Politecnico di Milano
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

A DESIGN METHODOLOGY FOR AN INNOVATIVE PARALLEL CONTROLLER IN HIGH-LEVEL SYNTHESIS

Relatore: Prof. Fabrizio FERRANDI
Correlatore: Ing. Christian Pilato

Tesi di Laurea di:
Silvia Lovergine
Matricola n. 735041
Vito Giovanni Castellana
Matricola n. 735006

Anno Accademico 2010-2011

Contents

Introduction **23**

1 Preliminaries **31**

1.1 Introduction to High Level Synthesis 31

1.2 FSMD Architectural Model 33

1.2.1 Data Path 34

1.2.2 Finite State Machine 35

1.3 High Level Synthesis Flow Overview 39

1.4 Front End 41

1.4.1 Lexical Processing And Intermediate Representation Creation 41

1.4.2 Control/Dataflow Analysis and Algorithm Optimization 41

1.4.3 Graph-Based Representations in High Level Synthesis . 42

1.5 Synthesis 53

1.5.1 Scheduling 54

1.5.2 Resource Allocation 56

1.5.3 Module Binding 58

1.6 Controller Synthesis 58

1.7 Back End 62

1.7.1 Generation 62

2 State of the Art **65**

2.1 Design Methodologies Features in HLS 66

2.2 HLS Prehistoric Period 69

2.3 First HLS Generation 69

2.3.1 Architectural Models 71

2.4 Second HLS Generation 73

2.4.1 Architectural Models 74

2.4.2 Improving Control-Oriented Specifications Synthesis . . 81

2.5 Third HLS Generation 85

2.5.1 Architectural Models 88

2.5.2 Specification Description 89

2.6 Why Fourth HLS Generation is Prospected 96

2.7 Conclusions 103

3 Proposed Methodology **107**

3.1 Project Organization 108

3.2 Analysis Process Overview 110

3.2.1 Motivational Example 111

3.2.2 Parallelism Identification 113

3.2.3 Enabling Conditions Identification 117

3.2.4 Enabling Conditions Representation 118

3.3 Parallel Controller Graph 119

3.3.1 False-labeled control flow edges insertion algorithm . . 123

3.3.2 Back-edges insertion algortihm 127

3.4 Activation Function 131

3.4.1 Control Path and Control Path Signal 135

3.4.2 Control Dependencies Activation Signal 137
3.4.3 Data Dependencies Activation Signal 138
3.4.4 Preserving the Dominance Property 142
3.4.5 Back Edges Activation Signal 145
3.4.6 Activation Function Formulation 146
3.5 Controller Synthesis Process Overview 148
3.6 Architectural Model 149
3.7 Activation Function Module 151
3.7.1 Join Module 152
3.7.2 Or Module 155
3.7.3 Condition Join Module 155
3.8 Control Module 156
3.8.1 Control Element 157
3.8.2 Priorities Manager 159
3.9 Controller Synthesis Flow 162
3.9.1 Common Ports Instantiation and Interfaces Creation . 162
3.9.2 Priorities Managers Allocation 164
3.9.3 Activation Function Modules, Control Modules and Condition Join Modules Creation 164
3.9.4 Interfaces Connection 165
3.10 Optimizations 165
3.10.1 Static Single Assignment Form Transformation 167
3.10.2 Activation Functions Simplifications 168

4 Experimental Evaluation **183**
4.1 High Level Synthesis Details 183
4.2 Experimental setup 187
4.3 Performance Evaluation 190

4.4 Area Evaluation 194
4.5 Concluding Remarks 200

5 Conclusions and Future Works **201**

Bibliography **207**

List of Figures

1.1 *Typical Architecture composed of a Finite State Machine (FSM) and a Datapath.* 34

1.2 *Sequential circuit's Huffman model.* 38

1.3 *Typical HLS flow.* 40

1.4 *Pseudo-code and CFG of Euclid's algorithm* 43

1.5 *FSM representation of Euclid's algorithm CFG.* 44

1.6 *Euclid algorithm dominance tree.* 45

1.7 *Euclid's algorithm post dominance tree.* 45

1.8 *Euclid's algorithm CDG.* 46

1.9 *Pseudo-code and DFG of a program that computes the average of 4 numbers* 47

1.10 *Pseudo-code and DDG of "lowError"function* 48

1.11 *CDFG of the "lowError"function example.* 49

1.12 *PDG of the "lowError"function example.* 51

1.13 *Example code and corresponding translation in SSA form.* 52

1.14 *Example code and corresponding translation in SSA form, with* $\phi - function$ *insertion.* 53

1.15 *"average"function example scheduled CFG under resource constraints.* 55

1.16 *"low error" function example scheduled CDFG.* 55

1.17 *Hierarchical FSM model.* 60

1.18 *Hierarchical controller model in the presence of diramation constructs.* 61

1.19 *RTL description written with different binding details.* 62

2.1 *Main Features Characterizing Design Methodologies in HLS.* . . 68

2.2 *Standard Structure of a Centralized FSM.* 72

2.3 *Parallel Decomposition of a Finite State Machine.* 76

2.4 *Cascade Decomposition of a Finite State Machine.* 76

2.5 *Generalized Decomposition of a Finite State Machine.* 77

2.6 *Tipical HLS tools output including RTL and RTL test benches.* 87

2.7 *Example of CFG for a Specification.* 91

2.8 *A possible FSM for the CFG in Figure 2.7.* 92

2.9 *Hardware Implementation for the FSM in Figure 2.8.* 92

2.10 *Example of Petri Net Specification of a Controller.* 95

2.11 *Sales of electronic system-level synthesis tools. Source: Gary Smith EDA statistics.* 97

2.12 *Area generated by different HLS tools on a quantization and inverse-quantization (QIQ) module.* 99

2.13 *Maximum frame rate achieved for a video application on a DSP processor and an FPGA plus HLS tools.* 101

2.14 *FPGA resource utilization on a wireless receiver, implemented using HLS tools versus hand-written RTL code.* 101

3.1 *Project Flow: the Steps in which this Project is Organized.* . . . 109

3.2 *CFG corresponding to the Motivational Example C code.* 114

3.3 *PDG obtained from the motivational example C specification.* . . 116

3.4 *First version of the PCG for the motivational example obtained from the corresponding PDG by applying the first steps of the PCG construction algorithm.* . . . 122
3.5 *Loop forest tree for the motivational example specification.* . . 124
3.6 *PCG for the motivational example after false-labelled control flow edges (red edges) insertion.* . . . 126
3.7 *A possible representation produced by the front-end for loops structure and the loop forest of the corresponding specification.* 130
3.8 *Complete version of the PCG for the motivational example.* . . 132
3.9 *Activation function graphical representation* . . . 133
3.10 *Example showing the need of considering control paths in data dependencies activation signal computation.* . . . 140
3.11 *Example PCG.* . . . 149
3.12 *Synthesized architecture scheme.* . . . 150
3.13 *Finite state machine representing 3-inputs join module* . . . 152
3.14 *Portion of a PDG showing incoming edges of a loop condition node* . . . 153
3.15 *Example of inadequacy of standard join module* . . . 154
3.16 *Modified 2-input join module* . . . 154
3.17 *Condition join module schematic representation.* . . . 156
3.18 *Control Element module schematic representation.* . . . 157
3.19 *Control Element module FSM representation.* . . . 158
3.20 *Priorities manager schematic representation.* . . . 159
3.21 *Example parallel controller graph.* . . . 160
3.22 *Interaction between priorities manager and control elements at control step t=1; red edges denotes high-valued signals.* . . . 161
3.23 *Control module schematic representation.* . . . 162

3.24 *Controller synthesis step added in the HLS flow.* 163
3.25 *Example code and corresponding PCG.* 167
3.26 *SSA translation of previous example's code and corresponding PCG.* 168
3.27 *Karnough Map for the activation function of node 7 in the PCG shown in Figure 3.8, after don't care conditions addition on the output values.* 173
3.28 *Schematic representation of the circuit portion implementing the AF in SOP form for node 5 of the PCG in Figure 3.8.* . . 180
3.29 *Schematic representation of the circuit portion implementing the AF in POS form for node 5 of the PCG in Figure 3.8.* . . 181

4.1 *Panda framework schematic overview.* 184
4.2 *PandA analysis flow.* 185
4.3 *GCC internal structure.* 185
4.4 *Overview of ISE's design flow.* 190

List of Tables

1.1 *Typical HLS inputs and outputs* 32

2.1 *Regular Expression DAG Symbols.* 84

2.2 *EDA segments revenue analysis ($ MILLIONS). Source: EDAC MSS Statistics Report and Desaisive Technology Research analysis.* 98

3.1 *Level assignment for the Operations in the PDG.* 117

3.2 *Activation Functions in extended form for the Operations in the PCG obtained from the motivational example.* 147

3.3 *Activation Functions in extended form for the nodes of the PCG in Figure 3.25.* 169

3.4 *Activation Functions in extended form for the nodes of the PCG in Figure 3.26.* 169

3.5 *Common properties of boolean algebra useful to simplify activation functions.* 170

3.6 *Truth table for the reduced activation function of node 7 in the PCG shown in Figure 3.8.* 172

3.7 *Truth table for the activation function of node 7 in the PCG shown in Figure 3.8, after don't care conditions addition on the output values.* 173

3.8 An optional table caption ... 174

3.9 *Truth table for the extended activation function of node 6 in the PCG shown in Figure 3.10, after don't care conditions addition on the output values.* 179

4.1 *Unique parallel region class benchmarks characteristics.* 188

4.2 *Multiple parallel regions class benchmarks characteristics.* 189

4.3 *Clock Cycles for the execution of class1 benchmarks.* 191

4.4 *Clock Cycles for the execution of Synth_FIR_lx1, Synth_FIR_lx2 and Synth_2FIR_l_Biquad benchmarks.* 191

4.5 *Clock Cycles for the execution of Synth_complex benchmarks.* 192

4.6 *Clock Cycles for the execution of Synth_lx's benchmarks.* 193

4.7 *Number of resources needed to implement the parallel controller and estimation of the required flip-flops (Estimated #FF). #CE is the number of Control Elements, #PT of the Priority Managers, #C-JOIN of the Conditional Join Modules, #JOIN of the other Join Modules, #OR of the OR modules. Estimated #JOIN-FF is the number of estimated flip-flops for the Join Modules.* 195

4.8 *Estimated number of Flip Flops for controller and datapath, in the case of both FSM and PC-based designs.* 196

4.9 *Total number of estimated and actually used Flip Flops.* 198

4.10 *Number of used LUTs.* 199

Abstract

Il presente lavoro di tesi propone una metodologia di progetto, integrata nella sintesi ad alto livello, per la costruzione di un controllore parallelo che affronti il problema dell'estrazione del parallelismo. Il controllore parallelo proposto è in grado di identificare autonomamente e a run-time le condizioni che vincolano l'esecuzione delle istruzioni e abilitarne l'esecuzione il prima possibile, ovvero non appena le dipendenze e i vincoli sulle risorse sono soddisfatti. Tali condizioni vengono ricavate tramite l'analisi di una nuova rappresentazione, appositamente definita per rappresentare il parallelismo. Questo controllore non è composto da un insieme di macchine a stati locali comunicanti, ma è più simile ad una descrizione comportamentale della specifica, composto da diversi moduli, che interagiscono secondo uno schema basato su token, che rappresenta un paradigma di comunicazione molto semplice. Questa scelta permette di evitare il problema dell'overhead di comunicazione. Il controllore parallelo, inoltre, è ottenuto tramite un approccio dal basso verso l'alto. In tal modo non è necessaria la costruzione dell'intera macchina a stati, evitando i problemi legati alle tecniche di decomposizione. Infine, l'aver dotato il controllore della capacità di identificare autonomamente le suddette condizioni ha portato a dover affrontare il problema della formalizzazione del parallelismo. Tale problema consiste nel trovare un modo per formalizzare l'informazione sul parallelismo in modo tale che il controllore possa capirla e

gestirla automaticamente. Poiché il controllore è in grado di gestire segnali, tale informazione è stata formalizzata come una composizione di segnali, che indica il momento in cui un'istruzione può essere eseguita, ovvero quando le suddette condizioni sono soddisfatte. Tale composizione di segnali definisce una funzione associata a ciascuna istruzione. Le affinità con l'algebra di Boole rendono tali funzioni facili da ottimizzare per mezzo di tecniche standard.

* * * * * * * * * * *

This thesis work proposes a design methodology to build a parallel controller in high-level synthesis, able to handle with the parallelism problem. The proposed controller is able to automatically identify the conditions that enable the execution of an instruction as soon as the dependencies and the resource constraints are satisfied. Such conditions are obtained through a parallelism identification phase, performed by means of the analysis of a novel intermediate representation, properly defined to represent the inherent parallelism. In the proposed architecture, the task of assign a control step in which each instruction will be executed is implicitly performed at run-time by the parallel controller. The proposed controller structure is not composed of a set of communicating local finite state machines. It is instead closer to a behavioral description of the specification composed of several modules, interacting according to a token-based scheme, which is a very simple communication paradigm. This choice prevents to incur in the communication overhead problem. Moreover, it is obtained through a bottom-up approach. In this way, the entire finite state machine construction is not needed, avoiding also the problems related to decomposition techniques. Finally, making

the controller able to identify by itself the conditions that enable instructions execution led to face another aspect of the parallelism issue, that is the parallelism formalization. Such problem consists in how to formalize the information about the parallelism, obtained from the analysis of the proposed IR, in such a way to make the controller able to understand and automatically manage this information. Since the controller is able to manage signals, such information has been formalized as a composition of signals, indicating when the execution can start, i.e. when the above mentioned conditions are satisfied. Such composition of signals defines a function associated with each instruction. Similarities with Boolean Algebra makes such functions easily manageable and optimizable by means of standard techniques.

Estratto in lingua italiana

Con il passare degli anni, le metodologie proposte per il progetto dei circuiti digitali hanno subito una migrazione verso livelli di astrazione sempre più alti. Questo processo ha portato all'introduzione delle tecniche di *sintesi ad alto livello* (HLS). A tali tecniche viene attribuito il termine "alto livello" a sottolineare il fatto che una specifica viene implementata in hardware sfruttando direttamente una sua descrizione *comportamentale*, senza alcuna informazione circa la sua descrizione *strutturale*. Solitamente, il modello architetturale risultante dal processo di sintesi ad alto livello consiste di due parti: un *datapath*, che include un insieme di moduli RTL opportunamente connessi, utilizzati per eseguire le operazioni presenti nella specifica, ed un *controllore*, che fornisce la logica per attivare l'esecuzione di tali operazioni.

La sintesi ad alto livello fu concepita nel 1974 da M. Barbacci. Da allora sono passati decenni di miglioramenti, riguardo per esempio il linguaggio utilizzato per descrivere la specifica, l'architettura finale proposta o il supporto alla sintesi di applicazioni appartenenti a diversi domini applicativi. Tuttavia, nonostante i vari miglioramenti, il register-transfer level (RTL) rimane ancora il livello di sintesi e di specifica dominante. Le ragioni di ciò possono essere trovate nel fatto che esistono ancora molti aspetti da definire e formalizzare nell'HLS. Ad esempio, i primi lavori proposti per la sintesi ad alto livello era-

no puramente legati ad uno specifico dominio applicativo, tipicamente quello del Digital Signal Processing (DSP) e delle applicazioni orientate al flusso di dati. A partire dalla metà degli anni novanta vari gruppi di ricercatori iniziarono a concentrarsi anche su domini applicativi diversi. Tuttavia, ancora oggigiorno, il limitato grado di accettazione dell'HLS da parte degli utenti dimostra la necessità di una metodologia generale che supporti la sintesi nei vari domini applicativi.

Tra i vari aspetti dell'HLS che necessitano miglioramenti, in questo lavoro di tesi ci si concentra sul problema dell'*estrazione del parallelismo*, e sugli aspetti ad esso connessi. Nel seguito verranno presentate alcune considerazioni per chiarire lo scenario attuale circa i suddetti problemi.

A partire dai primi anni duemila, i linguaggi di programmazione ad alto livello come C e simili sono diventati lo standard per la descrizione di una specifica in HLS. L'adozione di questi linguaggi ha permesso di sfruttare trasformazioni ed ottimizzazioni a livello di compilatore, portando enormi vantaggi in termini di qualità dei risultati della sintesi. Tuttavia, questi linguaggi sono caratterizzati da una natura sequenziale, che non permette di rappresentare esplicitamente il parallelismo presente in un'applicazione. Quindi l'adozione dei linguaggi simili al C per la descrizione della specifica ha introdotto il problema dell'*identificazione del parallelismo*. Per risolvere questo problema sono state proposte alcune estensioni a tali linguaggi, come ad esempio Handel-C [2]. Handel-C fornisce delle istruzioni non standard per controllare l'instanziazione dell'hardware, con particolare enfasi sul parallelismo. Questo tipo di estensioni, tuttavia, non è supportato dalla presenza di metodologie per l'inserimento automatico delle istruzioni non standard. Infatti, l'inserimento di tali istruzioni rientra tra i compiti del progettista, aumentando di conseguenza i costi di progetto. Un approccio più generale

per risolvere il problema dell'identificazione del parallelismo potrebbe basarsi sulla produzione di un'opportuna descrizione della specifica, più adatta a mostrare il parallelismo disponibile. Tuttavia, questo tipo di soluzioni porta ad un altro problema, ovvero quello della *rappresentazione del parallelismo*. Solitamente si costruisce una rappresentazione intermedia (IR) sotto forma di grafo, in quanto queste strutture risultano un buon mezzo per l'analisi della specifica, mostrandone chiaramente le proprietà. Quindi, riepilogando, il problema dell'identificazione del parallelismo può essere affrontato tramite la costruzione di un'opportuna IR capace di mostrare il parallelismo presente, ovvero cercando di risolvere il problema della rappresentazione del parallelismo. Questi due problemi, ovvero identificazione e rappresentazione del parallelismo, possono essere aggravati dalla scelta di un'architettura finale inadeguata. Infatti, anche se fosse possibile costruire una IR che permetta di identificare tutto il parallelismo presente, l'architettura finale potrebbe non essere in grado di sfruttare tale parallelismo, o potrebbe risultare troppo costosa in termini di area. Si consideri, per esempio, una specifica composta da due costrutti ciclici parallelizzabili. Scegliendo un'architettura finale composta da un datapath e da una FSM centralizzata come controllore, esistono diverse soluzioni per l'implementazione della specifica. Una prima soluzione potrebbe essere quella di sequenzializzare l'esecuzione dei cicli, non sfruttando quindi il potenziale parallelismo. Un'altra possibile soluzione potrebbe essere quella di realizzare una FSM in cui gli stati sono ottenuti dalla combinazione delle operazioni che possono essere eseguite contemporaneamente. Quest'ultima soluzione non limita lo sfruttamento del parallelismo, ma risulta in un'esplosione del numero di stati della FSM, diventando quindi troppo costosa in termini di area. Sulla base di queste considerazioni è possibile identificare un terzo aspetto legato all'estrazione del parallelismo, ovvero quello

dello *sfruttamento del parallelismo*.

La sintesi ad alto livello è di solito svolta attraverso diverse fasi, come lo scheluling delle istruzioni, il binding delle risorse, e la sintesi di datapath e controllore. Ciascuno di questi compiti rappresenta un problema NP-completo. Lo *scheduling* definisce una relazione d'ordine sull'insieme delle istruzioni associando un livello di priorità a ciascuna operazione. Dopodiché, sfrutta le priorità precedentemente calcolate per assegnare a ciascuna operazione il passo di controllo in cui verrà eseguita. Il *binding* associa ciascuna istruzione con una particolare istanza di una risorsa su cui l'operazione verrà eseguita. Le decisioni prese in queste fasi influiscono sulle performance e possono limitare lo sfruttamento del parallelismo. Ad esempio, quando più istruzioni indipendenti vengono associate alla stessa risorsa, la loro esecuzione deve essere necessariamente sequenzializzata. Dall'altro lato, quando più istruzioni indipendenti vengono associate a risorse diverse, la loro esecuzione viene forzatamente sequenzializzata se sono state schedulate in passi di controllo differenti. Uno scheduler efficiente dovrebbe permettere l'esecuzione di ciascuna istruzione il più presto possibile, ovvero non appena la risorsa a cui l'istruzione è associata diventa disponibile e non appena le istruzioni da cui essa dipende terminano la loro esecuzione. Tuttavia, non è sempre possibile stabilire in anticipo l'esatto ciclo di clock in cui questi vincoli saranno soddisfatti. Infatti esistono diversi fattori che non possono essere noti a tempo di compilazione, come ad esempio l'esito della valutazione delle condizioni per le istruzioni condizionali. Inoltre, non vincolare le istruzioni ad essere eseguite in un determinato passo di controllo porta spesso ad un'esplosione del numero degli stati della FSM. Infatti in una FSM siffatta ciascuna istruzione potrebbe trovarsi all'interno di più stati, ogni volta associata ad una diversa combinazione di istruzioni concorrenti.

Come precedentemente accennato, l'architettura finale è generalmente composta da un datapath ed un controllore. Il modello più comune per il controllore è quello della FSM centralizzata. Questo però non è l'unico modello esistente. Sono stati infatti proposti diversi modelli architetturali basati sulla FSM, come ad esempio FSM distribuite, parallele e/o gerarchiche. Tra le varie proposte, quella delle FSM parallele affronta il problema del parallelismo implementando una struttura di controllori comunicanti. Queste architetture sono guidate dagli eventi. I controllori comunicano tramite scambio di messaggi, supportati da opportuni protocolli di comunicazione e sincronizzazione. La peculiarità dei controllori paralleli è legata alla loro capacità di schedulare dinamicamente le istruzioni. Spostando la fase di scheduling a run-time essi riescono a sfruttare il parallelismo intrinseco nell'applicazione. Tuttavia, queste architetture sono affette da un overhead significativo dovuto a comunicazione e sincronizzazione. Inoltre, dato che sono ottenute generalmente tramite tecniche di decomposizione delle FSM, l'area totale risulta aumentata. La decomposizione delle FSM è una tecnica che divide una FSM monolitica in un certo numero di sotto-elementi, seguendo un approccio top-down. Spesso risulta impossibile effettuare una decomposizione esatta di una FSM. In queste situazioni, porzioni di circuito devono essere duplicate per essere incluse in diversi sotto-elementi.

Sulla base delle considerazioni fatte è possibile ricavare alcune conclusioni. Innanzitutto, il problema dell'estrazione del parallelismo nell'HLS è composto da tre sotto-problemi: identificazione, rappresentazione e sfruttamento. Per risolvere il problema nella sua interezza devono essere considerati diversi aspetti. Ad esempio, la definizione dell'architettura finale rappresenta un aspetto cruciale per lo sfruttamento del parallelismo. Le soluzioni architetturali proposte ad oggi per risolvere questo problema risultano troppo

costose. Inoltre, il progettista dovrebbe tenere in considerazione il fatto che i compiti del controllore e dello scheduler sono fortemente legati. Infatti, lo scheduler deve stabilire quando ciascuna istruzione deve essere eseguita, mentre il controllore abilita fisicamente l'esecuzione delle istruzioni in base alle decisioni prese dallo scheduler. Inoltre, in modelli architetturali quali i controllori paralleli, controllore e scheduler vengono fusi in un'unica entità. Infine, l'adozione di linguaggi di specifica simili al C porta alla necessità di definire opportunamente una IR capace di rappresentare il parallelismo. Questa IR deve essere attentamente definita anche in base alla particolare architettura finale scelta.

Il presente lavoro di tesi propone una metodologia di progetto, integrata nella sintesi ad alto livello, per la costruzione di un controllore parallelo capace di gestire automaticamente il parallelismo presente nell'applicazione. Il linguaggio di specifica utilizzato è simili al C. Quindi, verrà definita una nuova IR per supportare la rappresentazione del parallelismo. L'idea alla base di questo lavoro consiste nella costruzione del controllore attraverso un approccio dal basso verso l'alto. In questo modo, la costruzione della FSM centralizzata non è più necessaria, evitando quindi i problemi legati alle tecniche di decomposizione. Il controllore parallelo proposto è in grado di abilitare l'esecuzione di ciascuna istruzione non appena le operazioni da cui essa dipende sono state eseguite e non appena i vincoli sulle risorse lo permettono. Tale controllore è inoltre in grado di identificare autonomamente le condizioni da rispettare per abilitare correttamente l'esecuzione delle istruzioni, senza l'ausilio di alcuna decisione dello scheduler. Tali condizioni vengono ricavate tramite un'analisi della IR proposta. Questo significa che parte dei compiti dello scheduler è stata spostata nei compiti del controllore. In particolare, generalmente lo scheduler prima calcola le priorità per le istruzioni, indotte dalla suddetta

relazione d'ordine; dopodiché assegna un passo di controllo per l'esecuzione di ciascuna istruzione. Nell'architettura proposta, invece, quest'ultimo compito viene effettuato a run-time dal controllore, mentre lo scheduler continua a svolgere staticamente solo il primo. Quindi, lo scheduler associa le priorità alle istruzioni a compile-time, mentre il controllore attiva dinamicamente l'esecuzione delle operazioni sulla base delle informazioni ottenute tramite l'analisi della IR proposta. Dato che il binding delle risorse viene effettuato staticamente, può succedere che diverse istruzioni indipendenti vengano associate alla stessa risorsa. In queste situazioni le priorità calcolate staticamente vengono sfruttate per implementare un meccanismo di risoluzione dei conflitti.

Il controllore proposto non è formato da una serie di macchine a stati finiti comunicanti. Esso è invece più vicino ad una descrizione comportamentale della specifica, composta da diversi moduli che interagiscono secondo uno schema basato su token, che rappresenta un paradigma di comunicazione molto semplice. Questa scelta evita di ricadere nel problema dell'overhead di comunicazione. Infine, per poter rendere il controllore in grado di identificare autonomamente ed automaticamente le condizioni per la corretta abilitazione delle operazioni ha portato a dover affrontare un altro problema, quello della *formalizzazione del parallelismo*. È infatti necessario capire come l'informazione sul parallelismo, ottenuta dall'analisi della IR proposta, possa essere formalizzata in modo tale da poter essere compresa e gestita in modo automatico dal controllore.

