*, pages 329–336. IEEE Computer Society, 1996.

[44] G.D. Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. "EFSM Manipulation to Increase High-Level ATPG Effectiveness". In *ISQED'06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 57–62. IEEE Computer Society, 2006.

[45] R. Gupta, P.L. Guernic, S.K. Shukla, and J. Talpin, editors. *"Formal Methods and Models for System Design: A System Level Perspective"*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[46] N. Halbwachs. *"Synchronous Programming of Reactive Systems"*. Kluwer Academic Publishers, 1993.

[47] E. Hansen. "A Generalized Interval Arithmetic". In *Proceedings of the International Symposium on Interval Mathemantics*, pages 7–18. Springer-Verlag, 1975.

[48] I. G. Harris. "Fault Models and Test Generation for Hardware-Software Covalidation". *IEEE Design and Test of Computers*, 20(4):40–47, 2003.

[49] S. Horwitz, T. Reps, and D. Binkley. "Interprocedural Slicing using Dependence Graphs". *ACM Transactions on Programming Langauges and Systems*, 12(1):26–61, 1990.

[50] Y. Hoskote, D. Moundanos, and J.A Abraham. "Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors". In *ICCD'95: Proceedings of International Conference on Computer Design*, pages 532–537. IEEE Computer Society, 1995.

[51] Y. Hsieh and S. Levitan. "Control/Data-flow Analysis for VHDL Semantic Extraction". In *Proceedings of the 4th Asia-Pacific Conference on Hardware Description Languages*, pages 68–75, 1997.

[52] Y Hsieh and S. Levitan. "Model Abstraction for Formal Verification". In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 140–147. IEEE Computer Society, 1998.

[53] Y. Hsieh and S. Levitan. "Abstraction Techniques for Verification of Multiple Tightly Coupled Counters, Registers and Comparators". In *IEEE International High-Level Design Validation and Test Workshop*, pages 133–138, 2000.

[54] C. Hymans. "Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation". In *9th International Symposium on Static Analysis*, pages 493–498, 2002.

[55] C. Hymans. *Correct Hardware Design and Verification Methods*, chapter "Design and Implementation of an Abstract Interpreter for VHDL", pages 263–269. Lecture Notes in Computer Science. 2003.

[56] Cadence Design Systems Inc., 2008. Encounter RTL Compiler Synthesis.

[57] Institute of Electrical and Electronic Engineers (IEEE). *IEEE Standard VHDL Language Reference Manual:IEEE Std 1076*, 1993.

[58] Institute of Electrical and Electronic Engineers (IEEE). *IEEE Standard for Register Transfer Level (RTL) Synthesis:IEEE Std 1076.6*, 2004.

[59] Institute of Electrical and Electronic Engineers (IEEE). *IEEE Standard for Property Specification Language (PSL):IEEE Std 1850* , 2005.

[60] M.A. Iyer. "RACE: A Word-Level ATPG-Based Constraints Solver System For Smart Random Simulation". In *Proceedings of IEEE Test Conference*, pages 299–308, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[61] Y. Kesten and A. Pnueli. "Control and Data Abstraction: The Cornerstones of Practical Formal Verification". *International Journal on Software Tools for Technology Transfer*, 2(4):328–342, 2000.

[62] J. Kim and F. Kurdahi. "Finite State Machine Optimization Algorithms for Pipelined Data Path Controllers". In *Proceedings of 4th Annual IEEE International ASIC Conference and Exhibit*, pages P18–7/1–4, 1991.

[63] J. Kim and F. Kurdahi. "Assertion-based On-line Verification and Debug Environment for Complex Hardware Systems". In *Proceedings of IEEE International Symposium on Circuits and Systems*, pages 685–688. IEEE, 2004.

[64] T. Kirkland and M.R. Mercer. "Algorithms for Automatic Test-pattern Generation". *IEEE Journal of Design and Test of Computers*, 5(3):43–55, 1988.