I principali contributi offerti dal presente lavoro di tesi possono essere riassunti in:

- è stata proposta una nuova IR, chiamata ***Parallel Controller Graph* (PCG)**, per risolvere il problema della *rappresentazione del parallelis-*

mo. Il PCG è stato ottenuto tramite l'analisi di un'altra ben nota IR, ovvero il Program Dependence Graph (PDG) [22]. Il PDG è stato scelto come punto di partenza in quanto mostra le cosiddette dipendenze minime di un programma. Il PCG estende il PDG tramite l'aggiunta dell'informazione minima sul flusso di controllo.

- è stata svolta una ***analisi del PCG*** per risolvere il problema dell'*identificazione del parallelismo.* Tale analisi è stata condotta con lo scopo di individuare l'insieme minimo delle condizioni da soddisfare per abilitare correttamente l'esecuzione di una istruzione.
- sono state formalizzate le informazioni ottenute dall'analisi del PCG, risolvendo il problema della *formalizzazione del parallelismo.* In particolare, dato che il controllore è in grado di gestire segnali, l'informazione circa le suddette condizioni è stata formalizzata per ciascuna istruzione come composizione di segnali. Tale composizione di segnali prende il nome di ***Funzione di Attivazione*** **(AF)**. La funzione di attivazione associata ad una determinata istruzione indica quando essa può essere eseguita, ovvero quando le suddette condizioni sono rispettate.
- è stato proposto un ***controllore parallelo*** per risolvere il problema dello *sfruttamento del parallelismo.* La definizione di questa innovativa architettura è stata effettuata introducendo una serie di moduli interagenti, che sono stati definiti. Dopodiché, il controllore parallelo è stato ricavato tramite un approccio dal basso verso l'alto.
- sono state proposte diverse ***ottimizzazioni*** per ridurre la logica allocata per l'implementazione delle funzioni di attivazione.

Il lavoro di tesi è organizzato nel modo seguente. Nel Capitolo 1 viene introdotto il problema della sintesi ad alto livello e vengono descritti i modelli architetturali più comuni. Il Capitolo 2 fornisce una panoramica sul corrente stato dell'arte circa le tecniche di sintesi ad alto livello, focalizzando l'attenzione su tre principali aspetti: linguaggio adottato per la descrizione della

specifica, dominio applicativo ed architettura finale. Nel Capitolo 3 viene presentato il controllore parallelo proposto. In particolare, viene introdotto il Parallel Controller Graph proposto come IR, vengono formalizzate le funzioni di attivazione, ed infine vengono proposte una serie di ottimizzazioni applicabili al controllore parallelo. I risultati sperimentali sono riportati nel Capitolo 4. Infine, il Capitolo 5 riporta le conclusioni e fornisce suggerimenti per possibili lavori futuri che estendano le capacità del controllore parallelo.

Risultati Sperimentali e Conclusioni

La metodologia proposta per la creazione del controllore parallelo è stata implementata in C++ all'interno del framework PandA [5]. Dopodiché, sono state effettuate una serie di simulazioni utilizzando diversi benchmark per valutare le performance in termini di cicli di clock. In particolare, tramite la sintesi ad alto livello di tali benchmark, effettuata utilizzando il controllore parallelo come architettura obbiettivo, si è ottenuta la descrizione RTL delle specifiche. La sintesi è stata poi ripetuta utilizzando come architettura obbiettivo una FSM monolitica. Le descrizioni RTL ottenute sono state simulate tramite Icarus Verilog [4] ver. 9.3. I risultati delle simulazioni hanno mostrato che il controllore parallelo riesce a sfruttare il parallelismo stabilendo dinamicamente quali istruzioni mandare in esecuzione. Si è potuto notare che per le specifiche puramente dataflow, in cui tutto il parallelismo è identificabile staticamente, le performance ottenute con le due architetture sono identiche. Questo dimostra che il meccanismo di comunicazione basato su token non comporta un overhead di comunicazione. I risultati ottenuti invece per le applicazioni con potenziale parallelismo hanno mostrato un evidente miglioramento delle prestazioni, anche per quelle applicazioni contenenti regioni parallele di dimensioni sbilanciate, e dunque del tutto ingestibili statica-

mente. Oltre all'analisi delle performance, è stata effettuata una valutazione dell'area occupata attraverso la sintesi RTL effettuata tramite Xilinx ISE [6] ver. 11.1 su una FPGA Virtex-5 XC5VLX330T. I risultati hanno mostrato in questo caso un incremento dell'area occupata. Tuttavia, le ragioni di questi risultati possono essere facilmente spiegate. Infatti, il principale limite di questa prima versione del controllore parallelo è quella di non poter contare sulle ottimizzazioni circa l'allocazione dei registri, come invece accade per la FSM.

In conclusione, il controllore parallelo ha mostrato risultati soddisfacenti nell'estrazione del parallelismo. Tuttavia, è possibile suggerire una serie di ottimizzazioni che possono ulteriormente migliorare tale architettura ed estenderne le capacità. Tali suggerimenti possono essere schematizzati nel modo seguente:

- le ottimizzazioni descritte nel Capitolo 3, ovvero trasformazione in forma Static Single Assignment (SSA), semplificazione algebrica, riduzione tramite analisi del flusso e trasformazione in forma Prodotto di Somme (POS), potrebbero essere formalizzate ed implementate. Questo porterebbe ad una riduzione della logica necessaria per implementare le funzioni di attivazione, riducendo di conseguenza il numero di registri richiesti.

- l'allocazione dei registri può essere migliorata applicando delle tecniche di ottimizzazione. Tuttavia, le tecniche di ottimizzazione standard potrebbero risultare inadeguate per il controllore parallelo, in quanto l'esecuzione concorrente delle istruzioni stabilita a run-time complica l'analisi per il riuso dei registri. Quindi, potrebbe essere necessario proporre nuove tecniche di ottimizzazione.

- l'allocazione ed il binding delle risorse sono attualmente effettuati nello stesso modo per la FSM e per il controllore parallelo. In particolare, il controllore parallelo sfrutta gli algoritmi di allocazione e binding sviluppati per l'FSM. Tali algoritmi sono svolti a partire dalle informazioni contenute nel Control Flow Graph (CFG). Poichè il CFG fallisce nel riconoscere tutto il parallelismo, potrebbe succedere che venga allocato un numero di risorse di un certo tipo inferiore al numero di istruzioni indipendenti che hanno bisogno di quel tipo di risorsa per essere eseguite, limitando lo sfruttamento del parallelismo. Inoltre, diverse istruzioni indipendenti potrebbero essere riconosciute come dipendenti nel CFG, e dunque associate alla stessa unità funzionale. Per queste ragioni, dovrebbero essere sviluppati degli algoritmi per l'allocazione ed il binding delle risorse basati sull'analisi del PCG.

Introduction

Over the years, the proposed methodologies for the design of digital circuits moved to higher and higher abstraction levels, leading to High-Level Synthesis (HLS). The term “high-level” is attributed to that class of synthesis techniques implementing a *behavioral* specification directly into hardware, pointing out the absence of any information about the *structural* description of the specification. Usually, the architectural model resulting from such process is composed of two parts: a *datapath*, that includes a set of Register-Transfer Level (RTL) modules, properly connected to each other, to perform the operations, and a *controller*, that provides the logic to issue such operations.

Since 1974, when it was conceived, HLS had long time of improvements, concerning, for example, the input language adopted to describe the specification, the final architecture proposed and the support for the synthesis of specifications belonging to different application domains. Despite the improvements in HLS, however, the RTL is still the dominant specification and synthesis level. This is why many aspects must be still defined and formalized. For example, early HLS methodologies were purely domain-specific approaches, giving good results mainly in Digital Signal Processing and dataflow-oriented applications domains. From mid nineties researchers started to concentrating also on control domain. Nowadays, despite some

vendors claim their HLS tool effectiveness in all the domains, limited users acceptance degree shows the need of a general methodology unifying such domains.

Among the HLS needs, this thesis work focuses on the problem of *parallelism extraction*, and on those related. In the following such issues will be presented through some considerations, that will show the current scenario.

From early 2000, C-like programming languages become the standard in HLS as specifications description. C-like languages adoption allows code restructuring by means of compiler-based transformations and optimizations, leading enormous advantages in terms of quality of synthesis results. However, due to their sequential nature, such languages do not support explicit representation of the inherent parallelism in the specification, thus introducing the *parallelism identification* problem. Some language extensions were proposed to address this problem. An example of such extended languages is Handel-C [2], that includes non-standard instructions to control hardware instantiation with an emphasis on parallelism. Handel-C, however, as the other extensions proposed for high-level programming languages, is not supported by methodologies for the automatic insertion of non-standard instructions. Hence, they must be added by hand, increasing the design effort. A more general approach to address the parallelism identification problem could consists in producing a proper representation of the specification, exposing the available parallelism. However, this leads to another problem, that is the *parallelism representation*. Usually, graphs are used as Intermediate Representation (IR), since they provide a good way to analyze the specification, clearly showing its properties. Hence, the parallelism identification problem can be addressed by finding a proper IR that shows the available parallelism, i.e. by addressing the parallelism representation problem. Such problems

may be exasperated by an inappropriate choice of the final architecture. Indeed, even if it were possible to build a proper IR identifying all the inherent parallelism, the final architecture may be either not able to exploit it or too expansive in terms of area. Consider, for example, a specification containing two parallelizable loops. When a datapath and a centralized FSM are chosen to compose the final architecture, there are some feasible implementations. A first one could consists in sequentializing the loops, thus not exploiting the parallelism. Another one, instead, could consists in realizing an FSM in which the states are obtained as combinations of operations that can be simultaneously executed. Such solution does not restrict the parallelism exploitation, but results in an explosion of the number of states, resulting too expansive in terms of area. Hence, a third aspect of the parallelism problem has been identified, that is the *parallelism exploitation*.
High-level synthesis is generally performed through several sub-tasks, such as instruction scheduling, resources binding, or datapath and controller synthesis. Each of these sub-task is NP-complete. The *scheduling* defines an ordering relation over the set of instructions by associating a priority to each operation. Then, it uses the defined priorities for assigning to each instruction the control step in which it must be executed. The *binding* associates each instruction with the functional unit where it will be executed. The decisions taken in these phases affect timing performances, possibly restricting parallelism exploitation. For example, when multiple independent instructions are bound on the same resource, they must be scheduled in different control steps, sequentializing their execution. On the other side, when multiple independent instructions are bound on different resources, their execution must be sequentialized if they are scheduled in different control steps. A good scheduler should allow the execution of an instruction as soon as possible,

i.e. as the corresponding resource is available and the instructions which it depends on have been executed. However, the clock cycle in which such constraints will be satisfied cannot be always established in advance. Indeed, several factors, such as branch conditions evaluations, are unknown at compile time. Moreover, allowing one instruction to be possibly executed in different control steps often leads to the explosion in the number of states of the controller.

As above mentioned, the final architecture is usually composed of a datapath and a controller, and the most common model for the controller is the FSM. However, several FSM-based architectural solutions have been proposed to implement the controller, such as centralized Finite State Machines (FSMs), distributed FSMs, parallel FSMs and hierarchical FSMs. Among such proposals, parallel FSMs address the parallelism problem by implementing a controller structure composed of communicating sub-controllers. Such architectures are event-driven. Sub-controllers communicate through message passing, handled by proper communication and synchronization protocols. The parallel controllers peculiarity is that they are responsible also for the scheduling. Shifting the scheduling phase at run time allows to exploit the parallelism. However, such architectures present a significant communication and synchronization overhead. Moreover, since they are usually obtained by FSM decomposition, the overall area increases. FSM decomposition is a top-down approach that, starting from a centralized machine, divides it into sub-machines. Often, non-overlapping decomposition results impossible. Hence, in such cases, portions of circuit must be duplicated.

From these considerations, it is possible to infer some general conclusions. First of all, the problem of extracting parallelism in HLS is composed of three sub-problems: identification, representation and exploitation. Several

aspects must be considered to address the entire problem. For example, the definition of the final architecture represents a crucial aspect for parallelism exploitation. The architectural solutions proposed so far to address such problem result too expansive. Moreover, the designer should take into account that the controller model definition and the scheduling task are tightly connected problems. Indeed, the tasks performed by these two components are very close, since the latter must establish when an instruction has to be executed, while the former is responsible for enabling instructions execution, according to the decisions taken by the scheduler. Moreover, in architectural models such as parallel controllers, the controller and the scheduler are merged in a single entity. Finally, the adoption of C-like input languages lead to the need of a proper IR to represent the parallelism. Such IR must be carefully defined according with the choices made for the final architecture. This thesis work proposes an HLS design methodology to build a parallel controller structure able to handle with the parallelism problem. The input language adopted to describe the specification is, as common, C-like. However, a novel IR will be defined to represent the inherent parallelism. The idea at the basis of this work is that the controller can be obtained with a bottom-up approach. In this way, the entire FSM construction is not needed anymore, avoiding also the problems related to decomposition techniques. The proposed parallel controller is able to enable an instruction execution as soon as the instructions which it depends on have been executed and resource constraints are satisfied. It is able to automatically identify the conditions that enable instructions execution on the basis of any scheduler decision. Such conditions are obtained by the analysis of the proposed IR. This means that part of the scheduler's tasks is shifted on the controller. More in detail, the scheduler has to compute the priorities for the instructions, induced by

the pre-defined order relation. Then, it has to assign a control step in which each instruction will be executed. In the proposed architecture, this last task is performed at run-time by the controller, while the scheduler statically performs only the first one. In conclusion, the scheduler computes the priority associated with each instruction at compile-time, while the controller dynamically activate instructions execution on the basis of the information obtained from the analysis of the proposed IR. Then, since the resources binding could bound multiple independent instructions on the same resource, the statically-computed priorities are used to establish which of them will be executed as first, implementing a mechanism for conflicts resolution.

The proposed controller structure is not composed of a set of communicating local FSMs. It is instead closer to a behavioral description of the specification composed of several modules, interacting according to a token-based scheme, which is a very simple communication paradigm. This choice prevents to incur in the communication overhead problem. Finally, making the controller able to identify by itself the conditions that enable instructions execution led to face another aspect of the parallelism issue, that is the *parallelism formalization*. Such problem consists in how to formalize the information about the parallelism, obtained from the analysis of the proposed IR, in such a way to make the controller able to understand and automatically manage this information.

The main contributions of this thesis work can be thus summarized as follows:

- a new IR, namely the ***Parallel Controller Graph*** **(PCG)**, has been proposed to address the *parallelism representation* problem. The PCG has been obtained from the analysis of a well known IR, i.e. the Program Dependence Graph (PDG) [22]. The PDG shows the so called minimum control and data dependencies of a program. The PCG extends the PDG with the addition of the minimum control flow

information.

- an ***analysis of the PCG*** has been performed to address the *parallelism identification* problem. Such analysis aims to identify the minimum set of conditions that must be satisfied to enable an instruction execution.

- the information obtained from the analysis of the PCG has been formalized, addressing the *parallelism formalization* problem. More in detail, since the controller is able to manage signals, for each instruction, such information has been formalized as a composition of signals, indicating when the execution can start, i.e. when the above mentioned conditions enabling the instruction execution are satisfied. Such composition of signals defines a function, namely the ***Activation Function*** **(AF)**, associated with each instruction.

- a ***parallel controller structure*** has been proposed, addressing the *parallelism exploitation* problem. This innovative controller structure led to the introduction of different interacting modules, that have been defined. After that, the parallel controller has been obtained through a bottom-up approach.

- a series of ***optimizations*** have been proposed to reduce the logic allocated to implement the activation functions.

The thesis is organized as follows. In Capitolo 1, the high-level synthesis problem is introduced, and the most common architectural models are described. Capitolo 2 provides an overview of the current state of the art of the high-level synthesis techniques, focusing on three main features: the language adopted for the specification description, the application domain and the final architecture. In Capitolo 3, the proposed parallel controller is presented. In particular, the proposed Parallel Controller Graph is introduced, Activation Functions are formalized, and a series of optimization techniques that

can be applied to the parallel controller are finally presented. Experimental results are reported in Capitolo 4. Finally, Capitolo 5 draws concluding remarks and possible future works are proposed to extend its capabilities.

Chapter 1

Preliminaries

In this Chapter some preliminary concepts, at the basis of this thesis work, will be presented. However, for a fully comprehensive theoretical treatment on such topics, the reader should refer to specific works.
The Chapter is organized as follows: in Section 1.1 the high-level synthesis (HLS) process is introduced, then Section 1.2 presents the most widespread architectural model (FSMD) produced by means of HLS. The subsequent sections, from Section 1.3 to 1.7, aim to describe the various phases in the synthesis flow, according to the typical flow model proposed in Figure 1.3, as grouped in three main sections: front end (Section 1.4), synthesis (Section 1.5, 1.6), back end (Section 1.7).

1.1 Introduction to High Level Synthesis

High-Level Synthesis (HLS), also known as *behavioral synthesis* or *algorithmic synthesis*, is a design process that, given an abstract behavioral specification of a digital system and a set of constraints, automatically generates a Register-Transfer Level (RTL) structure that implements the desired behavior [55]. In Table 1.1, the typical HLS *inputs* and *outputs* are shown; their functionalities will be presented in the following.

INPUT	*OUTPUT*
- Behavioral Specification	- RTL Implementation Structure
- Design Constraints	(Datapath)
- Optimization Function	- Controller (FSM)
- Resource Library	

Table 1.1: *Typical HLS inputs and outputs*

Inputs:

- The *behavioral specification* consists in an untimed or partially timed algorithmic description in high-level language (such as C language), that is transformed in a fully timed and bit-accurate RTL implementation by the *behavioral synthesis flow*.

- The *design constraints* impose some limitations to the synthesis flow; for example they can specify the upper bound to the number of different instances of each resource, the cost limitations, the minimum performance level in terms of latency of the specification execution, the maximum area occupancy or the power consumption limit. In general, the design constraints represent targets that must be met for the design process to be considered successful.

- The *optimization function* is a cost function whose argument represents the design target to optimize. The most common features desired to be maximized/minimized are execution time, area and power consumption. Clearly the optimization function, as it generally happens, can depend on two or more variables. In such case, it is needed to manage a multi-objective optimization process, where a global optimum solution could not exists at all. Instead, a set of designs, all satisfying the constraints, for which is not possible to establish who is better, can co-exist. Given a set S of feasible solutions, all functions of n parameters, we say that $s \in S$ is *Pareto optimal* if there not exists another solution

s'∈S that improves one or more parameters without worsen at least another one.

- The *resource library* contains a collection of modules from which the synthesizer must select the best alternatives matching the design constraints and optimizing the cost function.

Output: it is a register transfer level description of the designed architecture, usually consisting of

- a *datapath* is the entity which *performs* the computation between primary inputs, which provide the data to be elaborated, and primary outputs, which return the results of computation.

- a *controller* is the entity which *manages* the computation, handling the data flow in the data path by setting control signals values, such as the FUs, registers and muxes inputs selection (see Figure 1.1). Controller inputs may come from primary inputs (control inputs) or from data path components (status signals as result of comparisons). It determines which operations have to be executed at each control step and the corresponding paths to be activated inside the datapath.

Different controller implementations approaches are feasible; generally it is implemented by hardwired logic gates, but it is also possible to build a programmable controller using the memory of specific custom processors. A common model to represent the controller is the Finite State Machine (FSM). The resulting architectural model, detailed in the following section, is known as FSM and Datapath (FSMD).

1.2 FSMD Architectural Model

The most common architectural model in high level synthesis is the finite state machine with datapath, as shown in Figure 1.1.

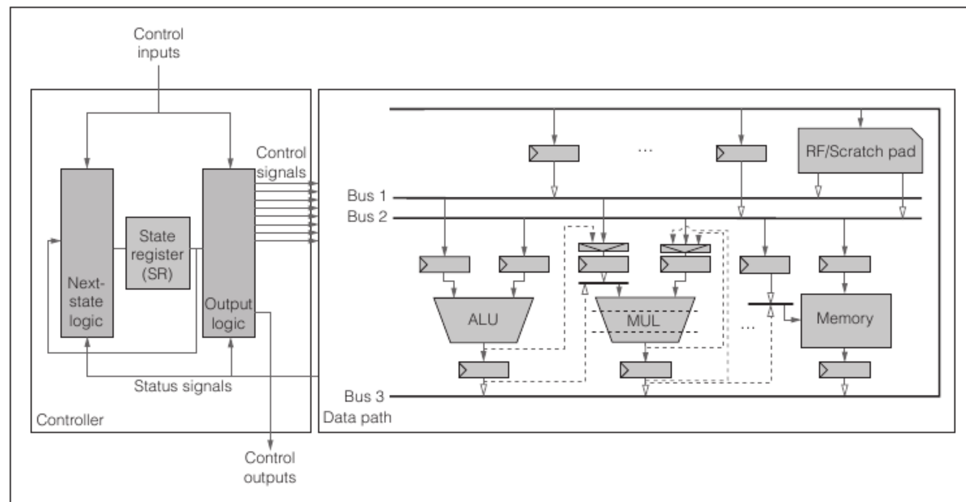

Figure 1.1: *Typical Architecture composed of a Finite State Machine (FSM) and a Datapath.*

1.2.1 Data Path

The data path includes a set of hardware resources, i.e. storage, functional and interconnection units, and defines how those modules are connected each other. All the RTL components can be allocated in different quantities and types, and can be customly connected at design time through different interconnection schemes, e.g. mux or bus based. Different architectural solutions could be adopted, allowing optimizations such as:

- multicycling: if each instruction requires exactly one clock cycle, then the clock cycle is lower-bounded by the higher required execution time; to overcome this issue, expensive instructions in terms of delay are executed through subsequent clock cycles;
- chaining: it is another solution to the previous problem; instead of reducing the clock cycle, instructions requiring less time are executed subsequentially in the same clock cycle;
- pipelining: instructions are divided in stages, and the clock cycle set to the time required to execute the slower one; if stages are obtained in

a way such that there is no concurrency on the resources that execute them, than different stages of different instructions may be executed in the same clock cycle.

Formally, a *data path* DP can be described as a graph $DP(M, T, I)$, where

- $M = M_o \cup M_s \cup M_i$ is the set of nodes, corresponding to the DP modules, i.e. instances of library components, where
  - M_o is the set of functional units such as adders, ALUs and shifters;
  - M_s is the set of storage elements, as registers, register files and memories;
  - M_i is the set of interconnect elements such as tristates, buses and multiplexers;
- $I \subseteq M \times M$ is the set of graph's edges, i.e. interconnection links.

1.2.2 Finite State Machine

The Finite State Machine (FSM) represent one of the most common models applied in architectural synthesis. Even though they can describe different kinds of sequential machines, FSMs are typically used for synchronous ones. *Synchronous machines* are characterized by the presence of an impulsive signal, i.e. the clock, propagated over the whole circuit, that determines the moment in which the inputs must be evaluated to possibly cause the transition from one state to another of the FSM. Hence, the order in which the inputs are received does not affect the execution, provided they come within the clock cycle. Instead, in the case of *asynchronous machines*, a global temporization does not exists, and an explicit communication protocol is required to ensure the computation correctness. The asynchronous machines can be classified in two main categories: *level machines*, in which the system state transitions are caused by changes in the level of the input variables, and *impulsive machines*, in which the presence or absence of impulses causes

such transitions.
Formally, a *finite state machine* is defined as the tuple $M = (I, O, S, S_0, R)$, where

- I represents the input alphabet,
- O denotes the output alphabet,
- S denotes the set of states,
- S_0 denotes the initial state,
- R denotes the global relation.

The *global relation* R is defined as $R \subseteq S \times I \times S \times O \to \{0, 1\}$ such that $R(i, u, s, t) = 1$ iff given as input $i = (i_1, i_2, ..., i_n) \in I$, M goes from the current state $s = s_1, s_2, ..., s_k \in S$ to the next state $t = t_1, t_2, ..., t_k \in S$ producing as output $o = o_1, o_2, ..., o_k \in O$
The main FSM controller components are:

- a state register (SR), that stores the current state of the FSM model describing the controller's operation;
- the next state logic, that computes the next state to be loaded in the SR;
- the output logic, that generates the control signals.

State-Transition Graph

A finite state machine M can be represented by its corresponding *state-transition graph* $G(S, E)$ where

- nodes $s \in S$ are the states of M,
- edges $e \in E \subseteq S \times S$ denote transitions between states.

State-Transition Relation

Given the global relation R and an input i,the *state-transition relation* determines the relationship between current state and next state. It is defined as $\Delta(i, s, t) = \exists o R(i, o, s, t)$. The *state-transition function* is usually denoted by $\delta(i, s, t)$.

Output Relation

Given the global relation of M, the initial state S_0 and an input i, the *output relation* gives the output value of M. It is defined as $\Lambda(i, o, s) = \exists t R(i, o, s, t)$. The *output function* is usually denoted by $\lambda(i, o, s)$.
Starting from the above definitions, it is possible to classify the different types of FSMs in three classes:

- Autonomous - The input alphabet is the empty set (e.g. counters).
- State based - Also known as Moore's machines, their output relation Λ depends only on the current state.
- Transition based - Also known as Mealy's machines, their output relation Λ depends on input values also.

The classical logic implementation of a FSM stores the states in storage elements (registers) while state-transition and output functions are synthesized in combinatorial logic. The typical structure of a synchronous sequential circuit, called Huffman model, is shown in Figure 1.2.

Finite State Machine with Data Path

The FSM model can be reasonably handled where variables are used to represent the different states. For instance, a 16-bit variable can represent $2^{16} = 65536$ states that needs to be stored in storage devices such as registers or register files. Adopting such encoding, the model can be defined as:

- a set Var of variables representing the states;

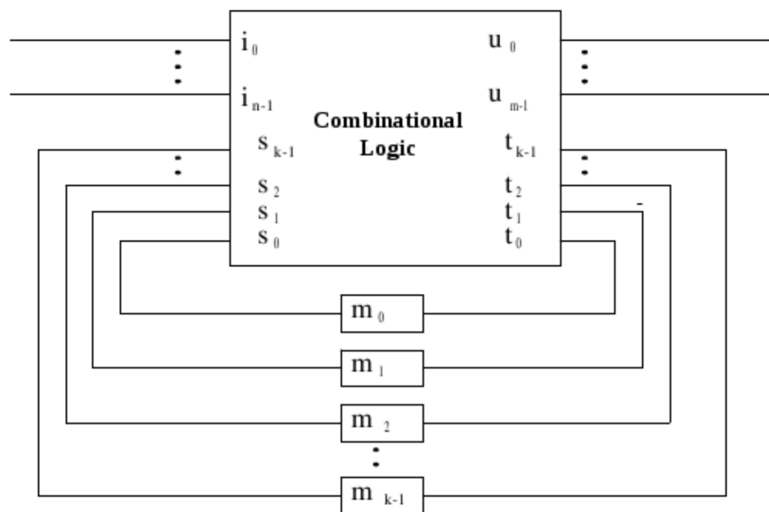

Figure 1.2: *Sequential circuit's Huffman model.*

- a set $Exp = \{\Delta(x, y, z, ...)|x, y, x, ... \in Var\}$ of functions;
- a set $Asg = \{X = e|X \in Var, e \in Exp\}$ of assignments;
- a set $Stat = \{(a, b)|a, b \in Exp\}$ of *state variables*, i.e. a relation among the Exp set.

Now the *Finite State Machine with Data Path* can be defined as the tuple $< S, I \cup B, O \cup A, \Delta, \Lambda >$, where:

- S, Δ, Λ denote the same sets defined for a FSM;
- $I \cup B$ denotes the input language; it is an extension of the set I defined for the FSM, in order to include some of the state variables $b \in B \subseteq Stat$;
- $O \cup A$ represents the output language, including some assignments $a \in A \subseteq Asg$

FSMDs can be adopted both to describe a design at RT level, or, at higher levels of abstraction, to represent a producer/consumer process where inputs are consumed and outputs are produced. Complex systems could be viewed

as processes compositions, where each process is modeled as FSMDs and communicates with the other ones. Communication is intent to be between control units, between data paths and between control unit and data path. The number of signals and the temporal relations between them during communication define a protocol, e.g. *request-acknowledge handshaking* protocol [7]. If the signals are managed by a unique clock, the system is said to be synchronous. On the contrary, if the clock rates are different, the system is said to be asynchronous.

1.3 High Level Synthesis Flow Overview

As mentioned above, the high-level synthesis is typically composed of different tasks, as shown in Figure 1.3. There exists many approaches in literature that perform these activities in different orders, using different algorithms. In some cases, several tasks can be combined together or performed iteratively to reach the desired solution. In all the cases, the HLS flow steps can be grouped in three main macro-tasks:

- Front End:
  performs *lexical processing*, *algorithm optimization* and *control/dataflow analysis* in order to build and optimize an internal representation (IR) to be used in the subsequent steps. Internal representations describe the specification underlining specific properties, on which a given task of the synthesis phase will focus on.

- Synthesis:
  in this phase the design decisions are taken, to obtain a RTL description of the target architecture that satisfies the design constraints. The number and type of hardware modules is established and each instruction is scheduled and assigned to one of the available resources that can execute it.

- Back End:

the resulting design is derived and reproduced in a hardware description language.

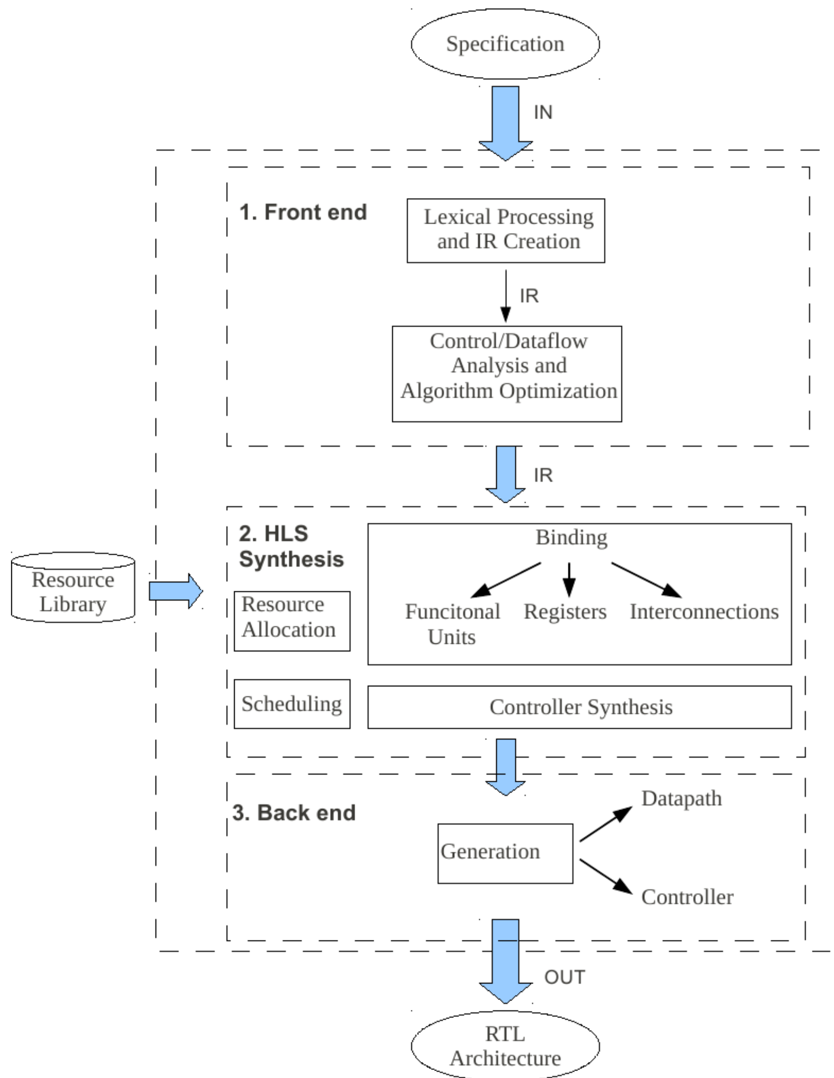

Figure 1.3: *Typical HLS flow.*

In the next sections, each task of the HLS-flow will be presented. Moreover the controller synthesis step will be deepened, since it is the focus of this thesis work.

1.4 Front End

An Intermediate Representation (IR) can be described as an interface between a source language and a target language. Such notation should describe source code properties independently with respect to the source/target languages details. Given as input a behavioral specification, the front end translates it in a proper internal representation, performing lexical processing, algorithm optimization and control/dataflow analysis.

1.4.1 Lexical Processing And Intermediate Representation Creation

The first step in the HLS flow consists in the source code parsing, that translates the high level specification into an internal representation (IR), as common in conventional high level language compilation, to abstract from language details. This is why it is common to refer to this step (together with the following one) as compilation phase. The resulting IR is usually a proper graph representation of the parsed code, and can be optimized or transformed producing several different additional representations of the same specification, as described in Section 1.4.3.

1.4.2 Control/Dataflow Analysis and Algorithm Optimization

The previous step provides a formal model that exhibits data and control dependences between the operations. Such dependencies are generally represented in a graph notation: the most widely adopted ones, that will be later discussed in Section 1.4.3, are the CFG (Control Flow Graph), the DFG (Data Flow Graph) and the CDFG (Control Data Flow Grah), but several other ones already exist. The control/dataflow analysis is a prerequisite for the subsequent steps, but it also allows further optimizations to better exploit the available parallelism. These optimizations are the same ones widely used

in optimizing and parallelizing compilers, such as dead code elimination and constant folding. Between these, one of the most impacting in this work is the Static Single Assignment (SSA) form transformation of the source code, that will be presented in Section 1.4.4, since it is common to refer to code translated in SSA as a kind of internal representation.

1.4.3 Graph-Based Representations in High Level Synthesis

In high level synthesis different graph based IRs are used, because of different tasks could take advantage of using a specific representation that better express a specific source code property. Starting from the definition of graph, this section will introduce the most common graph-based IRs used in HLS.

Graph

A graph $G(V, E)$ is characterized by

- a set of nodes V; in HLS such vertices $v \in V$ usually denote instructions, or sets of instructions to be executed.
- a set of edges E; each edge $e \in E$ represents a relationship between the source and the target nodes.

Control Flow Graph

The *Control Flow Graph* (CFG) is a graph modeling a (sub)program [13]. This kind of internal representation is commonly used by compilers for its suitability in static analysis, such as liveness analysis. Each node of the CFG represents an instruction, and each edge $p \to q$, oriented from p to q, denotes that the instruction q can follow p in the execution order: p is said to be a *predecessor* of q ($p \in Pred(q)$) and similarly that q is a *successor* of p ($q \in Succ(p)$).
The *entry point* is a node without predecessors: $Pred(entry) = \emptyset$.

Similarly, the *exit point* is a node without successors: $Succ(exit) = \emptyset$.
Non-conditional instructions have exactly one successor, while conditional ones have two or more successors and in the CFG they correspond to bifurcation points.
Finally a node with two or more predecessors is a *confluence point*.
In Figure 1.4 the CFG of one possible implementation of Euclid's algorithm is presented as an example.

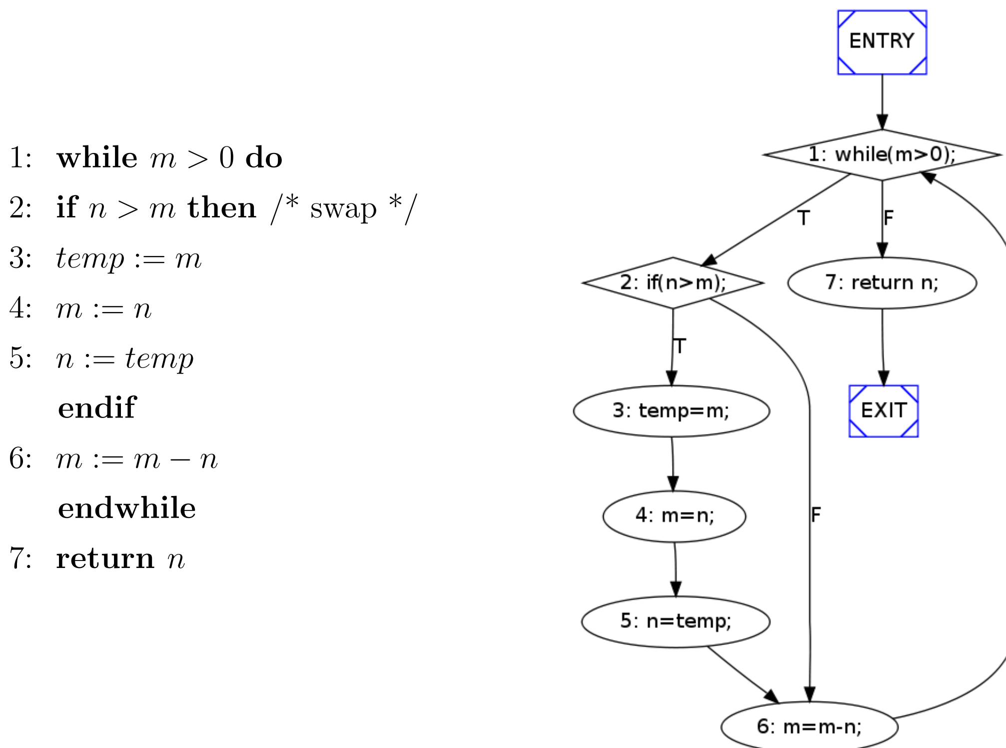

Figure 1.4: *Pseudo-code and CFG of Euclid's algorithm*

The CFG may also be viewed as the state-transition graph of a Finite State Automaton (FSA) that recognizes all the execution traces of the program. In Figure 1.5 the FSA representation of Euclid's algorithm CFG is shown. Such automaton has a terminal alphabet I, i.e. the set of all the instruction labels, and recognizes the regular language L(A) that is the set of strings $x \in I^*$ which label a path from the initial state of the graph to a final state. Those paths represent the potential execution traces of the (sub)program.

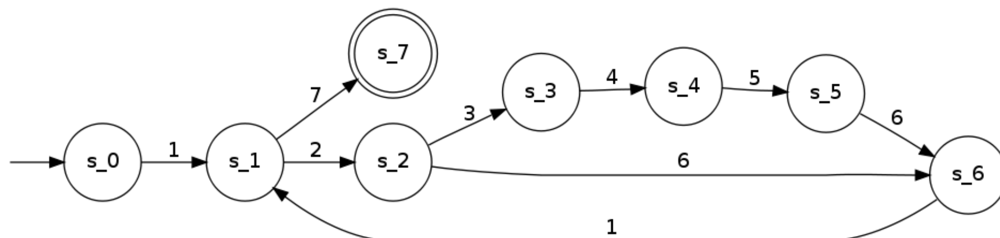

Figure 1.5: *FSM representation of Euclid's algorithm CFG.*

The language L(A) could be expressed by a regular expression (RE).
A RE for the Euclid's algorithm example is: $1(2(345|\epsilon)6)^{*}7$.
It's important to note that CFGs may be constructed at different granularity levels, such as instruction level or basic block level.

Control Dependencies Graph

The Control Dependencies Graph (CDG) is the graph representation of the control dependence relation among the nodes of the CFG. Let x be a conditional instruction in the CFG, and $v, u \in Succ(x)$. A node *y is control dependent on x via v if*:

- every path $x \to v \dashrightarrow exit$ traverses y ,
- $\exists$ a path $x \to u \dashrightarrow exit$ that does not traverse y.

Such control dependence is expressed in the CDG with the presence of an oriented edge $x \to y$.
It is also possible to define the control dependence starting from the Post-Dominance relation, dual concept of the Dominance relation.

Dominance and Post-Dominance Relations

Given a flow graph G, a node v *dominates* a node n if m occurs before n on every directed path from the entry node to n. In other words each trace reaching n cannot avoid to execute v before n. The dominance relation is reflexive: every node dominates itself. It is also transitive and anti-symmetric,

thus the dominance relation is a partial order, and it is representable as a tree. Dually, a node w is *post-dominated* by a node n if every directed path from w to the exit node contains n. The post-dominance relation is irreflexive: a node never post-dominates itself. Computing the post-dominance relation over a CFG G is the same of computing the dominance relation over the reversed graph G', thus the post-dominance tree coincides with the dominance tree of G'.

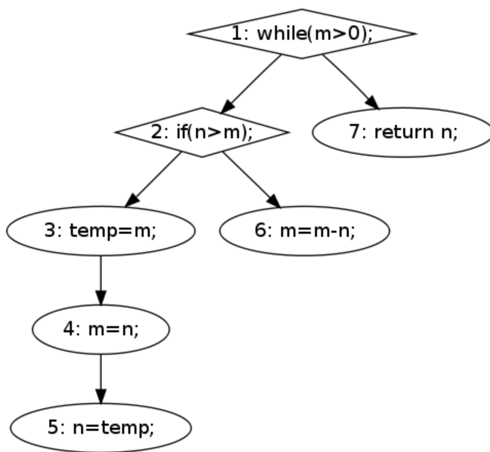

Figure 1.6: *Euclid algorithm dominance tree.*

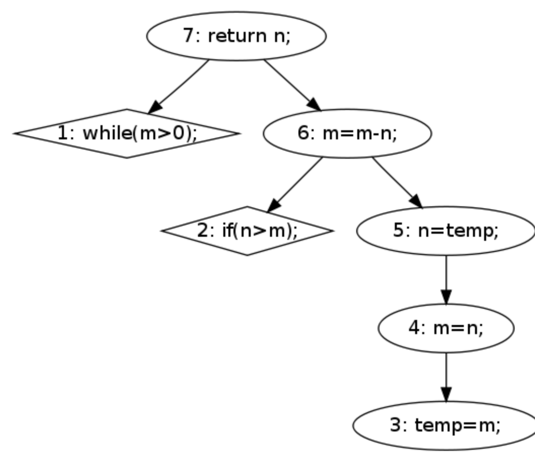

Figure 1.7: *Euclid's algorithm post dominance tree.*

Once introduced dominance and post-dominance relations, the control dependence relation can be defined as follows. Given a control flow graph G and two nodes $x, y \in G$, y is said to be control dependent on x if and only if:

- y post-dominates a successor of x;
- y does not strictly post-dominates x.

Saying that y is control dependent on x means that x is a point in which a divergence in the execution flow can occurr: there are execution paths to the exit node containing the node y, others that bypass it. As said before each edge $x \rightarrow y$ in the CDG represents that y is control dependent on x. Figure 1.8 reports the CDG of the Euclide's algorithm example.

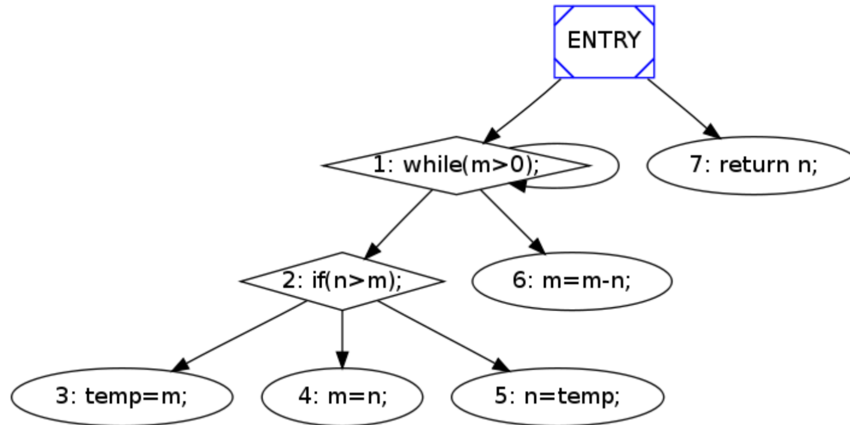

Figure 1.8: *Euclid's algorithm CDG.*

Data Flow Graph

Given a sequence of instructions I and two instructions $i, j \in I$, where $i < j$ in the sequence ordering, if i defines or uses a variable defined or used by j, there is a data-dependence between them. It is possible to identify four types of data dependencies:

- Flow dependencies, or Read After Write (RAW)
- Anti dependencies, or Write After Read (WAR)
- Output dependencies, or Write After Write (WAW)
- Input dependencies, or Read After Read (RAR)

Input dependencies mean that an instruction i is RAR dependent on an instruction j, but it does not need to wait until j completion to be executed. As a result RAR dependencies do not affect the scheduling. Anti and Output dependencies could constrain the scheduling of dependent instructions, but they can be removed using techniques such as *register renaming* [42]. For this reason, we often refer to RAW dependencies as "true"dependencies. The Data Flow Graph (DFG) is a graph representing such data-dependencies [31], where:

- nodes represent operations;
- edges represent the dependencies described above.

Defining a Basic Block (BB) as a sequence of instructions,starting with a label and ending with a jump, not containing any other labels nor jumps, each BB of a program has a DFG associated to. Usually DFG are constructed starting from a BB, thus not considering conditional constructs (as if-then-else constructs). DFGs are used to represent data-paths and to accomplish

average(a,b,c,d)

1: $temp1 := a + b$
2: $temp2 := c + d$
3: $temp := temp1 + temp2$
4: $avg = temp/4$

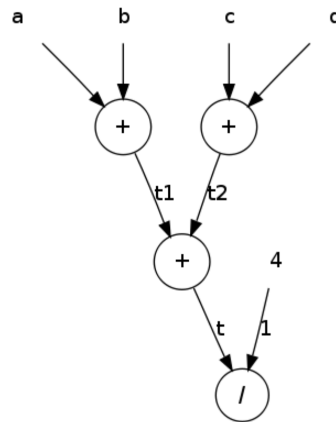

Figure 1.9: *Pseudo-code and DFG of a program that computes the average of 4 numbers*

scheduling tasks. In Figure 1.9 is reported the pseudo-code of a function performing the average of four numbers (taken as input), and the corresponding CFG.

Data Dependencies Graph

Even the Data Dependence Graph represents data dependencies between instructions, but unlike the CFG it does it at a higher level, representing a subprogram rather than a BB. The next example refers to a subprogram that takes as input two estimates (x1, x2), their corresponding expected values (y1, y2) and a threshold t, computes an error function and returns true if it is lower than the threshold. Pseudo-code and the corresponding DDG are

shown in Figure 1.10. Since both DDG and DFG represent data-dependencies that express the data-flow, often we refer at them both as DFG.

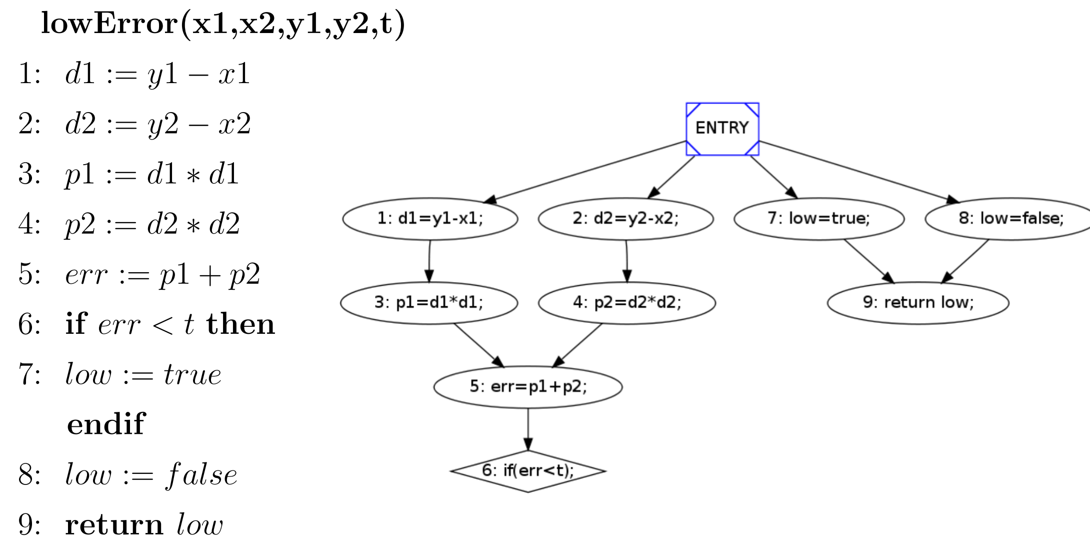

Figure 1.10: *Pseudo-code and DDG of “lowError”function*

Control Data Flow Graph

Control Data Flow Graphs (CDFGs) are built to capture both control and data flows, in a unique structure [76]. The CDFG is obtained starting from the CFG and the DFGs of a given program. Singe DFGs generally refer to basic blocks, the starting CFG is computed at basic block granularity level. It is common to represent CDFGs including additional triangle-shaped nodes, that represents bifurcation and confluence points in the control flow. In Figure 1.11 the CDFG of the “lowError“ function example is reported; in the graph it is possible to identify the preliminary basic block partitioning of the original code.

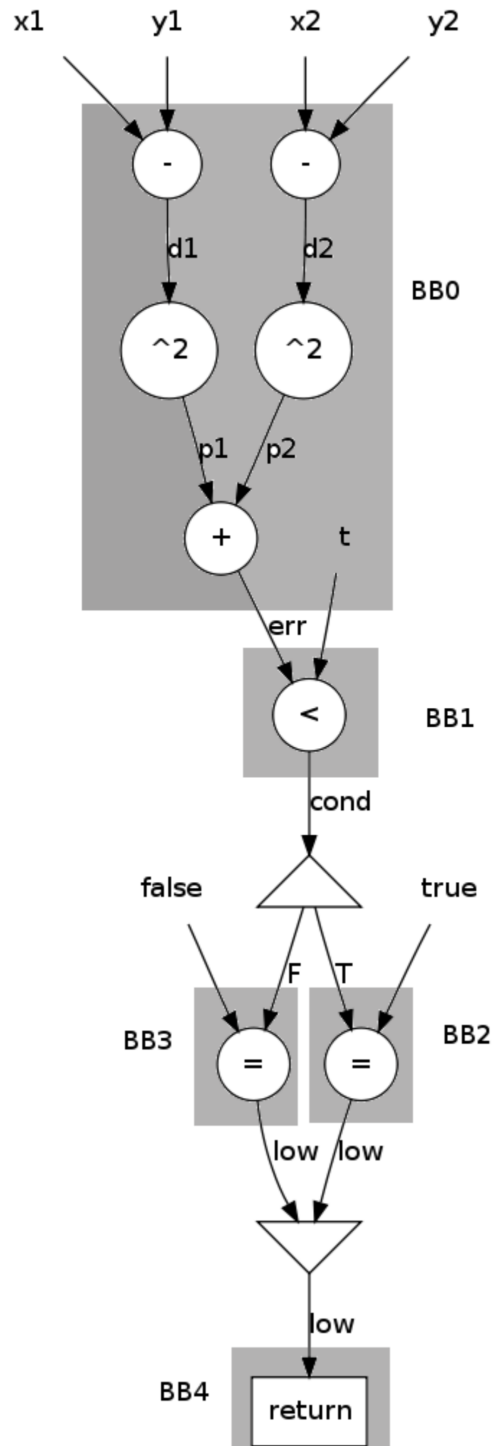

Figure 1.11: *CDFG of the "lowError"function example.*

Sequencing Graph

As the CDFG does, the sequencing graph describes both control and data dependencies, modeling control structures as diramations and loops with a hierarchical structure. It is possible to define a sequencing graph as a *graph hierarchy*, formed by several subgraphs (*models*) represented as DFGs that may be reached by different nodes, thus representing *model calls*. Link nodes are introduced in this representation, in order to connect different entity in the hierarchy. Model calls are represented as nodes pointing to entities in a lower level in the hierarchy. Nodes in a sequencing graph can:

- wait for their execution;
- being executed;
- have been executed.

When a node's execution starts, we say that the node is activated, or fired. The execution can start only when each predecessor of the node has been executed. Activating a source node of a subgraph, the whole subgraph is considered activated and its execution starts.

Program Dependence Graph

The Program Dependence Graph (PDG) [22] is a directed graph that represents both data and control dependencies of a given (sub)program. It is obtained merging the CDG and the DDG, thus each edge node represent an instruction (as in the starting graphs) and each edge represent a control (from the CDG) or data (from the DDG) dependence. In Figure 1.12 an example of PDG is shown, where solid edges denote control dependencies and dotted ones data dependencies. Notice that even in this case an entry node is added; in the example, to make the graph clearer, it is connected only to nodes in the PDG without predecessors; anyway, it is the only node in the PDG without incoming edges.