[65] Y. Kukimoto. *"BLIF-MV"*. The VIS Group, University of California, Berkeley, 1996.

[66] R. Kurshan. *"Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach"*. Princeton University Press, Princeton, NJ, USA, 1994.

[67] M. Lajolo, L. Lavagno, M. Rebaudengo, M.S. Reorda, and M. Violante. "Automatic Test Bench Generation for Simulation-based Validation". In *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*, pages 136–140, New York, NY, USA, 2000. ACM.

[68] V.J. Lam and K.A. Olukotun. "DCP: an Algorithm for Datapath/Control Partitioning of Synthesizable RTL Models". In *ICCD'98: Proceedings of the International Conference on Computer Design*, page 442. IEEE Computer Society, 1998.

[69] T. Lengauer and R. Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". *ACM Transactions on Programming Languages Systems*, 1(1):121–141, 1979.

[70] Bi-Xin Li, Xiao-Cong Fan, Jun Pang, and Jian-Jun Zhao. "A Model for Slicing JAVA Programs Hierarchically". *Journal of Computer Sciences Technology*, 19(6):848–858, 2004.

[71] Nathalie Rose T. Lim, Cheryl Anne G. Cordova, Christie Diane Y. Lopez, and Carissa P. Recto. "ANSI C Program Slicing Tool and Text Generator for an Interactive Learning Environment". In *ICALT '05: Proceedings of the Fifth IEEE International Conference on Advanced Learning Technologies*, pages 193–197. IEEE Computer Society, 2005.

[72] C. Liu and J. Jou. "An Automatic Controller Extractor for HDL Descriptions at the RTL". *IEEE Transactions on Design and Test*, pages 72–77, 2000.

[73] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. "Property Preserving Abstractions for the Verification of Concurrent Systems". *Formal Methods in System Design*, 6(1):11–44, 1995.

[74] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In *PLDI '05: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM, 2005.

[75] M. Nanda. *"Slicing Concurrent Java Programs: Issues and Solutions"*. PhD thesis, Indian Institute of Technology, Bombay, 2001.

[76] E. Macci, B. Plessier, and F. Somenzi. "Formal Verification of Digital Systes by Automatic Reduction of Data Paths". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(10):1136–1156, 1997.

[77] E. Macii, B. Plessier, and F. Somenzi. "Verification of Systems Containing Counters". In *ICCAD'92: Proceedings of the IEEE/ACM international conference on Computer-aided Design*, pages 179–182. IEEE Computer Society Press, 1992.

[78] W. Masri, A. Podgurski, and D. Leon. "Detecting and Debugging Insecure Information Flows". In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 198–209. IEEE Computer Society, 2004.

[79] K. McMillan. *"Symbolic Model Checking"*. Kluwer Academic Publishers, 1993.

[80] Mentor Graphics. *ModelSim® SE Foreign Language Interface Manual*, 2007.

[81] G. D. Micheli. *"Synthesis and Optimization of Digital Circuits"*. McGraw-Hill Science/Engineering/Math, 1994.

[82] K. Morin-Allory and D. Borrione. "Proven Correct Monitors from PSL Specifications". In *DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1246–1251. European Design and Automation Association, 2006.

[83] S. Muchnick and N. Jones. *"Program Flow Analysis: Theory and Application"*. Prentice Hall Professional Technical Reference, 1981.

[84] W. Muhammad, S. Coudert, R. Ameur-Boulifa, and R. Pacalet. "Semantic Preserving RTL transformation for Control-Data slicing in virtual IPs". In *INMIC'07: IEEE International Multitopic Conference*, pages 110–115. IEEE Computer Society Press, 2007.

[85] W. Muhammad, S. Coudert, R. Ameur-Boulifa, and R. Pacalet. "Separating Control and Data Processing in RT Level Virtual IP Components". In *PRIME '07: IEEE Microelectronics and Electronics Conference*, pages 273–276. IEEE Computer Society Press, 2007.