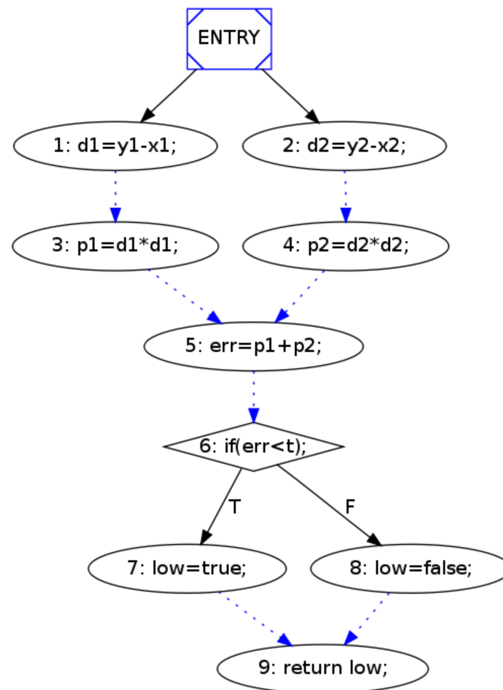

Figure 1.12: *PDG of the "lowError"function example.*

System Dependence Graph

A System Dependence Graph (SDG) is a collection of PDG, connected through call and parameter edges. It is useful to have a complete view of the dependencies between statements in the whole program, and to more easily detect the inherent parallelism.

Static Single Assignment Form

Once presented the main graph internal representations, it is now introduced the *static single assignment form* of the source code, commonly considered as another kind of IR. A subprogram is in Static Single Assignment (SSA) form iff each use of a variable has a single definition point [70]. SSA form is a desirable feature for several reasons, for example:

- it simplifies dataflow analysis and related optimizations;

- uses and definitions of variables in SSA form relate in a useful way to the dominator structure of the control-flow graph;
- multiple definitions of the same variable in the source program become different variables in SSA form, eliminating needless relationships; for example, instruction 4 in the example code proposed in Figure 1.13 exhibits a relation with both instructions 1 and 2, defining the variable a used by 4. The corresponding code, written in SSA form, does no longer hold the irrelevant relation between instruction 1 and 4.

1: $a := x + y;$	1: $a_1 := x_1 + y_1;$
2: $a := a + 3;$	2: $a_2 := a_1 + 3;$
3: $b := x + y;$	3: $b_1 := x_1 + y_1;$
4: $c := a + b;$	4: $c_1 := a_2 + b_1;$

Figure 1.13: *Example code and corresponding translation in SSA form.*

Since source code generally don't come in SSA form, many front-ends offer this kind of transformation as a feature. It is obtained, in straight line code, this way: each definition of a variable (e.g. a) is modified to define a new one (e.g. $a_1, a_2, ..., a_n$), and each use of the variable is modified to use the most recently defined version (see Figure 1.13). In the presence of diramation constructs, the transformation is a little more complex. If the same variable a is defined in two different branches of the diramation point, SSA transformation will introduce two new variables a_1 and a_2. If, in the original code, variable a is used at the confluence point, then in the transformed code it is needed to establish which variable between a_1 and a_2 must be considered. This is obtained introducing a notational fiction, called $\phi - function$. In the proposed scenario, at the confluence point a new assignment is introduced, $a_3 = \phi(a_1, a_2)$: the $\phi-$function is defined in order to denote that variable a_3 can assume the value a_1 or a_2 depending on the branch followed at execution time. A clarifying example is shown in Figure

1: $a := x + y$;
2: **if** $a < N$ **then**
3: $a := a + 1$;
else
4: $a := a + 1$;
endif
5: $c := a + b$;

1: $a_1 := x_1 + y_1$;
2: **if** $a < N$ **then**
3: $a_2 := a_1 + 1$;
else
4: $a_3 := a_1 - 1$;
endif
5: $a_4 := \phi(a_2, a_3)$;
6: $c_1 := a_4 + b_1$;

Figure 1.14: *Example code and corresponding translation in SSA form, with $\phi -$ function insertion.*

1.14.

1.5 Synthesis

The synthesis section consists of different steps strictly connected each other, so it is usual to consider them as a unique core in the HLS flow. This steps can be summarized in:

- Scheduling,
- Module Allocation,
- Binding.

A desirable feature for HLS is to estimate as soon as possible timing and area overheads, so that later steps could optimize the design. A feasible approach is to start from one of the above mentioned tasks and consequently accomplish the other ones. Then the obtained results could be used to modify the solution of previous steps: in this way the final solution, optimized towards a performance metric, is built incrementally. Another possibility is to fulfill the tasks partially, and complete them as results of other steps are available. For example functional units could be allocated in a first time,

and the interconnection allocation could be performed after the binding or scheduling tasks. Clearly, performing allocation, scheduling and binding as a unique task makes the synthesis a too complex process to be applied to real-world specifications, since each them is NP-complete. The choice between different ordering possibilities is dictated from the design constraints and tool's objectives. For example, under resource constraints, allocation could be performed firstly and scheduling could try to minimize the design latency. Instead, in time constrained designs, allocation could be performed during the scheduling; the scheduling process, in this case, could try to minimize the circuit's area while meeting the timing constraints.

1.5.1 Scheduling

The scheduling task introduces the concept of time: according to the data dependencies extracted by the front end all the operations are assigned to specific clock cycles. Also the concept of parallelism is introduced: if the dependences and the resource constraints allow it (i.e. if there are sufficient instances of each one), more than one instruction could be scheduled in the same clock cycle. A common approach addressing the scheduling problem can be summarized as follows:

- the Data Flow Graph (DFG) of the specification is considered;
- the Data Flow Graph is scheduled, producing a graph $G(V_0, E, C)$ where:
  - $v \in V_0$ are the nodes of the DFG, i.e. the operations to be executed;
  - $e \in E$ are the edges of the DFG, representing the data flow;
  - $c \in C$ are cycle steps.
- a scheduling function $\theta : V_0 \rightarrow \Pi(C)$ assigns to each node $v \in V_0$ a sequence of cycle steps, where $\Pi(C)$ is the power set of C, i.e. the set of all the subset of C.

As said before, a schedule could be differently constrained, e.g. time or resource constrained. As an example, considering the DFG in Figure 1.9 in which only additions and divisions appears, a possible schedule under resource constraints (1 adder and 1 divider available) is shown in Figure 1.15, assuming that each operation needs one cycle step to be executed.

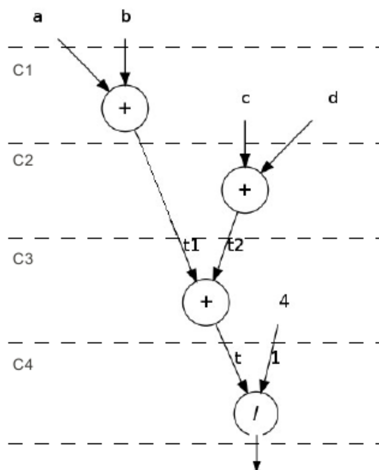

Figure 1.15: *"average"function example scheduled CFG under resource constraints.*

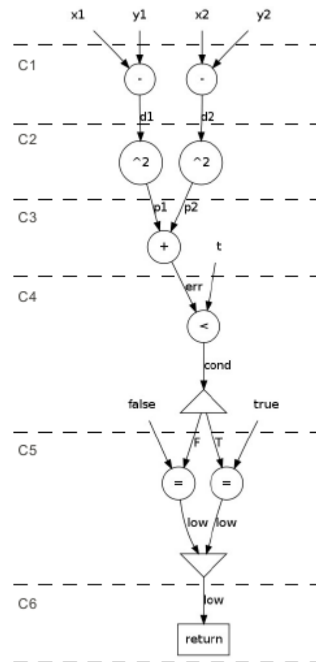

Figure 1.16: *"low error" function example scheduled CDFG.*

In the presence of control constructs such as if-then-else, the CDFG may be used for scheduling purposes. Figure 1.16 represents a schedule obtained starting from the CDFG in Figure 1.11: instructions in step C5 are executed once the branch condition (scheduled in C4) is evaluated. It is clear that those instruction are *mutually exclusive*, denoting that only one of them will be executed, on the basis of the branch condition. It is possible to define a *mutually exclusion function* as $m : V_0 \rightarrow \Pi(\mathbb{N})|m(v_i) \wedge m(v_j) = 0$ when v_i and v_j are executed under mutually exclusive conditions. It could be useful to underline that given a set of instructions and a set of dependencies

between them, several schedules could be obtained. Many algorithms to face the scheduling problem have been proposed in literature, managing either timing and resource constraints.

1.5.2 Resource Allocation

A set of hardware resources is established to adequately implement the design, satisfying the design constraints. Allocation defines the number of instances and the type of different resources from the ones available in the resource library, which describes the relation between the operation types and the modules. Since different hardware resources have different characteristics, such as area, delay or power consumption, usually this informations is included in the resource library, to guide both the allocation and the other related tasks. Obviously, at least one component for each operation in the specification model must be selected.

Resource Library

A library $\Lambda(T, L)$ is characterized by:

- a set T of operation types;
- a set L of of library components (i.e. modules);

The library function $\lambda : T \to \Pi(L)$, where as usual $\Pi(L)$ denotes the power set of L, establishes which modules $l \in L$ could execute operations of type $t \in T$. On the other hand, the function $\lambda^{-1} : L \to \Pi(T)$ defines the *operation type set of l*, written $\lambda^{-1}(l)$, i.e. the subset of operation types that a module $l \in L$ can execute. Given two operation of type $t_1, t_2 \in T$, if $t_1, t_2 \in \lambda^{-1}(l)$, then they can share module l. Moreover, $\lambda(t_1) \cap \lambda(t_2)$ describes the subset of modules that can be shared among operations of type t_1 and t_2.
In Section 1.2.1 the data path as been defined as a graph $DP(M_o \cup M_s \cup M_i, I)$, thus allocation task must determine the components belonging to each module set M_k. This task defines the allocation functions for each module set.

Functional Units Allocation

The *module allocation function* $\mu : V_0 \to \Pi(M_0)$ determines which module performs a given operation. Clearly, an allocation $\mu(v_i) = m_j, v_i \in V_0, m_j \in M_0$ is a valid one iff module m_j is an instance of $l_j \in \lambda(t_i)$, with t_i operation type of v_i, i.e. m_j can execute v_i.

Registers Allocation

Values produced in one clock cycle may be consumed in another one, and in this case such values must be stored in registers or in memory. Liveness analysis can allow different variables sharing a register, revealing if their life intervals overlaps or not, in order to reduce the number of registers, and the design overhead in terms of area. Even in this case, techniques developed in compiler theory are successfully applicable. In particular, after the scheduling, each edge that crosses a cycle step boundary represents a value that must be stored. Thus the scheduled DFG should be transformed to take in account such situation. Given a scheduled DFG $G(V_0, E, C)$, the *storage value insertion* is a transformation $G(V_0, E, C) \to G(V_0 \cup V_s, E', C)$ which adds nodes (storage values) $v \in V_s$ such that each edge $e \in E$ which traverse a cycle step boundary is connected to a storage value. The *register allocation function* could now be defined as $\psi : V_s \to \Pi(M_s)$; it identifies the storage modules holding a value from the set V_s.

Interconnection Allocation

The *interconnect allocation function* is defined as $\iota : E \to \Pi(M_i)$, and describes how the modules are connected and which interconnection is assigned for each data transfer. Different solutions could differently affect the design in terms of delay, area occupancy or interconnections complexity.

Once modules are allocated it is possible to define operation and modules selection functions. The *operation selection function* $\sigma_0 : V_0 \to L$ determines

on which type of library component $l \in L$ an operation $v \in V_0$ is executed. The *module selection function* $\sigma : L \rightarrow \mathbb{N}^{|L|}$ determines the number of instances of each library component that have been allocated. Given a library component $t \in L$, $\sigma(t)$ indicates the number of instances of l.

1.5.3 Module Binding

The allocation task defines the set M of modules that composes the data path. Each module $m \in M$ is an instance of a library component $l \in L$. Given a DFG $G(V_0, E)$, each operation $v \in V_0$ must be bound to a specific allocated module m. This task takes the name of module or resource binding. A resource binding is defined as a mapping $\beta : V_0 \rightarrow M_0 \times \mathbb{N}$; given an operation $v \in V_0$ with type $\tau(v) \in \lambda^{-1}(t), t \in L$, $\beta(v) = (t, r)$ denotes v will be executed by the component $t = \mu(v)$ and $r \leq \sigma(t)$, i.e. v is assigned to the r-th instance of resource type t. Module binding must ensure that the resource assigned to an operation is available in the cycle step in which the given instruction is scheduled. When the binding is performed before scheduling, the scheduling task will take care of schedule operations in order to avoid resource conflicts. Moreover, if two operations are mutually exclusive they can be bound to the same module, even if they are scheduled in the same cycle step. Different approaches can be followed to perform the binding; in the simplest case β is a one-to-one mapping, associating each resource to one operation.

1.6 Controller Synthesis

Once the datapath is built, it is possible to define the activation signals that the controller will have to generate in order to activate the data path modules. Controller synthesis can be performed following two main approaches:

- **Microprogrammed Controller** - each state of the FSM is coded as a microinstruction that specifies the data path activation signals and

the next state; if the resulting microprogram is stored in a ROM, the next state can be represented by the ROM address of the next microinstructions (i.e. associated to the next state). If the CFG describing the specification is linear, i.e. there are not conditional nodes, the next state could be computed by a simple counter without indicating it in every microinstructions.

- **Hardwired Controller** - the controller is synthesized as a combinatory circuit and registers.

Both the models represent the abstract model of the synchronous FSM. The design complexity increases in the presence of hierarchical structures, or in control dominated specifications. Thus different approaches, depending on the design complexity, will be next summarized.

Microprogrammed controller synthesis of linear data flow graphs

Usually, a read only memory is used, having a number of words equal to the number λ of control steps. The memory addressing requires $\lceil log_2\lambda \rceil$, and it is obtained, as said in the previous section, using a simple module-λ counter with reset signal, managed by the system clock. It is possible to recognize two kinds of controller micropragramming:

- **horizontal micropragramming** - is the easiest solution: a bit of the microinstruction is associated to each activation signal towards the data path. This solution clearly increase the size n_act of the ROM words, i.e. of the microinstructions.

- **vertical microprogramming** - the n_act bits are coded, thus reducing the microinstructions size, but increasing the path between controller and data path, due to the encoders insertion. Encoding can be successfully applied only for those bits corresponding to mutually exclusive activation signals, so a completely vertical solution is seldom feasible. Thus, in practice, encoding is applied only to subsets of mutually exclusive activation signals bits.

Hardwired controller synthesis of linear data flow graphs

The controller is a typical FSM with λ states corresponding to the identified control steps, whose activation signals are provided by the output function. Since the only input of the FSM is the clock signal, the state diagram is reduced in a loop of length λ.

Controller synthesis of hierarchical sequencing graphs

Hierarchy occurs in three possible cases:

- **model calls** - To each subgraph corresponding to a model call is associated a local control unit (or control block) whose activation signal is thrown by the model calls, thus activating the counter (in the case of microprogramming) or the transitions (if it is hardwired) of the local controller. When its activation signal is nullified, the local controller halts and a reset signal is generated. The activation signal of a control block is produced by a subgraph higher in the hierarchy (the "caller"graph) and can be concurrent with other activation signals; the execution of the control block is usually concurrent with the other instructions of the caller graph.The activation signal is valid until the execution of the called model terminates. A structure model is shown in Figure 1.17.

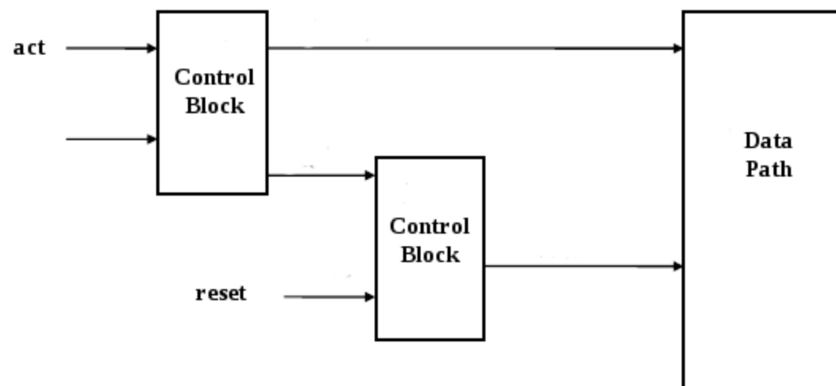

Figure 1.17: *Hierarchical FSM model.*

- **diramation** - Diramations can be treated as calls of alternative models, whose choice is dictated by the branch condition evaluation. A possible implementation (see example in Figure 1.18) should store the outcome of the condition's evaluation until the execution of the called model ends.

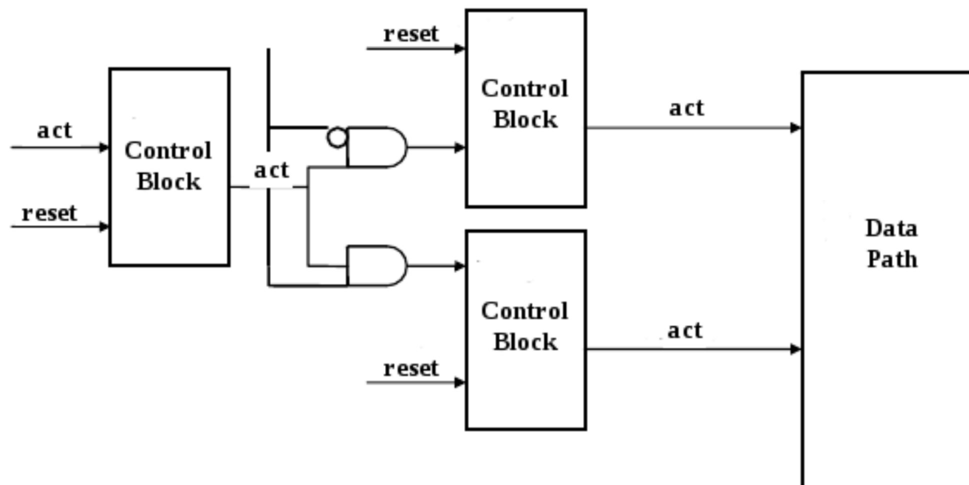

Figure 1.18: *Hierarchical controller model in the presence of diramation constructs.*

- **loops** - If the sequencing graph latency is known in advance (i.e. the number of loop iteration is determined) the controller synthesis may follow an approach similar to diramation constructs one. The loop body can be considered a model call repeated several times. Thus the activation signal of the loop body subgraph must be enabled for a time interval equal to the loop body delay multiplied for the iterations number. In the case the number of iterations is not known, during the last control step of the loop body, i.e. the last instruction to be executed, a *completion signal* must be generated to correctly implement the Hierarchical FSM.

1.7 Back End

1.7.1 Generation

The generation derives the design decisions made in the previous steps to generate a RTL model of the synthesized design, where a set of register-transfer components is adopted to represent both the data path and the controller.

Output: RTL Model

The architecture description is written out as RTL source code, then translated in a target language, and structured in such a way to optimize the subsequent logic synthesis task or to improve its readability. The RTL description could be written with different levels of detail, according to the decisions made in the binding steps. For example, a = b + c executing in state (n) could be written as shown in Figure 1.19 [19].

```
Without any binding:
  state (n): a = b + c;
  go to state (n + 1);

With storage binding:
  state (n): RF(1) = RF(3) + RF(4);
  go to state (n + 1);

With functional-unit binding:
  state (n): a = ALU1 (+, b, c);
  go to state (n + 1);

With storage and functional-unit binding:
  state (n): RF(1)=ALU1 (+, RF(3), RF(4));
  go to state (n + 1);

With storage, functional-unit, and connectivity binding:
  state (n): Bus1 = RF(3); Bus2 = RF(4);
  Bus3 = ALU1 (+, Bus1, Bus2);
  RF(1) = Bus3;
  go to state (n + 1);
```

Figure 1.19: *RTL description written with different binding details.*

RTL descriptions including only a partial resources binding are simple to be designed but force the logic synthesis process to perform the binding task

and eventually the related optimizations. On the contrary, RTL descriptions with additional binding details usually improve the predictability.

Chapter 2

State of the Art

Since the inception of information technology, the synthesis of digital circuits has been a main concern for researchers and scientific community. Over the years, the proposed design methodologies have been forced to move to higher abstraction levels due to the growing capabilities of silicon technology and to the increasing complexity of applications and architectures. Such factors, since early '70s, made more and more inadequate lower abstraction level methodologies such as Logic-level or Physic-level synthesis, and led to their overcome in favor of High-Level Synthesis (HLS).
Over the years, HLS has received more and more attention by the designers, for several reasons. One of the most important is the possibility of shorten the design cycle increasing the chance for companies of hitting the market window. Moreover, HLS led to the (at least partial) automation of the design process, significantly decreasing the development cost. Furthermore, the error rate is reduced by the presence of a proper verification phase. Finally, reviews reported in literature indicate that working at a higher level of the design hierarchy using high-level synthesis reduces the amount of code that must be developed by as much as two thirds [56]. Indeed, separating the design intent from the physical implementation avoids the tedious process of rewriting and retesting code to make architectural changes. This fact also facilitates the design space exploration process since a good synthesis system

produces several designs for the same specification in a reasonable amount of time, allowing the developers to consider different solutions and trade-offs. Generally, it is reported an overall reduction of the design effort, with respect to lower level methodologies, of 50% or more [56].
Although nowadays HLS has a long history behind, many aspects still need to be formalized and improved. A good means to better understand what has already been done, which aspects were overlooked and which gap this thesis work proposes to fill, could be to take a look to the past, tracking HLS evolution. Before this, however, a clarification about the features characterizing design methodologies in HLS will be provided in Section 2.1. According to the chronological classification proposed in [54], there are three generations in HLS evolution, in addition to a prehistoric period. Moreover, a fourth generation is prospected for the future. Such chronological HLS evolution will be traced in Sections from 2.2 to 2.5, showing how HLS is changed year over year, according to user needs. Then, in Section 2.6, current HLS generation will be deeply analyzed with the aim to understand why a new HLS generation is prospected to succeed in the next few years, and thus what are the aspects needing improvements.

2.1 Design Methodologies Features in HLS

To completely characterize an HLS design methodology, three aspects must be considered: the application domain, the specification description utilized and the final architecture obtained from the synthesis. These three factors represent the variables to carefully define before to start the design methodology planning. The choices made about these features will qualify the entire work, determining the context which the methodology is focused on.
Applications can be divided into two categories according to their domain. More in detail, it is possible to distinguish between data-oriented and control-oriented applications. The prior includes those applications performing computation on a massive amount of data, as dataflow intensive specifications

and Digital Signal Processing. It is the case, for example, of multimedia applications, since they work on a stream of data. The latter includes applications designed for the control. For example, protocol handlers fall in this category, since they implement the set of formal rules that must be observed when two or more entities communicate. Applications falling in one rather than the other category have different peculiarities, often making HLS design methodologies developed for one unsuitable for the other. Hence, one of the most relevant challenge in developing an HLS methodology can be to make it adequate for both the application domains.

Another variable to define in designing an HLS methodology is the input language adopted to describe the specification. From this point of view it is possible to distinguish between two macro-categories: low abstraction level descriptions and high abstraction level descriptions. The first category includes languages such as Hardware Description Languages (HDL). Despite HDLs are not the languages at the lowest abstraction level, they are considered, in this classification, as low abstraction level languages with respect of those included in the second category. There exist several kind of HDLs, from the primitive ISP and KARL, developed both around 1977, to the more common Verilog and VHDL. Such languages can precisely describe operations and circuit's organization. The category of high abstraction level descriptions can be in turn divided into the one of high-level programming languages, such as C, C++ or SystemC, and the one of graphical models, such as extended Finite State Machines (FSMs) or Petri Nets (PNs). The choice of the description language is often tightly related to the application domain. For example, describing a control-oriented applications as an extended FSMs may result simpler than specifying it by means of an high-level C-like language, often leading to more performant synthesis results.

Finally, the target architecture must be defined. As anticipated in Chapter 1, the architectural model is usually composed by datapath and controller. The most common approaches adopt an FSM as controller. Among these approaches, several solutions can be distinguished. For example, the archi-

tecture can be composed by a single centralized FSM, rather than of a set of distributed and/or parallel FSMs. Moreover, a hierarchical structure of FSMs can be built. However, there exist other solutions, based on the application behavior. Such approaches construct the circuitry starting from a behabioral description of the program, such as the Behavioral Network Graph (BNG).

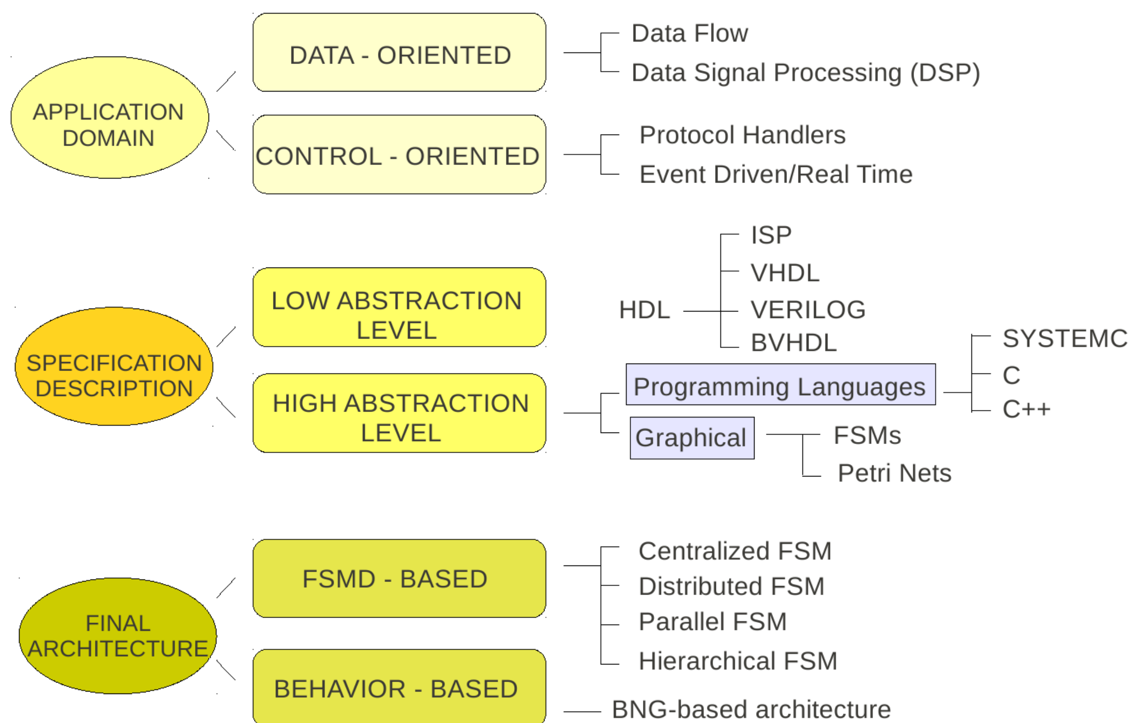

Figure 2.1: *Main Features Characterizing Design Methodologies in HLS.*

In Figure 2.1, for each of these three features (represented in the ellipsis) the corresponding categories are shown (represented in the rectangles), as just described. Moreover, some examples for each category is reported.
The three identified variables are strictly related each other. A good design methodology should find the right combination between the choice of the specification description and the definition of the final architecture, while

obtaining good performances for both the application domains. In the following sections HLS chronological evolution will be tracked, focusing on the choice made about these three features. Particular attention will be paid on the final architecture definition, since this aspect represents the central topic this thesis is focused on.

2.2 HLS Prehistoric Period

The seventies provided the basic ideas on which HLS is based, hence we refer to this years as prehistoric period. More in detail, in 1974 M. Barbacci noted that, theoretically, one could "compile"a behavioral description of the specification into hardware, without any information about its structural description, such as synthesizable Verilog, thus setting up the notion of design synthesis from a high-level language specification [32]. One of the pioneering research groups worked at Carnegie Mellon University (Pittsburgh, Pennsylvania) using input languages such as Instruction Set Processor (ISP), Instruction Set Processor Language (ISPL), and Instruction Set Processor Specification (ISPS). Such languages were HLDs developed ad hoc mainly to describe DSPs applications. As above mentioned, the target hardware circuit consisted of a structural composition of data path, control and memory elements.
Even though the prehistoric works showed interesting and highly groundbreaking traits for the research, it was not yet the time for HLS spread in industrial design, on which this new approach had a very little impact in that years.

2.3 First HLS Generation

The period from 1980s to early 1990s characterized the first generation, that have seen a decomposition of the HLS tasks into several subtasks, such as hardware modeling and controller generation, on which this thesis work is fo-

cused, rather than scheduling, resource allocation and binding. Even if many of these subtasks are NP-hard problems, and they are often strictly dependent on each other, they are almost independent from a logical point of view, representing different problems that can be faced with different techniques. Such decomposition, nowadays still adopted, had the aim of simplifying the whole problem solving, focusing on one problem at a time, according to a *divide et impera* paradigm. Moreover, in this way was possible to obtain more performant circuits as result of the synthesis process, facing each subtask with a different and appropriate methodology.
This age started with primitive and poorly performant algorithms in almost all the subtasks, that for this reason were usually addressed as single-objective optimization problems. For instance, the school of thought supported by P. Paulin, J. Knight and their colleagues proposed to fix a latency constraint trying to optimize area occupation [67][66]. The advantage of fixed latency was an easy embedding of the resulting designs into systems with more relaxed timing constraints. Only at the end of this generation the improvements made in the underlying algorithms allowed simultaneous consideration of timing and resource constraints. These improvements lead to a successful appliance in the design of filters and other Digital Signal Processing (DSP) functions.
In [37] are clearly identified two relevant aspects of first-generation HLS research: first, the usage of a domain-specific input language, generally oriented to describe DSP algorithms, and second the attempt to commercialize the research ultimately failed. There was a long time of commercialization attempts, in which the technology changed input languages, application scope, and user interfaces many times, making HLS methodologies no more behind the times. Such technological changes led companies to stop HLS tools promotion on the market, although their internal use went on. Thus, it can be inferred that the fundamental aspect restraining first HLS generation spread was a limited acceptance by final users, i.e. the designers, that found most drawbacks than advantages in using HLS. More in detail, such limitations

were concerning:

- *neither necessary nor useful*: concurrent changes in design technologies for integrated circuits, due to recent adoption of RTL synthesis, were revolutionizing design methodologies. In such a scenario, automatic placement and routing technologies offered by RTL synthesis seemed more desirable, and the idea that HLS could fill a design productivity need was an unlikely one.

- *input languages*: the type of input languages adopted in this first generation presented great difficulties, since they consisted in domain-specific HLDs developed ad hoc. Adopting new input languages for a new and unfamiliar design approach was an obstacle for many.

- *quality of results*: the resulting design was often inadequate, due to primitive scheduling, simple underlying architectures and expansive allocation criteria.

- *domain specialization*: methodologies and techniques implementing the early tools were focused and properly worked on DSP design, concentrating on dataflow and signal processing. They were not appropriate for the vast majority of early Application-Specific Integrated Circuit (ASIC) designs, which concentrated on control logic.

In conclusion, it is possible to say that early works in HLS were mainly focused on scheduling heuristics for dataflow-dominated specifications as shown in [65], [28], [16] and [26], with first attempts to automate the synthesis of data paths, as described in many works, the most important of which are [83], [50], [36] and [64]. As well explained in [54], it was the era in which the research mainly concentrated on datapath-domain-specific applications.

2.3.1 Architectural Models

In this era the dominant model adopted for the final architecture consisted in a composition of datapath and controller. More in detail, the classical

approach in designing a controller consisted in synthesizing it as a *centralized Finite State Machine* (FSM). Many works were published about FSMs, the main of them are [23], [21], [80] and [81]. Such approach is still widespread.

Centralized Deterministic FSM

The centralized deterministic FSM approach is the simplest from the architectural model point of view. The controller is modeled by a single FSM. More in detail, as shown in Figure 2.2, the controller structure is composed by a combinational block implementing the state transition function and the output function. The states are generally represented through *edge-triggered* registers, characterized by responding to the change of state on the variation front instead of on the level of voltage. An important advantage of this structure is a simple clocking scheme, with a single clock which is not gated, so that correct function and timing can be easily verified. The maximum speed of such a structure is determined by the time required for changing the flip-flops output and by the maximum propagation delay through the combinational block. Since this delay directly corresponds to the size of combinational block, decentralized approaches seems more desirable. Moreover, since the FSM is intrinsically sequential, parallelism is allowed only for those operations that are assigned to the same control step. In other words, the parallelism extraction task is totally moved to scheduling and binding phases, often leading to an explosion in the number of states [27].

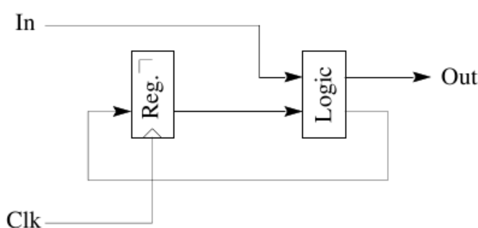

Figure 2.2: *Standard Structure of a Centralized FSM.*

2.4 Second HLS Generation

The period from mid 1990s to 2000s characterized the second-generation HLS evolution. This age finally led the major EDA companies, such as Synopsys, Cadence, and Mentor Graphics, to commercialize their HLS tools. Synopsys offered Behavioral Compiler [49], Cadence's Alta group provided Visual Architect, and Mentor Graphics proposed Monet tool. Although these tools were tried out seriously by a number of users, the technology achieved a failure again.
The main failure reasons can be summarized as follows:

- *input languages*: behavioral Hardware Description Languages (HDLs) were used as inputs. This choice was determined by the mistaken assumption about who would use HLS, i.e. RTL synthesis users, that were, by that time, almost familiar with such languages. This led to a failure on several fronts. First, current RTL synthesis users decided to keep their own tools, instead of switching to new tools that neither provided substantial improvements in quality of results at the same effort, nor gave substantial reductions in effort with the same quality of results. Second, algorithm and software designers were discouraged by the need, that HLS adoption would mean, of learning HDL. Third, using HDL as input language implied simulation times as long as with RTL synthesis, hence HLS adoption did not gave any advantage in terms of time. Finally, it resulted impossible to exploit compiler-based language optimizations.

- *quality of results*: the resulting design was still inadequate, as well as unpredictable and widely variable. Moreover, formal validation methods were not been proposed yet. Hence, understanding if the synthesis results were correct resulted very hard.

- *lacks in control-dominated specifications synthesis*: designers often used specialized datapath compilers in conjunction with RTL synthesis, as

shown for example in [30]. However, HLS could be applied also to control-dominated algorithms. First attempts of applying this extension were proposed in this period, but with worse results with respect to dataflow-dominated algorithms synthesis. Therefore, HLS started to be considered a partial solution, giving quite good results only in certain conditions. At that time, many researchers thought that understanding the reasons of poor results in control synthesis was not a right investment of their time. Among the causes for this lacks there were the use of inappropriate or insufficient Intermediate Representations (IRs), that did not include informations about the control. Indeed, the most common IR adopted in that period was the DGFG, as shown in [62] and [61].

The second generation was the first age of commercial EDA and behavioral-synthesis tools driven by hardware description languages. From the research point of view, many works of that period, as [74], [69] or [82], were focused on new scheduling techniques and strategies. Moreover, about the application domain, some publications of that period, as [20], were still focused on datapath synthesis. However, there was some attempt in control-dependent specifications synthesis, as in [72], [52] and [85].

2.4.1 Architectural Models

In early second generation, first attempts to modify the target architecture were proposed. Indeed, early second generation researchers realized that using a centralized FSM, synthesizing the controller, meant a great waste in terms of area. As a consequence, the whole circuit latency increased, thus giving worse performances. A first approach to solve such problem consisted in FSM decompositions, leading to the definition of a *distributed controllers*, possibly organized in a *hierarchical structure*. Such decompositions aimed to reduce the area and/or the delay within an FSM. Many works were published about this technique, such as [78] and [63]. In mid nineties, a successive approach followed, based on the idea that the primary responsibles for growing

area were the interconnections. Indeed, interconnections comported the critical path delay to increase, determining a longer circuit clock cycle. The critical path delay in a RTL circuit with a datapath and a controller is the register to register data flow path with the longest delay. It consists generally of three components: controller delay, control wire delay and datapath delay. In this period several techniques were proposed to reduce the critical path delay, at different design levels. For instance, careful module generation techniques at RTL level could produce faster modules, improving the performance in the critical path. Moreover, logic minimization methods could be used at the logic level reducing the number of gate levels in the critical path. Unfortunately, these techniques met the controller and datapath delay reduction work, while giving no benefits in wiring delay reduction. Furthermore, especially at high frequencies, the interconnect wiring delay results the dominant factor in the circuit delay. On the other hand, as described in [39], the control path delay has been found to be the slowest segment of the overall critical path delays. For all these reasons, some mid-90s approach, such as [39] and [18], performed FSMs decomposition with the aim to reduce control path and control wire delays.
Finally, FSMs decomposition techniques were proposed to address another problem, i.e. power overhead. In the following the main techniques proposed in these years about FSM decomposition will be deeply explained, after a brief overview on formal FSMs decompositions methodologies. After that, some considerations about distributed and hierarchical controllers will be highlighted.

Formal FSM Decompositions Methodologies

Since sixties FSM decomposition problem was treated from a formal and theoretical point of view. Three main decomposition techniques were identified: *parallel decomposition*, *cascade decomposition* and *generalized decomposition*. *Parallel decomposition* is the simplest technique. As shown in Figure 2.3, the submachines M_1 and M_2 are supplied with the same input sequence I. Such

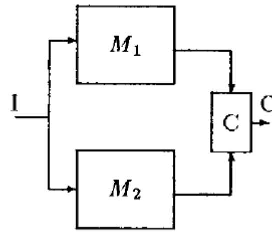

Figure 2.3: *Parallel Decomposition of a Finite State Machine.*

submachines operate independently, providing informations about their internal state to a combinatorial circuit C, whose job is to generate the output sequence O.

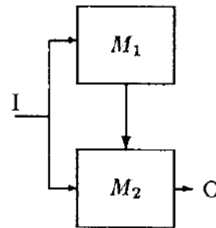

Figure 2.4: *Cascade Decomposition of a Finite State Machine.*

Cascade decomposition divides a given finite state machine into an independent and dependent component. In Figure 2.4 an FSM cascade decomposition is shown. Observe that the given FSM is broken up into two submachines M_1 and M_2, each driven by the same input sequence I. The obtained submachines do not operate independently. Indeed, M_2 is supplied, by means of auxiliary inputs (see the edge from M_1 to M_2 in Figure 2.4), with information about the current internal state of M_1. Such information influences the state transitions of M_2, and enable M_2 to generate the appropriate output sequence O. The possibility of passing state information from M_1 to M_2 makes cascade decomposition more powerful than parallel decomposition. Then the prior can be viewed as a generalization of the latter.

Generalized decomposition produces a model in which each submachine is provided with information about the current state of the other, as shown in Figure 2.5. In this case the internal behavior of each machine depends

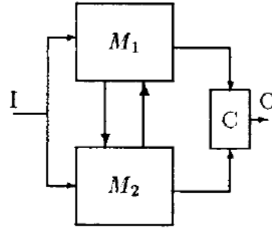

Figure 2.5: *Generalized Decomposition of a Finite State Machine.*

both by the behavior of the other and by the input sequence I. Parallel and cascade decompositions can be viewed as particular cases of the generalized one.
Among the works treating FSMs decomposition in a formal way there is the one proposed by Hartmanis [40] in 1960, who applied an algebra on partitions of states. This work focused on *cascade decomposition*. Such methodology was extended in the few subsequent years in [53], [35] and [41] to preserve the covers for the cascade decomposition finded. About parallel decomposition we cite [29]. Finally, we highlight that generalized decomposition [78] received much less attention.

FSM Decomposition reducing area and delay within an FSM

From late eighties, researchers started to apply decomposition techniques, formalized years before from a theoretically point of view, in FSMs design for logic implementation, as reported in [78] and [63]. They found soon that cascade decomposition has limited use in FSMs design, since specifications of centralized controllers in microprocessor chips do not usually have good cascade decompositions. Obviously, also parallel decomposition resulted inadequate, being less general than cascade one. Hence, they started to search for factors producing a good generalized decomposition. In [78], for instance, such factors were identified in sets of states and transition edges obtained from a State Transition Table [79] specification of the given FSM. These factors were extracted and represented as a factoring submachine. Then the occurrences of these factors in the original machine were replaced by calls to

the factoring submachine. As discussed before, the objective was reducing the overall area and delay within an FSM. For this reason were defined procedures to find all the *exact factorizations*, i.e. those that maximally reduce the number of states and transition edges in the original machine. An important result of this research is expressed by the following theorem, that shows the significant advantages of exact factorization in terms of area, due to the great reduction in the total number of edges and states that such technique involves.

Theorem 1. *A decomposed submachine M_i, produced by factorization from an original machine M, via an exact factor with $N_I(i)$ internal states and $N_E(i)$ exit states in each occurrence $O_F^i \in M$, will have*

$$\sum_{i=1}^{N_R}(|e(i)| - N_I(i))$$

edges less than the original machine M, where $e(i)$ is the set of internal edges in $O_F^i \in M$, and N_R is the number of possible factors for M.

Unfortunately, exact factorization often produced too small submachines, resulting in useless decompositions. Moreover, exact factorization may not exists at all for a given machine. Hence, techniques to find good, though inexact, factors in an FSM were proposed.

FSM Decomposition reducing control path and control wire delays

Another group of researchers concentrated on the critical path reduction, as [18] and [39]. More in detail, techniques to reduce control path and control wire components of the overall delay were proposed. Researchers focused on such components since the prior was found to be the slowest segment of the critical path, and the latter was found to be the dominant factor, especially at high frequencies.

Eppling in [39] identified *control points* in a machine, representing minimal partitions of a centralized controller, aiming to divide it into multiple local controllers. Each control point can be viewed as the controller managing one

operation in the machine. Hence, if an FSM contains N operations, then N control points can be individuated inside that machine, one for each instruction. Eppling used a *wire length extraction* technique followed by *clustering* of control points into local groups. Clustering was targeted at minimizing wire lengths.

Papachristou and Alzazeri extended such work in [18]. They firstly partitioned an RTL based controller output into control points, and then partitioned the datapath around its constituent. At this point the control points individuated were grouped so that all control points enabling the same kind of operation were placed in the same datapath partition. Hence, in this case, clustering resulted from datapath partitioning. In this way multiple local controllers were generated, each controlling one datapath partition, also said functional block. The last step of such technique was in the layout phase, in which each controller were placed physically close to its corresponding functional block, shortening wire lengths and thus reducing delays, especially at high frequencies.

As will be described in the next chapter, we will introduce an extension of the concept of control point, considering not only the operations in the program, but also the needed and available resources in the target architecture. More in detail, a new control component managing one occurrence of a resource in the machine, with all its associated instructions, will be introduced. Hence, if an FSM contains for example three additions and four multiplications, and the target architecture has at least three adders and four multipliers, then seven new components can be individuated inside that machine, one for each resource needed and available.

Observe that such approach starts from an RTL circuit, and thus it is not included in the HLS flow. Better results can be obtained integrating this work in the RTL generation process. Moreover, datapath partitioning and close physical placement of local controllers can be used to infer useful hints for resource binding.

Distributed Controller

As above mentioned, a distributed controller structure can be obtained by FSM decomposition. From what said so far, it is possible to infer that the objective of decomposition was to reduce the delay inside an FSM, or to reduce the critical path delay, rather than to reduce the power overhead. The concept of parallelism extraction is not included in the aims of realizing a pure distributed controller structure. As in the case of centralized FSM, the task of individuating set of instructions that can be simultaneously executed does not concern the controller. Consider, for example, the above described technique proposed by Eppling in [39], or the one presented by Papachristou et al. in [18]. From the analysis of these methodologies it is possible to infer that pure decomposition can directly be applied on an RTL circuit, without being included in a HLS flow. In other words, decomposition takes as input a centralized FSM, that can be obtained through HLS, and manipulate it to reduce area and delay. Scheduling and binding have been already performed on the centralized FSM and do not change after decomposition. Hence the set of possible states for the distributed machine is the same obtained for the centralized one. However, in this case the state of the entire machine has not to be explicitly represented, since can be obtained as a composition of the submachines states, leading to a considerable reduction in the number of total states.

Hierarchical Controller

Inside Hierarchical Controllers different subcontrollers are organized in a hierarchical structure, usually in accordance with the hierarchical relation obtained from the *Hierarchical Task Graph* (HTG) [57] of the specification. In the multi-level hierarchy, a controller at one level distributes groups of operations among its direct descendant. Each controller can start its computation after the activation signal from its father has been received. When a local computation is terminated, the corresponding subcontroller send a signal to its father, implementing a synchronization mechanism that enable controllers

at higher levels to properly activate subcontrollers.
A hierarchical controller structure is completely compatible with distributed one. Moreover, it is also compatible with parallel controllers, that will be presented in the next. Hence, it is possible to implement distribute and hierarchical controllers, or distribute, parallel and hierarchical controllers.

2.4.2 Improving Control-Oriented Specifications Synthesis

As above mentioned, this era was characterized by the first attempts of addressing the control-oriented specifications synthesis. Applications belonging to such domain have very different features with respect to their dataflow-dominated counterpart, thus needing different approaches. The most relevant proposals in such direction were the ones based on non-deterministic finite state machines, that will be now described.

NFA-Regular Expressions Based Controller

Usually, machines such as protocol handlers or communication encoders, results too complex to be described through a deterministic FSM model. They need instead *Nondeterministic Finite Automata* (NFA) to be concisely described. An NFA, or *Nondeterministic Finite State Machine* (NFSM) [58], is a finite state machine in which for each pair $\{$*state, input*$\}$ there may be several next states. It was formally proved that for any given NFA it is possible to construct an equivalent deterministic FSM through standard methods, such as *powerset construction* or *subset construction* (see [79], Theorem 1.19, section 1.2, and [45]). Obviously, the first step needed to synthesize an application, described with an NFA-based model, is transforming such description in its deterministic equivalent. For this reason, NFA become explosive and untenable when implemented in hardware with standard approaches, that simply transform it in a deterministic FSM. For example, one of the most widespread approaches were based the use of *Esterel* [24], a synchronous reac-

tive language allowing an inherently non-deterministic machine description. Its commands reacted to inputs from the outside world, by performing tasks and sending outputs. Each reaction to a specified input was allowed to occur independently of other reactions, creating an NFA model. However, the Esterel compiler, as above mentioned, created a deterministic *State Transition Graph* (STG) [80], from the NFA specification, that often resulted explosive. As a consequence, various techniques were proposed to address this problem, based on the concept of classical *regular expressions* (REs) [10]. It is well known that any FSM can be specified as a regular expression, representing the set of all the strings belonging to the *formal language* recognized by the automaton. Even though this means that the classical regular expression description is always allowed, such specification is not guaranteed to be as concise as other types. This is the main reason why regular expressions alone are not enough, and should be used together with, not instead of, a nondeterministic description.

Moved by such considerations, in [75] and [11], regular expressions were used as a specification for Programmable Logic Array (PLA) designs, to be converted into an NFA state transition diagram, which in turn was directly encoded as product terms of a PLA implementation. However, such technique may lose some of the informations present in the regular expression. For example, let us consider a regular expression $e = (a|b)^+(b|c)^*$ over the alphabet $\Sigma = \{a, b, c\}$. Such expression can be partitioned in two components: $(a|b)^+$ and $(b|c)^*$. Such kind of decomposition, defined *natural partitioning* in [8], can be useful to identify points in which the machine can be divided in submachines, distributing the control for a possible FSM factorization. When the correspondent NFA state transition diagram is obtained from e, such information about natural partitions may go lost. Indeed, both the partitions contain the character b. Hence, when b is read, if we consider only the NFA representation, there is no way to understand what natural partition this character belongs to.

A subsequent approach, proposed in [9], was the *Production Based Speci-*

fication one. They considered, as specification language, the productions derived from the *grammar* corresponding to the formal language recognized by the automaton. Indeed, as described in [60], there exist three equivalent models for the description of languages: automata, regular expressions and grammars. Similarly, from the finite state machines point of view, there exist three equivalent models for a (deterministic or nondeterministic) FSM description: the language it recognizes, the regular expression describing such language and its associated grammar. The production based specification model provided a hierarchical regular expression language augmented with some unique operators. An algorithm for direct construction of the circuit from a regular expression based tree was presented, which did not require conversion of the RE to a NFA state transition diagram. This direct construction often produced fast circuits, but with redundant state bit encodings.
Finally, Crews et al. [8] discussed techniques for high-performance controller synthesis, from the complexity point of view. More in detail, they proposed sequential optimization techniques whose complexity scales with the number of states bits, rather than the number of states. This work aimed to provide viable synthesis techniques for designs which are too large for synthesis with conventional methods. The methodology proposed in [8] started from classical *regular expressions* [10] specifications, deriving from it an NFA description. Once obtained, the NFA model were encoded as a *tree-based extended regular expression*, and explored with the aim to construct efficient controllers. More in detail, they assumed a controller specification as a regular expression in the form of a *Directed Acyclic Graph* (DAG) [48]. In Table 2.1 is shown the meaning of various symbols used in the specification DAG. Using DAG representation it is possible to specify any completely deterministic automata without making use of traditional deterministic models, such as STGs or actual state encoding. Moreover, from this specification, there are direct gate level implementations which scale with the number of state bits in the controller, which can be logarithmically smaller than the number of machine states. Once obtained the DAG representation, they performed

symbol	meaning	nodes type
,	concatenation of events (left then right)	sequential non-leaf nodes
$\|\|$	OR (either event below)	sequential non-leaf nodes
&&	AND (events occur simultaneously)	sequential non-leaf nodes
$*$	Kleene closure (0 or more)	sequential non-leaf nodes
$+$	1 ore more	sequential non-leaf nodes
action	designates an output activation	sequential non-leaf nodes
function	boolean function (of inputs only)	combinational (terminal) nodes

Table 2.1: *Regular Expression DAG Symbols.*

natural partitioning, identifying proper sub-DAGs. For example, considering the manipulation rules they adopted to minimize the original ER, specified as follows

$$(A, B)||(A, C) \rightarrow (A), (B||C) \quad \text{(Rule 1)}$$
$$(A, C)||(B, C) \rightarrow (A||B), (C) \quad \text{(Rule 2)}$$
$$A||A \rightarrow A \quad \text{(Rule 3)}$$
$$A, (A)^* \rightarrow (A)^+ \quad \text{(Rule 4)}$$
$$(A^*)^* \rightarrow (A)^* \quad \text{(Rule 5)}$$
$$A, (A\{action\}) \rightarrow (A, A)\{action\} \quad \text{(Rule 6)}$$

it is possible to identify a sub-DAG inside each open/close round-parentheses pair. Minimization is the main optimization technique proposed in this work. It aimed to remove unobservable states from the system. After that, each unique action in the DAG was put into correspondence with a unique output of the controller. The output was set high only if the sub-DAG below the action accepted, i.e. when the sequence of inputs for that sub-machine matched the entire sequence specified in the DAG. After reducing the num-

ber of terminals in the tree, the circuit was synthesized by traversing the resultant DAG. The construction required one register for each path to a terminal node. The circuit was generated recursively, by allocating registers at the terminals and constructing logic functions of the register outputs (present state bits) according to the type of sequential operator at each node. Logic functions were stored as Binary Decision Diagrams (BDDs) [12] during construction.
Despite such methodologies presented some advantages in controller synthesis for complex applications, regular expressions and nondeterministic automata based approaches resulted in general computationally more expansive than other approaches, thus not being considered general approaches.