[86] R. Namballa, N. Ranganathan, and A. Ejnioui. "Control and Data Flow Graph Extraction for High-level Synthesis". In *Proceedings of IEEE Computer society Annual Symposium*, pages 187–192, 2004.

[87] V.P. Nelson. *"Digital Circuit Analysis and Design"*. Prentice Hall, 1995.

[88] M. Nourani, J. Carletta, and C. "A Scheme For Integrated Controller-datapath Fault Testing". In *DAC'97: Proceedings of the 34th Design Automation Conference*, pages 546–551, 1997.

[89] K. Ottenstein and L. Ottenstein. "The Program Dependence Graph in a Software Development Environment". In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM Press, 1984.

[90] V. Paruthi, N. Mansouri, and R Vemuri. "Automatic Data Path Abstraction for Verification of Large Scale Designs". In *ICCD'98: Proceedings of the International Conference on Computer Design*, pages 192–194. IEEE Computer Society, 1998.

[91] S. Pasricha, N. Dutt, and M. Ben-Romdhane. "Fast Exploration of Bus-based on-chip Communication Architectures". In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 242–247. ACM, 2004.

[92] S. Ramesh, A. Kulkarni, and V. Kamat. "Slicing Tools for Synchronous Reactive Programs". In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 217–220. ACM Press, 2004.

[93] G Ritter. *"Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation"*. PhD thesis, Université Joseph Fourrier Grenoble I, France, 2001.

[94] A. Scarpelli. "Standard VHDL Analyzer and Intermediate Representation". In *NAECON'98: Proceedings of the IEEE Aerospace and Electronics Conference*, 1998.

[95] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, and I. Bolsens. "Hardware Reuse at the Behavioral Level". In *DAC'99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 784–789. ACM, 1999.

[96] D. Sciuto, L. Baresi, and C. Bolchini. "Software Methodologies for VHDL Code Static Analysis based on Flow Graphs". In *EURO-DAC '96/EURO-VHDL '96: Proceedings of the conference on European design automation*, pages 406–411. IEEE Computer Society Press, 1996.

[97] L. Shannon and P. Chow. "SIMPPL: An Adaptable SoC Framework Using a Programmable Controller IP Interface to Facilitate Design Reuse". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(4):377–390, 2007.

[98] S. Swan. "SystemC Transaction Level Models and RTL Verification". In *DAC '06: 43rd ACM/IEEE Design Automation Conference*, pages 90–92, 2006.

[99] T. Tolstrup, F. Nielson, and H. Nielson. *Parallel Computing Technologies*, chapter "Information Flow Analysis for VHDL", pages 79–98. Lecture Notes in Computer Science. 2005.

[100] E. Torbey. *"Control/data Flow Graph Synthesis using Evolutionary Computation and Behavioral Estimation"*. PhD thesis, Ottawa, Ont., Canada, 1999.

[101] University of California, Berkeley. *"Berkley Logic Interchange Format (BLIF)"*, 2005.

[102] S. Vasudevan, E.A. Emerson, and J.A. Abraham. "Improved Verification of Hardware Designs through Antecedent Conditioned Slicing". *International Journal on Software Tools for Technology Transfer (STTT)*, 2005.

[103] V. Vedula, J.A. Abraham, J. Bhadra, and R. Tupuri. "A Hierarchical Test Generation Approach Using Program Slicing Techniques on Hardware Description Languages". *Journal of Electronic Testing: Theory and Applications*, 19(2):149–160, 2003.

[104] G. Venkatesh. "The Semantic Approach to Program Slicing". In *PLDI '91: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 107–119, New York, NY, USA, 1991. ACM.

[105] L. Wang, C. Wu, and X. Wen. *"VLSI Test Principles and Architectures: Design for Testability"*. Morgan Kaufmann, 2006.