2.5 Third HLS Generation

In early 2000s started the third HLS generation, that represents the current one. The most important improvement characterizing this era is the adoption, as HLS inputs, of C-like specification languages, as documented in many works, as [84]. This made possible to take advantage of research into compiler-based optimizations for code parallelization, regardless of HDL-driven improvements. Designers had no more to learn unfamiliar HLDs, using instead modified versions of C, C++ or SystemC languages, and design outputs gained a significant improvement. Indeed, compiler-based transformations and optimizations allow code restructuring. For example, on a DSP processor, the application's control flow can be rearranged so that intermediate data always fits in cache, or, on a Field-Programmable Gate Arrays (FPGA), the initial C-like code can be restructured in such a way to extract potential parallelism, resulting in a streaming pipelined implementation. Moreover, if the original specification result inappropriate for the target hardware, restructuring can be used for example to get the design to work in real time, or to fit within the available hardware resources. Many works were published about compiler-based optimizations, as for instance [33], [34],

and [68].
This is surely the generation in which tools commercialization increased. Many EDA companies offered their own synthesizers. Some of these tools primarily target Application-Specific Integrated Circuits (ASICs) and Application-Specific Standard Products (ASSPs) designs, while others primarily target FPGAs designs, and still others target both. The rise of FPGAs, due to their increased capacities and to the incorporation of specialized features such as multipliers and distributed memories, played an important role in HLS commercialization success. Since applications and architectures complexity were significantly growing, implementing in hardware code portions as accelerators became common use. FPGAs were used for such purpose, and using HLS with FPGA targets was found to be a perfect way to quickly map an algorithm into hardware. Most of the tools offered are based on methodologies capable of handling dataflow and DSP domain, a small amount is focused on control, as for instance Esterel EDA Technologies Esterel Studio, and a few sustain to be suitable for both dataflow and control, as Bluespec Compiler (BSC) and Cadence C-to-Silicon (C2S).
Another significant improvement of this generation is related to the verification phase. Some high-level synthesis tools generate SystemC representations of synthesized designs, enabling much faster simulation runs compared with traditional RTL simulations. However, low-level simulations may still be needed for the verification, especially in the case of integrated system, since HLS ones may not simulate all of the interfaces of the synthesized hardware, as I/O, memory subsistems or other interfaces. Indeed, for example, all FIFO buffers are modeled as infinite-depth C or C++ arrays, whereas in the final implementation they will always have a finite depth. Furthermore, temporal aspects of the design, such as FIFO push and pop, cannot be modeled in untimed languages such as C or C++. To facilitate SystemC and RTL simulations, HLS tools typically automatically generate RTL test benches based on high-level test benches provided by the user, in addition to the RTL module implementing the design, as shown in Figure 2.6.

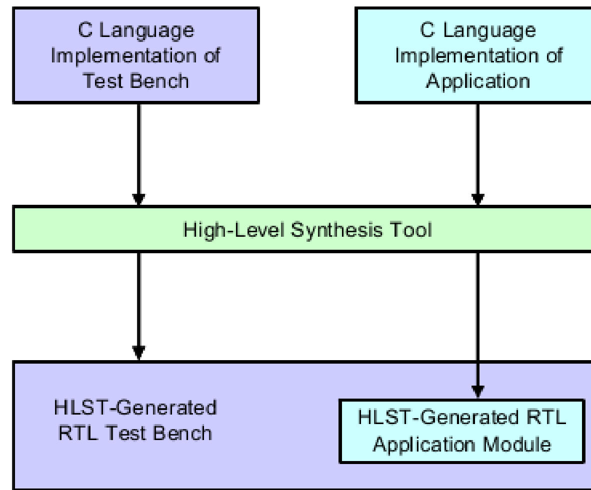

Figure 2.6: *Tipical HLS tools output including RTL and RTL test benches.*

Finally, high-level synthesis represents an efficient means for design space exploration. Indeed, HLS tools can provide an estimate of the resource utilization and clock frequency that the synthesized implementation is likely to meet. Obtaining this estimate does not require invoking the RTL tools and can be obtained at any point in the design process, enabling early design exploration, for example to identify achievable cost-performance points. Hence, shifting the estimation phase at an higher level produces significant benefits in terms of time and costs for design space exploration. For this reason, as documented in [77], some of the most widespread uses of HLS tools in industry today are for architecture exploration and rapid prototyping. In architecture exploration, hardware and system architects can use HLS tools to quickly model the functionalities of different architectures and thereby derive an upper bound on their respective silicon area, performance, and power. This allows to make a more accurate choice between the various alternatives. In rapid prototyping, a system can be quickly modeled in an FPGA, enabling system simulation, system-level performance analysis, and so on. Moreover, area and timing estimates can be used to assess the synthesis results and, as necessary, to make improvements to the implementation by modifying

the high-level representation. Nevertheless, in some cases estimates result misleading and inaccurate, over/under estimating up to 25%.

2.5.1 Architectural Models

In third HLS generation a new kind of final architecture was proposed, i.e. *parallel controllers*. The adoption of high-level input languages describing the specification led to difficulties in exploiting the parallelism, due to the sequential nature of such languages. Hence, parallel controllers were proposed to address such problem.

Parallel Controller

Historically, in literature the two terms "distributed" and "parallel" have been often indifferently attributed to a controller structure. In our opinion, despite many similarities, there exists a subtle difference between them. Both the terms indicate the presence of an underlying structure composed by subcontrollers that work simultaneously, possibly interacting each other to compute their next state. In that sense a distributed controller can be viewed as parallel too, since subcontrollers work simultaneously. On the other side a parallel controller results distributed too, since it is organized in a local controllers structure. However, we are interesting in another meaning of the term "parallel", indicating if the controller is able to auto-detect the parallelism in the specification, while meeting finite resource constraints. Observe that at compile time only an estimation of the parallelism can be performed, since many informations, such as branch conditions results are unknown. Hence, if scheduling and binding are statically performed, part of the exploitable parallelism may be lost. In this sense, a pure distributed controller, obtained by decomposition, cannot be parallel too, since it ignores the parallelism extraction problem, already addressed in the previous phases of the synthesis, while a pure parallel controller is for sure distributed too. In other words, a parallel controller is responsible for the scheduling and the binding tasks. There exist also hybrid solutions in which, for example, the controller

works as scheduler, while the binding is previously statically performed. Obviously, implementing a parallel controller structure requires a non trivial communication protocol handling subcontroller-to-subcontroller communication, subcontroller-to-resource communication and resource-to-subcontroller communication.
Parallel Controllers have been often implemented adopting a specification description capable to express the parallelism in the application. As will be explained in the next, Petri Nets [17] [46] have been found to be a proper model supporting the implementation of efficient parallel controllers. The token-based architecture obtained through a Petri nets model faithfully mirror the one needed to represent communication in a parallel controller.
However, parallel controller can be also obtained when a C-like description of the specification is used, with the help of a proper IR, as in the case for example of the BNG. Such IR, as will be clarified in the next, has the peculiarity to represent also the final parallel architecture.
Since parallel controllers construction techniques depend on the description adopted for the specification, in the following such aspects will be deepened.

2.5.2 Specification Description

As above mentioned, C-like high-level programming languages found great acceptance in third HLS generation. However, they are affected by the problem of parallelism extraction. As a consequence, adopting such languages, building a proper IR to support the synthesis become a critical problem. Historically the standard IR adopted in conjunction of C-like specification languages is the CDFG. A relevant alternative approach is based on the Behavioral Network Graph, that is an intermediate representation able to describe also the final architecture, thus unifying high-level synthesis and logic synthesis domains. However, both such approaches are afflicted by the parallelism extraction problem. The alternative could be using graphical description languages for describing the specification, such as Petri Nets.
In the following such approaches will be explained.

Control-Data Flow Graph Based Approaches

Control-Data Flow Graph (CDFG) based approaches are the most common in HLS representing a standard de-facto. Most of the commercial tools are based on such IR, and many reasearch works are found on it too. The features of CDFG that make it useful as an initial specification from which to synthesize are the ability to perform static scheduling and the analysis of resource needs that is possible.
Hybrid techniques were proposed based on DFG or on CFG. Fon instance, Jung et al. in [38], considered a DFG specification in which each node represented a hardware library module containing a synthesizable VHDL code. From such specification they automatically synthesized a control structure, called *cascaded counter controller*, supporting asynchronous interaction with outside modules while implementing the synchronous dataflow semantics of the graph at the same time.
DFG, CFG and CDFG models, however, may lack in exploiting part of the available parallelism in the specification, and may underestimate the amount of needed resources. For an example in which CDFG fails in recognizing the available parallelism see the specification reported in Figure 3.2 (Chapter 3, Section 3.2). The underestimation of the amount of needed resource is a consequence of sequentializations forced by false dependencies.

Behavioral Network Graph Based Approaches

Behavioral Network Graph (BNG) representation was proposed by Bergamaschi in [73] with the aim of unifying the domains of HLS and Logic Synthesis. The BNG is an RTL/gate-level representation of a behavioral specification. The construction methodology to build a BNG starts from the CFG to create a logic network representing the FSMs for all possible scheduling. Then DFG and the results of data-flow analysis are considered to obtain a logic network representing the datapaths for all schedules. Finally, control and data portions are merged to obtain the BNG.
Since this thesis focuses on the control, in the following only control BNG

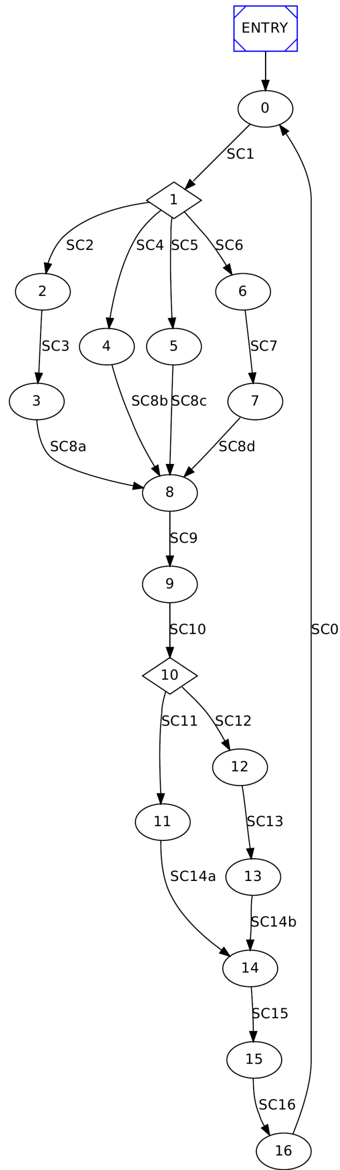

Figure 2.7: *Example of CFG for a Specification.*

construction methodology will be deepened. The first step in scheduling a CFG consists in individuating places bounding a state. Such places consists in cuts on the CFG edges, named *state cuts.* A state cut on a CFG edge indicates that all overlying and underlying operations must be scheduled in different control steps, thus belonging to different states of the machines. A state cut variable SC_{ij} is defined for each node i of the CFG and for each i's incoming edge j, with value 1 if a state cut occurs in the corresponding place, 0 otherwise. State cuts can be placed in different positions according to the available resources. Moreover, different solutions can be obtained, also considering the same number of resources, if there are less resources of a certain kind than operations needing that resource kind potentially executable in the same control step. Consider for example the CFG in Figure 2.7. If state cuts are placed for instance between the operations 2-3, 3-8 and 13-14, the FSM shown in Figure 2.8 is obtained, whose implementation using one-hot-encoding results in the logic network is shown in Figure 2.9.

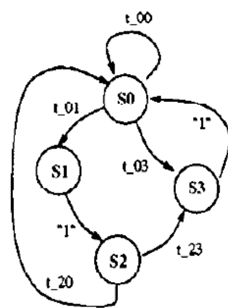

Figure 2.8: *A possible FSM for the CFG in Figure 2.7.*

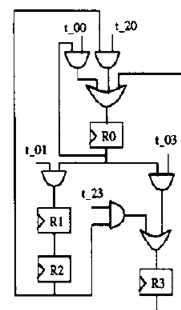

Figure 2.9: *Hardware Implementation for the FSM in Figure 2.8.*

Each state cut has direct implications on the storage elements and interconnections in the datapath. Indeed, when a state cut is placed inside the lifetime interval of a value, it forces that value to be stored in a register since its definition is in one state and its use in another. In other words there exists a direct correspondence between state-cuts and registers in the one-hot-encoded FSM. Hence, the positions of the state cuts determine the basic

control and datapath logic, thus, for the BNG to represent all schedules, it needs to encompass the different hardware configurations for different choices of state-cuts. For this reason *State-Value Nodes* (STNs) are introduced. A STN is a logic structure which represents the choice of either having or not having a state cut on a particular control-flow edge. In other words an STN is a switch which can choose between storing the input value (if the associated state cut variable has value 1, thus a state cut occurs) or passing it through the output immediately (if the associated state cut variable has value 0, thus a state cut does not occur). From a more practical point of view, when $SC_{ij} = 0$ the STN simplifies to a wire, thus not enforcing a new state. When on the contrary $SC_{ij} = 1$, the STN simplifies to a register, thus enforcing a state transition.
The algorithm for the control BNG construction can be summarized in the following steps:

1. Traverse the CFG and, for each node i, associate a variable SC_{ij} with each i's incoming edge j. If the node i has a unique incoming edge, then the subscript j can be omitted. A node with an indegree greater than one is called *join node*.

2. Traverse the CFG and for each node i with a single predecessor and a single successor, create a STN_i. The net at the output of the STN_i gate (STN_i net) represents the control signal activating the operation in node i.

3. For each join node i, create a STN_{ij} for each predecessor j and connect all STN_{ij} nets to a single OR gate. The output of the OR gate is called STN_i net.

4. For each node i with multiple successor edges (*fork nodes*), create a STN_i and connect its output net to as many AND gates as successor edges. Each AND gate has two inputs: the first input is net for the fork node, STN_i, and the other input is a net representing the condition on the corresponding successor edge. This condition net may be a primary

input or a net coming from the datapath. The output of each AND gate is called net STN_{ij}.

5. Connect the multiple STN_i obtained in the same topology as the CFG.

Despite such techniques has the enormous advantage of providing the controller of scheduling and binding capabilities, it has the great drawback of starting from the CFG. As above mentioned, the CFG contains false dependencies avoiding to completely exploit the available parallelism in the specification.

Petri Nets Based Approaches

A *Petri Net* [43] (PN) is a mathematical modeling language for the description of distributed systems. Thanks to its ability in representing the parallelism inside an application, this model started to be used as graphical specification description language in HLS, as an alternative to high-level programming languages. A Petri net can be viewed as a marked version of a *Petri Net Graph* [43].

Definition 1. *A Petri Net Graph (PNG) is a 3-tuple (S, T, W), where:*

- *S is a finite set of places, i.e. nodes representing conditions*

- *T is a finite set of transitions, i.e. nodes representing events that may occur*

- $W : (S \times T) \bigcup (T \times S) \rightarrow N$ *is a multiset of arcs, i.e. it defines arcs and assigns to each arc a non-negative integer arc multiplicity.*

Observe that no arc may connect two places or two transitions.

Definition 2. *A Petri Net is a 4-tuple (S, T, W, M_0), where:*

- *(S, T, W) is a Petri net graph*

- *M_0 is the initial marking, a marking of the Petri net graph*

A Petri net representation of a controller structure is equivalent to the one obtained by dividing the specification of a controller into a number of concurrent processes, producing a set of sub-controllers, and then implementing each controller as an FSM and linking them together with control lines and/or semaphore bits subject to the initial parallel specification. An example of PN specification of a controller is shown in Figure 2.10.

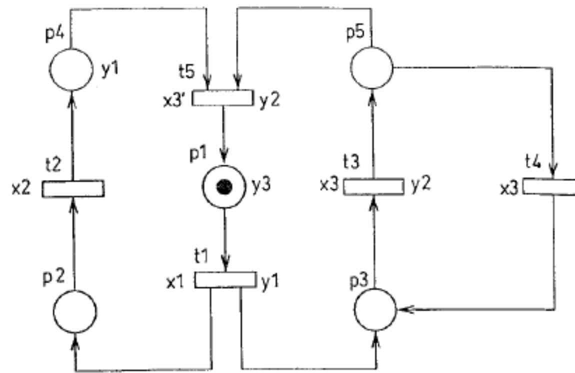

Figure 2.10: *Example of Petri Net Specification of a Controller.*

Petri nets have found relatively acceptance in HLS research. Biliński et al. [51], for example, proposed to exploit the graphical representation of concurrency provided by Petri nets to synthesize a parallel controller structure. They based this choice on the observation that such graphical representation is often easier to understand, hence it can reduce the likelihood of parallel synchronization errors. The main improvement derived from the work in [51] was that, unlike previous works, they did not need the construction of a reachability graph from the Petri net for synthesizing parallel controllers, with consequently saving in computational time and memory. Such work was later extended by the same authors in [47], explicitly implementing VHDL simulation cycle implications into the Petri net model. Here a formal controller decomposition methodology for a Petri net specification was presented. Finally a hierarchical PN-based approach was proposed in [44].

The main advantage in using PNs is that they allow easy specification of co-operating subsistems, and the use of formal validation methods. The model represents a token-based architecture, implementing the communication into the controller structure. A PN-based model, however, requires priority and synchronization schemes to share variables and to implement sub-controllers communication, increasing design costs.

2.6 Why Fourth HLS Generation is Prospected

The commercial adoption of high-level synthesis mirrors the technology's various phases, as shown in Figure 2.11. The years from 1994 to 1996 are included in the first generation. As above mentioned, at that time HLS was no more than an experiment, and this meant low sales levels. The years from 1997 to 1999 characterized the second generation, in which an increment in the sales level of the major EDA companies was recorded. After a first success, the commercialization attempt failed leading to the sales decrease that characterized the years from 2001 to 2003. From 2004, third generation emersion mirrors a growth in sales, that still continues to rise, as is prospected it will do in the future. At this point, as HLS future seems to be positive, the question that arises is why fourth HLS generation is prospected. To answer this question a deeper analysis on third HLS generation defects and lacks is needed.
Hence, in this section current HLS generation will be deeply analyzed with the aim to discover the reasons why a fourth one is prospected, and which gaps next HLS generation should fill. Moreover, will be clarified how this thesis work proposes to contribute to HLS improvement.
User experiences, documented in technical papers, can be a good means to really understand the effectiveness of the improvements claimed in third HLS generation. Unfortunately, this usually takes many years before enough meaningful reports become available, also considering restrictions, often imposed by tool's vendors, from publicly discussing comparative benchmarks.

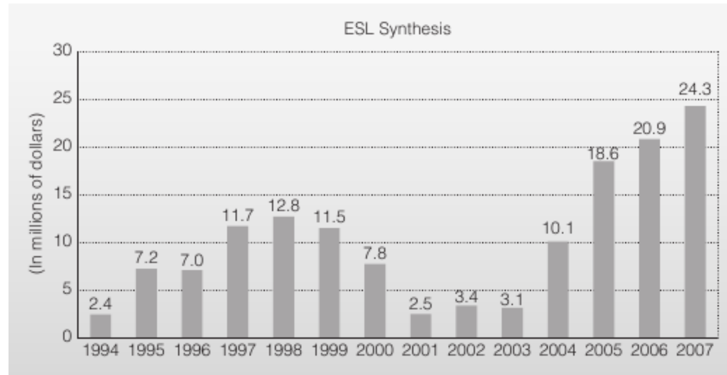

Figure 2.11: *Sales of electronic system-level synthesis tools. Source: Gary Smith EDA statistics.*

Hence, since we have seen that market trends are very close to technological changes in HLS, first of all, we can take a look at the market analysis about EDA segments revenue in the last five years, conducted by the EDA Consortium (EDAC) Market Statistics Service (MSS), shown in Table 2.2. From such analysis is possible to infer that, despite high-level design and verification is constantly growing year over year, and RTL design and verification is loosing market share, still in 2010, the latter has registered revenues for more than three times higher with respect to the prior. This means that RTL remains the dominant specification and synthesis level. To understand the reasons for this, we will make our observations on the basis of what researchers and industrial groups have published so far about their experiences in HLS usage. Among the most recent works in this direction we mainly refer to that one proposed by Sarkar et al. in [77], in which HLS usages, improvements and lacks are discussed, mainly focusing on ASICs targets, and to [15] in which an independent benchmarking and analysis firm, named Berkeley Design Technology Incorporation (BDTI), proposes a certification program to evaluate high-level synthesis tools for FPGAs. In the following, the aspects related to each of these domains will be discussed separately.

Among the most common utilizations of HLS in industry, there are design

	2006	**2007**	**2008**	**2009**	**2010**
High-level design and verification (ESL)	121.6	139.5	159.5	180	205
Year-over-year growth	16%	15%	14%	13%	14%
Percentage of total computer-aided engineering segment	5%	6%	7%	8%	9%
RTL design and verification (traditional)	651.2	696.9	634.6	625	625
Year-over-year growth	7%	7%	-9%	-2%	0%
Percentage of total computer-aided engineering segment	29%	28%	29%	28%	28%
Total computer-aided engineering segment	2213.4	2467	2199.7	2200	2250
Year-over-year growth	14%	11%	-11%	0%	2%
Total EDA market	4078.3	4464.8	3853.7	3850	3960
Year-over-year growth	14%	9%	-14%	0%	3%

Table 2.2: *EDA segments revenue analysis ($ MILLIONS). Source: EDAC MSS Statistics Report and Desaisive Technology Research analysis.*

space exploration and rapid prototyping. In ASIC target design, these utilizations are usually more widespread than using HLS with the purpose of generating RTL code to synthesize into silicon. However, these kind of applications does not have neither the stringent area or power requirements involved in designing an ASIC, nor the requirement for tight integration with other design tools involved in an ASIC design flow. Hence, although such uses of HLS tools are encouraging, they are not really mainstream uses of a synthesis tool. Despite starting the design process with RTL causes both the quantity of the source description and the time required for verification tasks to increase explosively, and despite advances in HLS algorithms, RTL remains the dominant specification and synthesis level in designing ASICs. The reasons for this can be summarized as follows:

- *difficulties in meeting design goals*: when applied in a strictly constrained context, as ASIC design, HLS shows difficulties in meeting design goals, such as optimizing area and power, and meeting throughput and latency requirements. The experience documented in [77] has shown, for example, that different HLS tools produces very different designs in terms of area. These results can be viewed in Figure 2.12, in which the synthesis produced by three different tools, T1, T2 and T3, are compared with the one obtained by hand writing the RTL code. Note also the dramatic change in area between the first model and

	Area (μm^2)	
HLS tool	**Initial**	**Final**
Manual		20,800
T1	49,000	26,100
T2	38,500	23,000
T3	42,000	25,400

Figure 2.12: *Area generated by different HLS tools on a quantization and inverse-quantization (QIQ) module.*

the model optimized for area. This large variation in results become more weighty when, as frequently happens, it is not available a manual

RTL to compare with. Moreover, this investigation demonstrated that even C-like code is not efficiently portable across tools. Indeed, was found that, when applying the same C-like model, written for one tool and properly modified for another, the resulted design is significantly different in area, and no code or tool's configuration changes would reduce it. As reported in [59], high-level language choice can be a critical factor for successful use of HLS. C/C++ offers the highest level of algorithmic exploration, but sequential execution semantics with no explicit support for parallel structures. Hence, obtaining good results for the synthesized RTL requires an advanced parallelizing compiler embedded in the synthesis tool to automatically extract parallelism in an application. The quality of results become so very dependent on the methodologies implemented in the embedded compiler and on the kind of IRs adopted in optimizations.

- *limits in full exploitation of HLS advantages*: often overall design cycle and productivity benefits cannot be fully exploited. For example, optimization of area and power requires fine-tuning the code, which makes more difficult its reuse in other tools and technologies. Constraints posed by different tools also limit the number of ways a designer can influence the microarchitecture of the design, and sometimes the quality of result (QoR) provided by the HLS tool is far from what an experienced designer can do manually. Finally, the design process sometimes degenerates into a long series of optimizations, decreasing the productivity and effectively stripping the code of its portability and flexibility.

The experience described in [15] shows that HLS can significantly increase the productivity of current FPGA users, especially for those that use DSP processors in highly demanding applications. Indeed, they compared the performances obtained by a mainstream DSP processor with the ones obtained using HLS tools on FPGAs. The result was that the latter approach gives

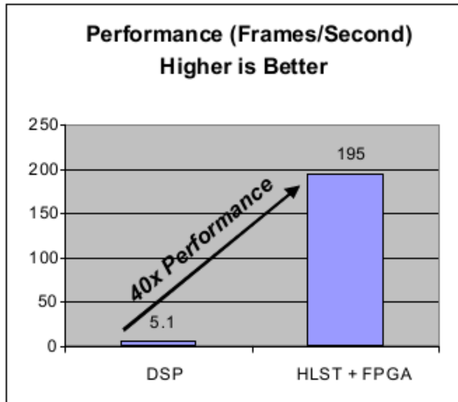

Figure 2.13: *Maximum frame rate achieved for a video application on a DSP processor and an FPGA plus HLS tools.*

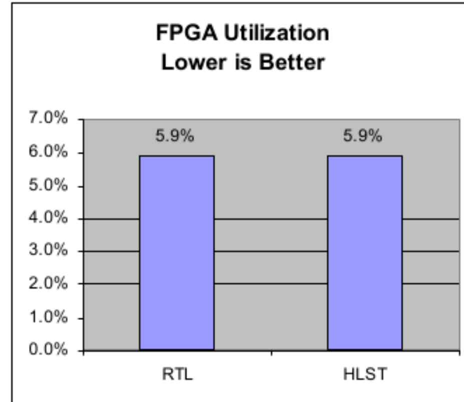

Figure 2.14: *FPGA resource utilization on a wireless receiver, implemented using HLS tools versus hand-written RTL code.*

40X better performance than the prior, as shown in Figure 2.13. Moreover, was found that HLS tools are able to achieve FPGA resource utilization levels comparable to hand-written RTL code, as shown in Figure 2.14. A surprising outcome of this analysis is that the synthesis work for FPGA required a similar level of effort as the one required for the DSP processor, despite historically DSP processor implementation resulted easier. However, these results are strictly related to the application. Indeed, other works, as [14], show that, in some high parallelizable signal processing applications, FPGAs could achieve up to 100X higher performance and 30X better cost-performance than DSP processors.
Unfortunately, HLS manifests some deficiency also in FPGAs domain, the main of them can be summarized in:

- *difficult parallelism and control handling*: this aspect is related to C/C++ adoption as input languages. As above mentioned, such languages give the enormous advantage to offer the highest level of algorithmic exploration, facilitating code restructuring and optimization works. This is the main reason why they should be adopted. At the same time, however, they represent a challenge in parallelism and con-

trol management, due to they sequential execution semantics without explicit support for parallel structures. To successfully face this problem more attention should be paid on properly choose the Intermediate Representations (IRs). A poor IR can prevent to exploit an adequate degree of parallelism, or can provide poor informations about control handling. As we will see in the next, the first step of this thesis work is a deep analysis of the specification dependencies, leading to the definition of a proper IR. Moreover, C-like languages adoption can be not suitable for the description of control-oriented applications, that can be easily represented for example by extended FSMs, or by Petri Nets.

- *variable quality of results*: HLS on FPGAs gives good results and a lot of interest in DSP-domain applications, especially for wireless and wired communications and for image processing, as reported in many technical reports, such as [86]. This fields are stricly related to dataflow-dominated applications and signal processing. However, good results can be generally obtained in control-dominated applications synthesis too, provided a relatively low level of data dependencies. In most cases, tools that are able to efficiently manage dataflow-dominated applications get worst results when applied to controlflow-dominated ones, and vice-versa. Some vendors state that their tools can efficiently handle both data and control flow dominated domains, as for instance Mentor Graphics Catapult C. Other tools vendors, instead, claim effectiveness in a specific domain, as for example Synfora PICO Express, able to efficiently handle control. Despite third-generation HLS tools claim good effectiveness in both dataflow and control, the emphasis on DSP fits in well with the overall perception that third-generation HLS is particularly suitable for signal-processing domains. This is the main reason why a fourth generation multidomain HLS toolset is prospected. Indeed, none of these tools is comprehensive in targeting all the domains: control and dataflow; FPGA and ASIC; random logic and processor-based; and hardware, software, and mixed hardware-software forms.

As a result, these tools do not let designers explore the design space in even a semiautomated format without changing toolsets and investing a lot of work in tracking and optimizing various branches of the design tree.

2.7 Conclusions

From what discussed so far about HLS history, in both research and industrial areas, it is possible to drawn some general conclusions. As we will see in the next chapters, such conclusions represent the basis for this thesis work.
First of all, the adoption of C-like high level languages led the enormous advantage of allowing algorithmic exploration and compiler-based optimizations. On the other hand, such languages do not support explicit parallel constructs. Hence, a proper IR is needed to adequately exploit the parallelism in the given specification. Such IR must take into account both data and control dependencies. Moreover, it has to facilitate the identification of different instructions that can be execute simultaneously, while preserving the original application meaning.
Another key point concerns the difficult acceptance of HLS methodologies. While it is often used in design space exploration, many limitations in other kind of utilizations come up against. To make of HLS the mainstream approach, more general methodologies need to be developed, supporting synthesis processes in a wider range of domains. Despite some vendors claim their HLS tool efficiency in all the domains, such approach results common accepted nowadays only in some application domains, such as DSPs, in which HLS effectiveness is largely proved by a great amount of technical reports. This problem is not related only to application domains, but also to the kind of target technology. For example, in most of the designs, HLS still presents more problems than benefits, usually giving quality of results not comparable with those obtained by hand-writing the RTL code. As discussed before, users acceptance degree is a good means to evaluate real HLS perception

and effectiveness, since the market has shown to faithfully mirror technology changes. In conclusion, HLS is still inadequate to efficiently identify the inherent parallelism that can be exploited in hardware, in a wide range of application domains.

Another relevant conclusion is that the adoption of high-level programming languages describing the specification led to the parallelism extraction problem. The main source of parallelism in a specification are branch conditional constructs, such as loops. Hence, a methodology able to recognize the parallelism and thus to handle with branch conditional constructs should be proposed. As above described, this task can be performed on the basis of a proper IR.

A first step to overcome these problems could be a careful definition of the final architecture, focusing on the *controller synthesis phase*. The most common model adopted for the controller synthesis has been so far the Finite State Machine one. However, different models have been proposed in literature for such purpose, and there exist other solutions well fitting this work. The aim of the controller model renewing should be to improve the exploitation of parallelism, regardless of the application domain.

Moreover, the proposed final architecture must be defined taking into account all these factors. Architectures composed by datapath and a parallel controller structures seem to fit in supporting parallelism exploitation, if properly designed. None of the proposals in this direction, however, can address the parallelism problem without significantly increasing the communication overhead, due to sub-controllers synchronization. Indeed, techniques adopting high-level programming languages are still mainly based on CDFG, thus incurring in parallelism problem, while graphical descriptions based ones incur in the communication overhead problem.

Since the language adopted in this work to describe the specification is a C-like programming language, to propose an HLS methodology able to build a performant controller structure, handling with both the application domains, two relevant choices have to be made, respectively about a proper IR

for its construction and of an adequate architectural model on which to base it. At present, neither of these two aspects have been completely addressed in literature. Alternative proposals, indeed, result not enough convincing to become the mainstream approach, while common ones still presents several drawbacks.
This thesis work aims to fill the above mentioned gaps. More in detail, we will define a proper IR supporting C-like specifications synthesis. Such IR will be obtained starting from another well known IR, the Program Dependencies Graph [22]. Then, this new IR will be used to get a parallel execution model, through the creation of a parallel and distributed controller structure. As will be described in the next, the final architecture proposed in this work is not obtained by decomposing a centralized FSM, and it is not composed of a set of communicating local FSMs. It is instead closer to a direct implementation of the behavioral specification. Such architectural model is proposed to dynamically extract the available parallelism in the specification, regardless of its application domain.
In conclusion, observing HLS evolution, the need of unifying HLS domains was found to be a critical and actual problem for HLS to succeed. In particular, the synthesis in control-oriented application domains has given practical evidence to be a weak link in high level synthesis, leading to a very little acceptance degree in such domains. Moreover, a careful controller synthesis methodology can persuade also ASIC designer in using HLS, facilitating the meeting of the design goals and reducing the limits in full exploitation of HLS advantages.
In such a scenario, this thesis work proposes the integration in the HLS flow of a methodology for building a parallel controller structure able to extract the highest degree of parallelism from a C described specification, regardless of its application domain.

Chapter 3

Proposed Methodology

As above mentioned, the purpose of this thesis work is to propose an approach for the controller synthesis in HLS. The analysis of the state of the art suggests the need of a twofold improvement. On one side a proper Intermediate Representation has to be developed, and on the other side an adequate choice of the controller architecture is required. Both of these aspects will be addressed in the methodology proposed in this thesis. In particular, regards the first aspect, a new kind of IR will be introduced starting from the Program Dependence Graph (PDG) of the specification, while about the second one, an appropriate distributed and parallel controller structure will be defined. It will be also described how to derive the implementation of such architecture from the proposed representation.
More in detail, given an high-level language specification of an application, the addressed problem is to integrate a methodology in the HLS flow to automatically synthesize a distributed and parallel controller structure that:

- is able to extract all the available parallelism in the specification
- enables the concurrent execution of the identified parallel operations, provided that resource constraints are satisfied
- efficiently manages control constructs in the specification

The tasks performed to obtain such controller can be summarized in two macro steps: the *analysis process*, leading to the definition of the Intermediate Representation, and the *controller synthesis process*, including the generation of the hardware needed to control the datapath. As will be described in the following, both of these macro tasks encase a series of sub-activities, that will be compared with the approaches proposed in literature, to present similitudes and differences.
The project organization will be detailed in Section 3.1. Section 3.2 overviews the analysis process, clarifying the steps which it is divided into, and motivating the choices taken in this work. Moreover, in this section a motivational example will be introduced to further clarify the scenario. In Section 3.3, the new proposed IR, namely the *Parallel Controller Graph* (PCG), will be presented. Then, in Section 3.4, it will be explained how the PCG is used to create the logic needed to properly enable instructions execution. In Section 3.5, an overview of the controller synthesis process will be presented, starting the description of the second phase of this thesis work. Such last phase will be directly inferred from the results of the previous analysis phase. In Section 3.6 the proposed architectural model for the controller will be introduced. Such architectural model will be further detailed in Sections 3.7 and Section 3.8. Then, Section 3.9 describes how to generate the controller. Finally, Section 3.10 describes a series of optimization techniques that can be applied to the resulting model.

3.1 Project Organization

As above mentioned, among the phases in the HLS flow shown in Figure 1.3, this thesis focuses on Controller Synthesis. A relevant aspect needing improvements in HLS is related to the parallelism extraction, since C-like high-level languages have sequential execution semantics without explicit support for parallel structures. In this sense a parallel controller structure can contribute to overcome this issue. In literature, such kind of structure

has been already proposed. However, usually parallel controllers are implemented with the support of non trivial communication and synchronization protocols, that significantly increase the communication overhead [51] [44]. Moreover, the proposed controller is also distributed. Distributed structures

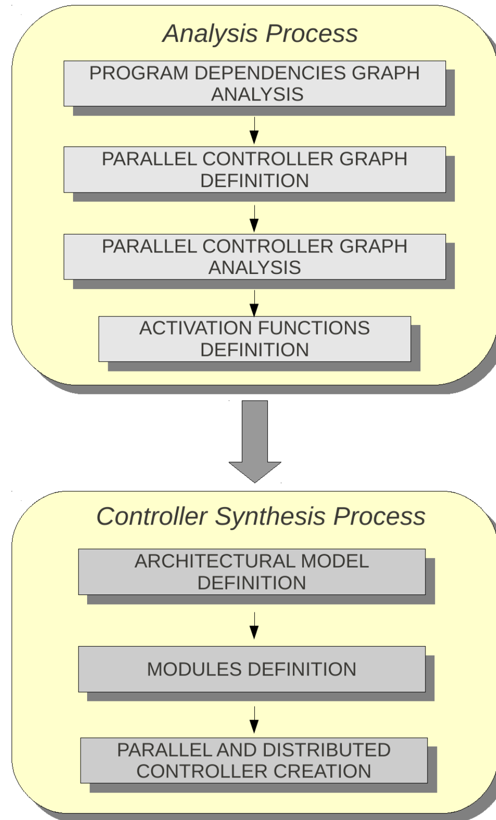

Figure 3.1: *Project Flow: the Steps in which this Project is Organized.*

are usually obtained by a top-down approach that starts from a monolithic FSM and applies decomposition techniques. Hence, a preliminary step needs to be performed for the construction of the centralized FSM. A bottom-up approach seems more suitable, since it allows to avoid the entire FSM construction. The final controller architecture is not composed of a series of FSMs, but it is composed of several modules, interacting according to a token-based scheme, which is a very simple communication paradigm.
Several steps have been considered to realize the proposed model. Such

phases, organized in two macro-tasks, are shown in Figure 3.1. The former is the *Analysis Process*, in which, through an analysis of the Program Dependence Graph, a new Intermediate Representation, called *Parallel Controller Graph* (PCG), will be defined. Then, the PCG will be in turn analyzed to obtain the conditions that enables an instruction to be executed. Such conditions will be represented by a special form of a logic function, in which the meaning of standard logic operators, such as AND and OR, is partially extended. The *Controller Synthesis Process* allocates and connects the modules needed to implement the target controller architecture, according to function associated with each operation.

3.2 Analysis Process Overview

One of the objective of the analysis phase, is to build an Intermediate Representation from which an execution model will be constructed, preserving the inherent parallelism. Such a model can be characterized by a producer/consumer paradigm: an instruction i (i.e. the consumer) can be executed if and only if the instructions on which it depends on (i.e. the producers) have been already executed. Hence, given an high-level language description of the specification, the problem addressed in the analysis phase is to *identify the conditions enabling an instruction execution, preserving the available parallelism*. This task involves a series of sub-tasks, that can be briefly summarized as follows:

1. *Parallelism Identification*: choice of a representation exposing the available parallelism in the specification

2. *Enabling Conditions Identification*: identification of the conditions enabling the execution of each instruction

3. *Enabling Conditions Representation*: choice of a proper representation for conditions that enables instructions execution

In the next section a motivational example will be introduced to better explain such problems and how they have been addresses. More in detail, this example is presented with the aim to clarify what is needed to perform an analysis oriented to the construction of a parallel controller structure, and to explain the choices made in this thesis work.

3.2.1 Motivational Example

A representative example of C-written specification will be now introduced to better explain how the analysis phase should be carried out. The selected example, shown in the following, shows some exploitable parallelism. Such parallelism is represented by the presence of two loops, loop1 and loop3, that can be potentially simultaneously executed. Moreover, it is characterized by an elevated number of both data and control dependencies, increasing the number of control steps needed to schedule all the operations, and complicating the parallelism extraction.
In this case, it may result difficult to statically identify a control step in which each operation can start, since the program presents conditions whose evaluation result determines which instructions must be executed. Conditions evaluation results may be unknown at compile time. For example, it may be unknown at compile time the number of iterations for loop2, since it depends on the value of the variable n2, which in turn depends on the value of the variable i, which in turn depends on the value of the variable a that is an incoming argument of the function. For this reason it results impossible to know its value before run time.
For this reason, a general approach should not aim to find the exact control step in which activate each operation, but only to identify the conditions that enable their execution. For example, the condition enabling the execution of the instruction 16 can be expressed as "after the values for both the variables read, a and b, have been defined". Moreover, considering the producer of variable *b*, i.e. instruction 14, another kind of enabling condition can be identified. Instruction 14 execution, indeed, is possible only when the branch

condition of instruction 13 has been evaluated as true. Obviously, this informations has to be formally identified and properly represented. Moreover, all the considerations made so far need to be formalized to define a general approach.

```
    void MotivationalExample(int a, int b, int c, int * out)
    {
1:      int i=0;
2:      while(i<a)                #loop1
        {
3:          int j=0;
4:          int n2=i+1;
5:          while(j<n2)           #loop2
            {
6:              int t1=j+1;
7:              c=a+t1;
8:              i=i+c;
9:              j=j+1;
            }
10:         i=i+1;
        }
11:     int k=0;
12:     while(k<10)               #loop3
        {
13:         if(a>k)
            {
14:             b=k+a;
            }
15:         k=k+1;
        }
16:     int t2=a*b;
17:     int res=t2+c;
18:     *out = res;
    }
```

3.2.2 Parallelism Identification

The key to identify the parallelism available in the specification is the adoption of a proper model as Intermediate Representation (IR). As above mentioned, the most common approaches adopted in high level synthesis use graphs as IRs. More in detail, the *Control Data Flow Graph* (CDFG) represents the standard approach. In CDFG-based methodologies control dependencies are derived from the CFG, while data dependencies from the DFG. Even if data dependencies between operations are well shown by the CFG, it fails on recognizing the available parallelism, i.e. it serializes control constructs introducing "false"control dependencies.
This issue will result clearer considering the CFG in Figure 3.2, obtained from the proposed example. The two loops, loop1 and loop3, in the specification could be executed in parallel, not having neither data nor control dependencies between their instructions, but the CFG serializes them introducing an edge (from node 2 to node 11), and so a "false"dependency. Thus, a CDFG based execution model will ignore the parallelism provided by the two parallel loops. Even if in this particular case, front-end optimizations, such as loop merging, could reduce the problem's impact, a general solution is definitely required to deal with these situations.
For this reasons, the *Program Dependencies Graph* (PDG) seems more adequate to be adopted as the starting point of the analysis process, representing the so called *minimum dependencies* in the specification.
Formally, the minimum dependency relation can be defined as follows.

Definition 3. *Given a* $PDG = G(V, E)$*, where*

- $v \in V$ *are the graph nodes, i.e. the instructions*
- $e = (v_1, v_2) \in E$ *are the graph edges, with* v_1 *source node and* v_2 *target node*

we define the minimum dependency relation M *among the set of instructions* V*, as:*

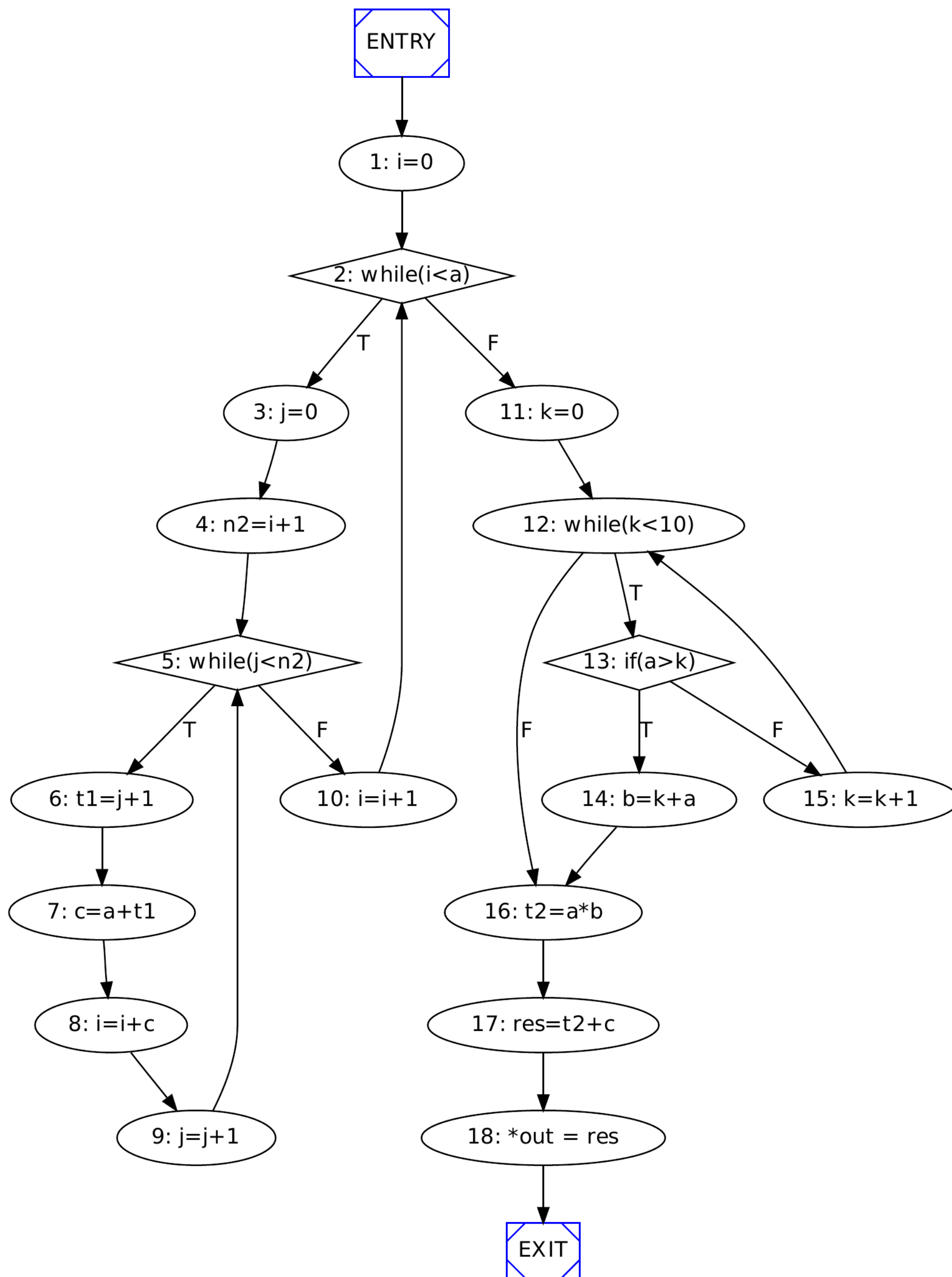

Figure 3.2: *CFG corresponding to the Motivational Example C code.*

$$v_1 M v_2 \Leftrightarrow \exists e \in E | e = (v_1, v_2).$$

Based on this property, the PDG is able to address the parallelism identification problem.
In fact, the information contained in PDG allows the recognition of different sources of parallelism exploitable in the specification. Consider, for instance, the PDG shown in Figure 3.3, obtained from the proposed example. Solid black edges represent control dependencies, blue dotted edges represent true data dependencies and green dotted edges shows false data dependencies (i.e. anti and output ones). In this worth noting that, in Figure 3.3, only one edge is represented even when there are multiple dependencies between two edges. More in detail control dependencies have to be considered as the more relevant ones, followed by true data dependencies, and by false data dependencies. Such representation clearly shows that there is not any real dependence between nodes 2 and 11, hence loop1 and loop3 can start their execution simultaneously.
The PDG seems a good starting point to create a parallel execution model, since an important property holds. Let us define a *level* for each operation in the program dependence graph as the maximum of the parents levels plus one. For instance, considering the PDG in Figure 3.3, level 0 will be assigned to the entry node, level 1 will be assigned to all the entry node successors, i.e. nodes 1 and 11, and so on. Note that level 3 has to be assigned to node 13, since the node with maximum level among its predecessors is node 12, that has level 2. Table3.1 reports the level assignment for the operations, according to their position in the PDG.
As described in [22], all the operations at the same level in the PDG can be executed simultaneously, since no dependencies between them occur. Hence, for example, the execution of the operations 3, 4 and 13 can overlap, if an adequate number of resources is available. Moreover, the PDG allows to discover the *highest* level of parallelism in the source code: if all the instructions in one PDG-level are executed in the same control step, the ASAP instruction scheduling is obtained, according to the set of minimum dependencies,

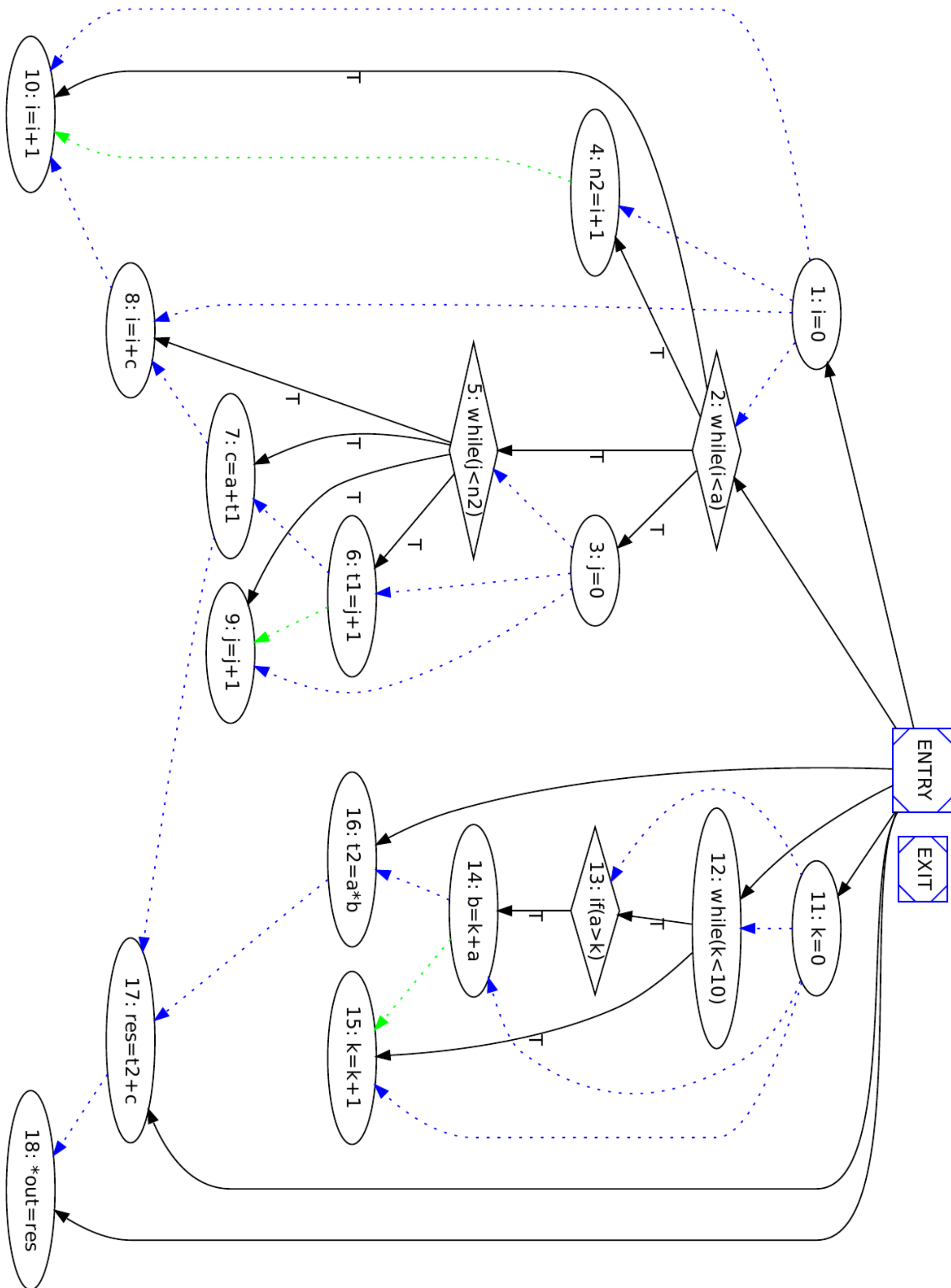

Figure 3.3: *PDG obtained from the motivational example C specification.*

LEVEL	*OPERATIONS*
0	ENTRY
1	1, 11
2	2, 12
3	3, 4, 13
4	5, 14
5	6, 15, 16
6	7, 9
7	8, 17
8	10, 18

Table 3.1: *Level assignment for the Operations in the PDG.*

that, by definition, exploits the highest level of parallelism.

3.2.3 Enabling Conditions Identification

The section describes how the PDG can be also adopted to identify and describe the conditions that enable instructions execution. Considering, for example, the instruction 3 in the motivational example, it is possible to infer the information about such conditions from the PDG in Figure 3.3. Indeed, the PDG shows that 3 depends on instruction 2 by means of the oriented edge from 2 to 3. Thus, the PDG suggests that 3 can be executed after the instruction 2 has been executed. However, saying that 3 has to be executed after 2 is not enough, since in this case the nature of the dependency between the two operations is of control type. This means that another condition must be added, i.e. 3 can be executed after 2, and the execution of 3 is subject to the positive evaluation of the condition "$i < a$" . In any case the PDG shows also this information by marking with a "TRUE" label the edge from 2 to 3. However, the PDG is not able to provide all the informations needed. Hence, the PDG presents a relevant drawback: it does not show all the informations

needed by the execution flow. This fact makes it unusable as basis to build the parallel controller structure. Hence, it has to be adequately expanded to construct a real execution model. To understand this issue, consider, for example, instruction 17. The information inferred from the PDG is that 17 has a data dependency with both 7 and 16. Hence, it can be executed after both 7 and 16 have been executed. This information is clearly wrong. Indeed, operation 7 belongs to loop2's body. Thus, it has to be executed many times before the final value for the variable c is computed. Associating to 17 the above described activation function, after 7 is executed for the first time, the execution of 17 can start, provided that also 16 has been executed. As a consequence, 17 would read the wrong value for the variable c. The missing information in this case is that 17 cannot start until the termination of loop2 execution. Moreover, since loop2 belong to loop1, 17 must be waiting for loop1 termination too. This is just an example in which PDG fails in identifying enabling conditions. In particular, it is clear that control and data dependencies are not enough to identify the minimal set of the enabling conditions for an operation. There exist, indeed, situations in which the flow has to be necessarily considered. In conclusion, a new IR called Parallel Controller Graph is proposed to address these issues. By extending the PDG with the missing information.