[106] M. Weiser. "Program Slicing". In *ICSE'81: Proceedings of the 5th International Conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[107] P. Wilsey, D. Martin, and K. Subramani. "SAVANT/TyVIS/WARPED: Components for the Analysis and Simulation of VHDL". In *IVC/VIUF'98: Proceedings of International Verilog HDL Conference and VHDL International Users Forum*, pages 195–201, 1998.

[108] P. Wolper. "Constructing Automata from Temporal Logic Formulas: A Tutorial". In *Summer School on Trends in Computer Science, LNCS 2090*, pages 261–277. Springer-Verlag, 2002.

[109] J. Zhu. “MetaRTL: Raising the Abstraction Level of RTL Design”. In *Design, Automation and Test in Europe (DATE). Conference and Exhibition 2001. Proceedings*, pages 71–76, 2001.

Résumé

De nos jours la conception des IP (IP: Intellectual Property) peut bénéficier de nouvelles techniques de vérification symbolique: abstraction de donnée et analyse statique formelle. Nous pensons qu'il est nécessaire de séparer clairement le *Contrôle* des *Données* avant toute vérification automatique.

Nous avons proposé une définition du «contrôle» qui repose sur l'idée intuitive qu'il a un impact sur le *séquencement* de données. Autour de cette idée, le travail a consisté à s'appuyer sur la sémantique des opérateurs booléens et proposer une extension qui exprime cette notion de *séquencement*. Ceci nous a mené à la conclusion que la séparation parfaite du contrôle et des données est illusoire car les calculs dépendent trop de la représentation syntaxique. Pour atteindre notre objectif, nous nous sommes alors basés sur la connaissance fournie par le concepteur: séparation *a priori* des entrées contrôle et des entrées données. De cela, nous avons proposé un algorithme de «slicing» pour partitionner le modèle. Une abstraction fut alors obtenue dans le cas où le contrôle est bien indépendant des données. Pour accélérer les simulations, nous avons remplacé le traitement de données, défini au niveau bit par un modèle d'exécution fonctionnel, tout en gardant inchangé la partie contrôle. Ce modèle intègre des aspects temporels qui permet de se greffer sur des outils de *model checking*. Nous introduisons la notion de *significativité* support des données intentionnelles dans les modèles IP. La significativité est utilisée pour représenter des dépendances de données booléennes en vue de vérifier formellement et statiquement les flots de données. Nous proposons plusieurs approximations qui mettent en œuvre cette nouvelle notion.

Abstract

Hardware verification has become challenging due to ever-growing complexity of today's designs. We aim at assisting verification of hardware intellectual property (IP) modules at register transfer level (RTL) by means of data abstraction and static formal analysis techniques. We believe that before applying data abstraction, it is necessary to clearly define and separate the *Control* and *Data processing* of modules.

The consideration of control and data in hardware has previously been a subjective judgment of the designer, based on the syntax. We intuitively define the "Control" as an entity responsible for the *timings* of the data operations in IP modules. The proposed definition was envisaged for separating Control and Data, independent of the subjective choice or the specific syntax. We have worked around a few semantic issues of the definition and demonstrated by reasoning, that an ideal separation of control and data is not achievable according to the proposed definition due to the syntax dependent boolean computations. We therefore, separate the Control and Data based on designer's knowledge. A control-data slicing algorithm is proposed to split the module into a *control slice* and a *data slice*.

An abstraction is achieved in case of slicing with data-independent Control. The bit accurate RTL data slice is replaced by a functional data computation model for fast simulations. The control slice being critical entity with timing information, remains intact during this process. This provides us a way of abstracting the data processing and considering only the timing information for formal verification. We have proposed the notion of *significance* to represent the intentional data in IP modules. Significance is used to represent boolean data dependencies in modules for formal verification of the data flows. Approximations to data dependencies in IP modules have been realized with demonstration of their correctness. The verification technique based on significance is realized which enables to formally verify properties related to the datapaths.