3.2.4 Enabling Conditions Representation

After identifying the conditions enabling the execution of an instruction, they have to be stored in a convenient and easy-to-manage form. Such representation should be as concise as possible, and should be in an optimizations-compatible form. To choose a proper representation for the enabling conditions some preliminary considerations are needed.
The proposed model is event driven, following, as mentioned in the previous sections, the producer/consumer paradigm. More in detail, given the minimum dependency relation M induced by the PDG, $v_1 M v_2$ denotes that the execution of v_2 must wait until v_1 completion to start, thus v_1 repre-

sents a producer while v_2 represents a consumer. It is also possible to say that the execution of v_1 *activates* v_2, leading to the following interpretation of the statement $v_1 M v_2$: the execution of v_1 *produces an activation signal* consumed by v_2, enabling its execution. This last interpretation fits better in representing also enabling conditions induced by control dependencies, in which the conditional node evaluation result, showed through labelled edges, must be considered. Indeed, such last interpretation is more related to the product (activation signal) with respect to the producer (instruction), allowing the consideration of the product properties (in this case labels).
If an instruction v_2 has more than one incoming edges, i.e. $\exists v_{1a}, v_{1b} \in V$ such that $v_{1a} M v_2, v_{1b} M v_2$, then the activation signal for v_2 is obtained through a proper combination of the signals associated with the incoming edges, is called *activation function.* Furthermore, given an edge $e = (v_1, v_2)$, with v_2 having only e as incoming edge. This combination the execution of v_1 does not always lead to the activation of v_2: in fact, in the case of diramation constructs, the execution of the conditional instruction will not activate both the true and the false branch. Thus activation functions must consider these situations too, taking into account the result of the condition evaluation.

3.3 Parallel Controller Graph

Starting from the program dependencies graph, we introduce a new kind of internal representation, called *Parallel Controller Graph.* The name chosen for the graph denotes the objective of building a parallel controller structure, providing a proper execution model, i.e. the enabling conditions for the operations. The first issue that required to modify the PDG is the need of more informations about the control flow. This lead to the following formal definition for the PCG.

Definition 4. *The Parallel Controller Graph (PCG) is defined as an oriented graph $G(V, E)$, where V is the same set of nodes of the corresponding $PDG = G'(V, E')$, and the set E of edges is defined as*

$$E = D_T \cup D_{AO} \cup \cup C_U \cup C_L \cup C_{BE} \cup F_U \cup F_L$$

where:

- D_T *denotes the set of edges representing RAW (flow, or true) data dependencies; each edge* $d_t = (v_1, v_2) \in D_T$ *has a label indicating the variable defined by* v_1*;*
- D_{AO} *denotes the set of edges representing WAR (anti) and WAW (output) data dependencies; each edge* $d_{ao} = (v_1, v_2) \in D_{AO}$ *has a label indicating the variable used by* v_1 *and defined by* v_2 *in the case of WAR dependencies, or defined by both* v_1 *and* v_2 *in the case of WAW dependencies;*
- C_U *denotes the set of edges representing control dependencies without any condition, i.e. all the control dependencies edges outgoing from the entry node belong to* C_U *; the subscript U indicates that this kind of edges are unlabeled;*
- C_L *denotes the set of edges representing control dependencies with some condition; each edge* $c_l = (v_1, v_2) \in C_L$ *has a label indicating which branch of the diramation node the edge is associated to;*
- C_{BE} *denotes the set of loops back-edges.*
- F_U *denotes the set of control flow edges without any condition;*
- F_L *denotes the set of control flow edges with some condition;*

Labels can be viewed as properties characterizing some kind of edges. Labeled edges could be described by the 3-tuple (v_1, v_2, L), where v_1 represents the source node, v_2 the target node and L the label.
In the next, for each node v, the following notation will be adopted:

- $in(v) = \{e_i = (v_i, v) \in E\}$ will indicate the set of incoming edges;
- $out(v) = \{e_o = (v, v_o) \in E\}$ will indicate the set of outgoing edges;

- $fanIn(v) = |in(v)|$ will indicate the number of incoming edges;
- $fanOut(v) = |out(v)|$ will indicate the number of outgoing edges.

As above mentioned, the proposed model is based on the concepts of activation signal and activation function. For each node, the corresponding activation function depends on the set of the node's incoming edges $in(v)$, according to the above described product-related interpretation of the producer/consumer paradigm.
Given a $PDG = G'(V, E')$, the algorithm to obtain the set of edges E of the corresponding $PCG = G(V, E)$ can be summarized in the following steps:

1. A transitive reduction is performed, among all the control edges having the entry node as source, i.e. belonging to the set C_U: each edge $e = (ENTRY, v)$, such that $\exists e_1 = (ENTRY, v1)$ and exists a path from v_1 to v, is removed. The reason why this edges are eliminated is that, in such situations, it results redundant the condition that the entry node activates the execution of v, since the activation of v depends also on other instructions execution. Since no instructions can be executed until the program starts to run, i.e. until the entry node is activated, dependencies from the entry node are really needed only for those nodes depending only on the entry node. However, this is just an optimization that, if not performed, does not affect the PCG correctness.

2. Loops require additional labeled control flow edges $e_l \in F_L$ and back-edges $e_b \in C_{BE}$. Two simple algorithms to add such edges will be presented in the next paragraphs, including explanations about the reasons that make them necessary.

3. Nodes without outgoing edges are linked to the exit node; such edges will be included in the set $F_U \subset E$. This information is related to the specification flow. In other words, it is a means to identify when the computation terminates.

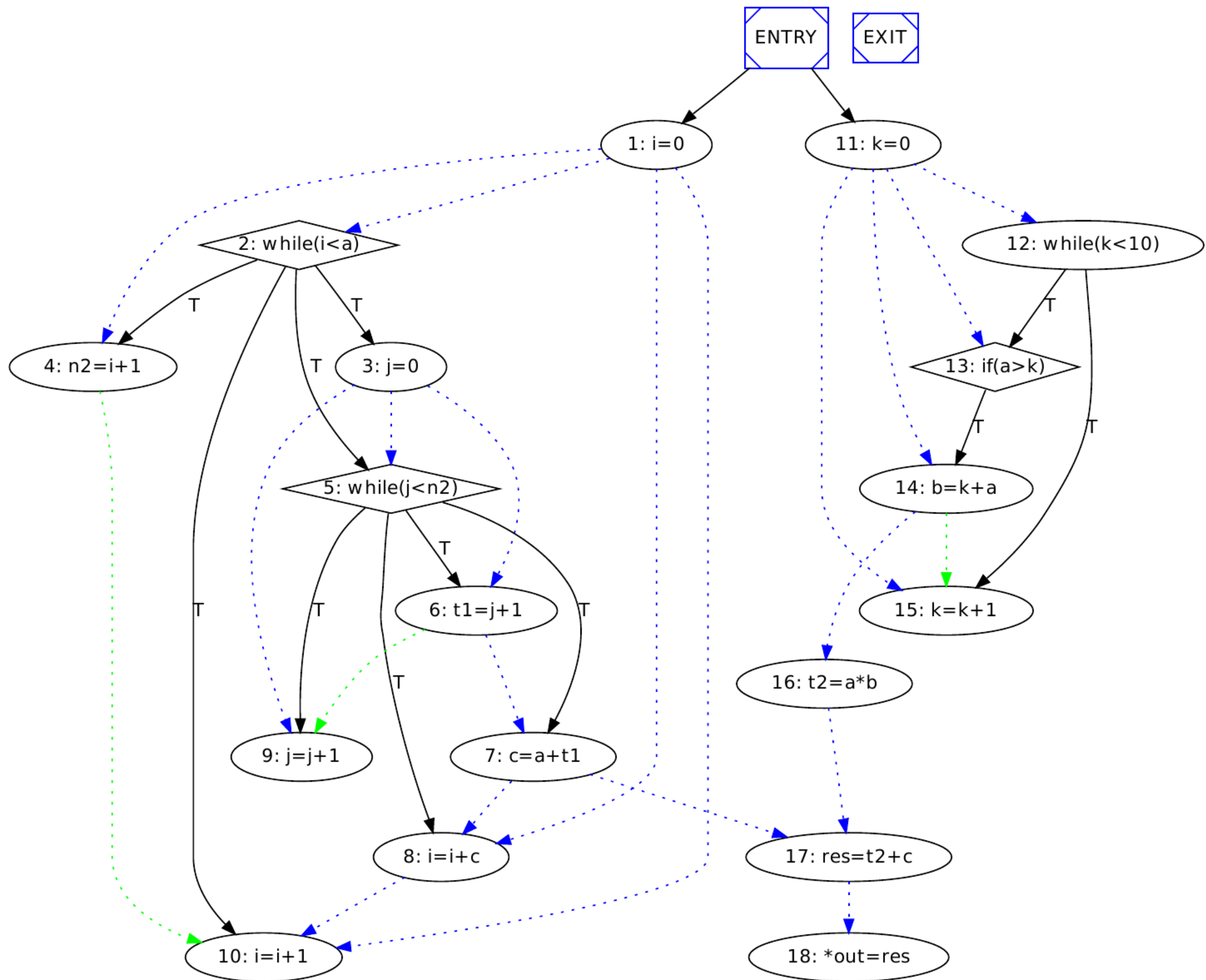

Figure 3.4: *First version of the PCG for the motivational example obtained from the corresponding PDG by applying the first steps of the PCG construction algorithm.*

Figure 3.4 shows a first version of the PCG, obtained from the corresponding PDG by applying transitive reduction on control edges starting from entry.

3.3.1 False-labeled control flow edges insertion algorithm

Consider the first version of the PCG in Figure 3.4, obtained for the motivational example C specification and the corresponding PDG in Figure 3.3. Node 16 is not control dependent on node 12, since does not exist, in the corresponding CFG, a path reaching the exit point from node 12 not containing node 16; thus node 16 is only data dependent on node 14. Since activation signals are associated with dependencies, i.e. with edges, in this case the execution of 14 leads to an activation signal that allows 16 to start (being $fanIn(16) = 14$), even if the while loop, loop3, is not finished yet. To handle with such situations, additional edges are introduced, having the loop header (e.g. node 12) as source and the data dependent node (e.g. node 16) as target. The rationale for making such edges as control flow edges, instead of the control dependence ones, will be clarified in Section 3.4.4. Introducing a false-labeled control flow edge from 12 to 16 is equivalent to state that instruction 16 must wait until loop3 termination to start. The proposed algorithm requires loops detection: this task is usually performed by the front-end, and recognizes the loop constructs in the source code, associating them an identifier $loopID$. In the adopted representation, each instruction of the source code is assumed to be contained in a loop: instructions belonging to a loop are associated to the corresponding $loopID$, all the others are associated to a dummy loop, namely $loop_0$. The set of the instructions belonging to a loop is denoted by the loop identifier itself. Once loops are detected, a loop forest is built.

A *loop forest* is a tree representing the nesting relations between the identified loops, where the root node is $loop_0$ (see Figure 3.5). Given a loop forest, $loop_j$ is defined as an *inner loop* with respect to $loop_i$ (*outer loop*) if there exist a path from $loop_i$ to $loop_j$ in the loop forest tree.

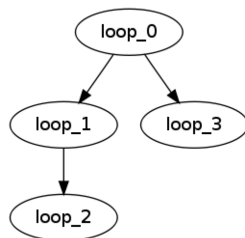

Figure 3.5: *Loop forest tree for the motivational example specification.*

Considering again the first version of the PCG Figure 3.4, since node 16 is not control dependent on node 12, node 17 is not control dependent on node 5. However, in this case, instruction 5 belongs to loop2 that is an inner loop with respect to loop1. Hence, loop2 termination is not enough to enable instruction 17 execution, since loop2 will be executed as many times as the number of iterations of loop1. Thus, 17 has to wait for loop1 termination too. To handle also with such situations, a false labeled edge will be added from the header of the outermost loop in the loop forest (e.g. node 2), with exception of loop0, to the data dependent instruction (e.g. node 17).
In conclusion, a false-labeled control flow edge must be added each time it occurs a data dependency with source an operation belonging to a $loop_i$, with $i \neq 0$, and target an operation belong to a $loop_j$, with $i \neq j$. However, additional aspects have to be considered.
Consider, for example, node 3 and 9 in Figure 3.4. They are data dependent. Moreover, node 3 belongs to loop1, while 9 belongs to loop2. Hence, all the so far mentioned conditions for a false-labeled control edge addition hold. However, in this case adding such control flow edge is equivalent to say that instruction 9 must wait until loop1 termination to start. It is obviously a paradox, since loop1 cannot terminated if instruction 9 has not been executed, since it belongs to a loop1's inner loop. Hence, if a data dependency occurs, with source an operation belonging to a $loop_i$, and target an operation belong to a $loop_j$, with $loop_j$ inner with respect to $loop_i$, no edges must be added.
Finally, consider the opposite situation, i.e. when a data dependency occurs,

with source an operation belonging to a $loop_i$, and target an operation belong to a $loop_j$, with $loop_i$ inner with respect to $loop_j$, as for example, for nodes 8 and 10 in Figure 3.4. In this case the false-labeled control flow edge is needed, since 10 must wait for loop2 termination before to start.
On the basis of this analysis, it is now possible to define the algorithm for the false-labeled control flow edges insertion. Such edges will be indicated with $e = (s, t) \in F_L$. The algorithm works as follows:

1. for each loop k (except $loop_0$), instructions are progressively analyzed in order to find data dependencies $d = (v_1, v_2) \in D_T \cup D_{AO}$, such that:

    - $v1 \in loop_k$
    - $v2 \in loop_w, loop_w \neq loop_k$
    - $loop_w$ is not an inner loop with respect to $loop_k$

2. each identified dependency $d = (v_1, v_2)$ is considered. The front-end associate to each $loop_k$ a loop header, that generally could include more than one instruction, composing a basic block. Denoting with $header_cond_k$ the loop condition test instruction of $loop_k$ header's basic block, the source node s of the additional control edge is set to the node corresponding to $header_cond_k$.

3. the target node t is set to v_2;

4. the edge just identified is labeled as a false-branch control flow dependency, denoting that the loop restarting condition is not satisfied, so the loop execution is finished;

5. finally the edge $e = (s, t, L = false)$ is added to the graph.

Figure 3.6 shows the resulting PCG after false-labeled edges insertion for the motivational example specification.

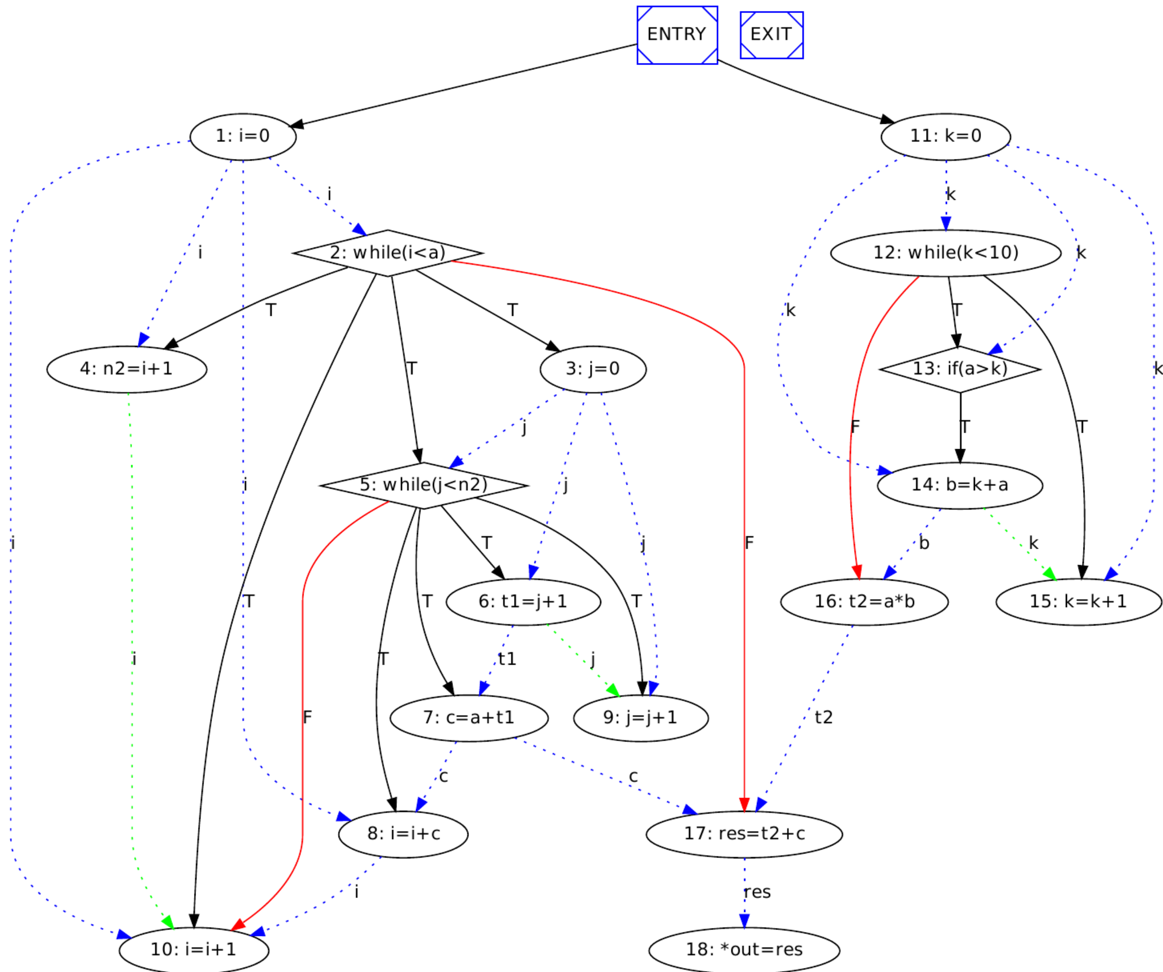

Figure 3.6: *PCG for the motivational example after false-labelled control flow edges (red edges) insertion.*

3.3.2 Back-edges insertion algortihm

When a loop is executed for the first time, its entry point is activated by the associated activation function, which depends on the node's incoming edges; using the current version of the PCG, it becomes impossible to establish how to manage the subsequent iterations of the loop, since no edges from the loop itself reach the header, making entry point execution start independent with respect to activation signals that should come from the loop body.

Consider, for example, node 2 in Figure 3.6. In this partial version of the PCG, node 2 results activated after node 1 has been executed. Since node 1 is executed one time, also node 2 does. Indeed, there not exists any relation between node 2 and the instructions belonging to loop1 from which it is possible to infer that node 2 has to be possibly executed many times. Moreover, also the information about conditions enabling the re-activation of node 2 misses.

To face this problem, back-edges are introduced in the graph. These edges must unequivocally identify the termination of a loop iteration. Usually, given a loop, a back-edge is considered as an edge having source the last instruction of the loop body, and target the loop header. This implies that any given loop is characterized by a unique back edge; moreover, most common loop detection algorithms work identifying back-edges in the CFG and thus associating them to a loop. This approach requires that an execution flow is well defined, but the current PCG version does not satisfy this requirement. The main difference in using PCG instead of CFG is that the latter supports the representation of loops with single entry point and single exit point, while the former supports the representation of multiple entry and exit points loops. Indeed, from the partial PCG representation, it is possible to identify a set of instructions, where each of them may be the last one of the loop. Similarly, the entry point of a loop may be not unique in the partial PCG, so it is possible to identify a set of instructions that may be executed for first in the loop iterations. This is an interesting feature, since the execution order is not known in advance, giving more chances of opti-

mization. Moreover, this representation supports parallel execution, since a set of instructions can be simultaneously executed as first/last of the loop.
In the next, for each loop k, the following notation will be adopted:

- $first(loop_k) = \{v_f \in V | \forall e = (v_i, v_f) \in in(v_f), v_i \notin loop_k\}$ will indicate the set of nodes not depending on any other node in the loop, i.e. the set of instructions that can be executed for first in the loop.
- $last(loop_k) = \{v_l \in V | \forall e = (v_l, v_j) \in out(v_l), v_j \notin loop_k\}$ will indicate the set of nodes such that does not exists any other node in the loop depending on them, i.e. the set of instructions that can be executed for last in the loop.

It should be clear at this point, that in the proposed representation more than one back edge is associated to the same loop.
Denoting with k the number of detected loops, the algorithm to add such back-edges in the graph works as follows:

1. for each $k \neq 0$, the set $first(loop_k)$ is computed;
2. for each $k \neq 0$, the set $last(loop_k)$ is computed;
3. the set of back-edges is computed as

$$D_{BE} = \bigcup_k \{e = (v_l, v_f), v_f \in first(loop_k), v_l \in last(loop_k), k \neq 0\},$$

4. each edge belonging to D_{BE} is added to the graph.

Some clarifications are needed for nesting loops. The proposed algorithm is able to handle even loop nestings under the assumption that the front-end produces a representation for the loops structure composed by three parts: header, body and trailer. Such representation is often adopted by front-ends. An example for such model is shown in Figure 3.7. Header and trailer are single basic blocks, while the body can be composed by several basic blocks. The trailer basic block represent the *exit* point of a loop. The instructions inside the trailer belong to the loop immediately higher in the loop forest.

Considering the example in Figure 3.7, loop2 trailer instructions belong to loop1 body, and loop1 trailer instructions belong to loop0. Adopting such assumptions ensures that, for each loop k, the set of instructions that can be executed for last in the loop cannot belong to an inner loop. For example, the set of instructions that can be executed for last in loop1 will never belong to its inner loop, loop2, since loop2 body instructions must be always followed by loops2 trailer instructions, i.e. by instructions belonging to loop1. This fact makes legal the definition of the set $last(loop_k)$ as above described, making also correct the presented algorithm.
However, other loop graph construction assumptions are possible. To handle with such situations, the algorithm should be slightly modified. Indeed, in other representations for loops structures may happen that the some of the instructions that can be executed for last in an outer loop belong to an inner loop, since the loop trailer may miss. Such instructions would never be detected as belonging to the set $last(loop_{outer})$, since such set contains, by definition, only instructions belonging to $loop_{outer}$.
A more general version of the algorithm for the back-edges insertion works as follows:

1. if a loop has no inner loops associated with, back-edges are added as explained above;

2. in the case of two-level loop nestings, with $loop_O$ the outer loop and $loop_i$ the i-th inner loop,

    - if $\nexists e = (v_o, v_j) \in out(v_o), v_o \in last(loop_O)$ such that $v_j \in loop_i \forall i \in I$, with I the set of inner loops, then the previous algorithm is applied;

    - if $\exists e = (v_o, v_j) \in out(v_o), v_o \in last(loop_O)$ such that $v_j \in loop_i$, then labeled-back edges must be added; indicating them as $e_{lbe} = (s, t_f)$, the source node s of each edge is the one corresponding to $loop_i$ condition test instruction, while the targets are nodes $t_f \in first(loop_O)$; the obtained edges must be false-labeled.

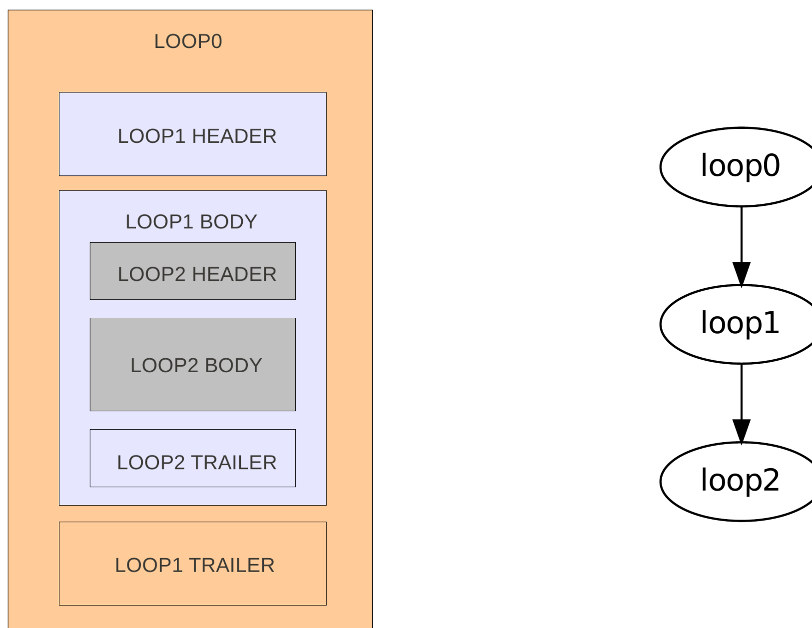

Figure 3.7: *A possible representation produced by the front-end for loops structure and the loop forest of the corresponding specification.*

3. in the case $loop_i$ is not an innermost loop, i.e. there exist other loops in the loop forest tree that are inner with respect to $loop_i$, then it is needed to restart the analysis from point 1, considering this time $loop_i$ as an outer loop.

In other words, the situation above described is recognized by the presence of an edge with an instruction belonging to $last(loop_O)$ as source, and an instruction belonging to $loop_i$ as target. Such edge indicated a dependency from an instruction of the outer loop to an instruction of the inner loop. This means that the inner loop instruction has to be executed after the outer loop instruction has been executed. Hence, the outer loop instruction has been wrongly recognized as belonging to the set of instructions that can be executed for last in the outer loop. Moreover, it means that loop inner termination is the right condition for loop outer to re-start. Indeed, since the target of the data dependency belongs to the inner loop, waiting for its execution is not enough. The correct value will be produced only after the

loop in which is contained, i.e. loop inner, is terminated. For this reason a false-labeled back-edge is added with source the inner loop header condition, and target the outer loop header condition.
Notice that, also applying the simplified version of the algorithm, in the case of end-condition loops, if the condition evaluation node belongs to $last(loop)$, backedges could require to be labeled. Furthermore, this situation may occur rather than not on the basis of front end code transformations and of the kind of IR produced. For example, many compilers usually insert additional instructions at the end of the loops, making them never end with a diramation node: adopting this internal representation in no cases labels will be needed for backedges.
Finally, Figure 3.8 shows the complete version of the PCG, obtained after back-edges insertion and after connecting the nodes without successors with the exit node through unlabeled control flow edges (step three of the PCG construction algorithm).

3.4 Activation Function

the producer/consumer paradigm implies that, when an operation is executed, a *done signal* is produced and delivered to the other instructions depending on it. Anyway, as described above, the reception of a done signal does not always result in the activation of a given operation, thus *activation functions* are introduced. The parallel controller graph representation allows to associate the signals produced by an operation i_1 with the outgoing edges of the vertex v_1 associated to i_1. Similarly, the signals $s_1, s_2, ..., s_k$ that enable the execution of an instruction i could be associated with the incoming edges $e_1, e_2, ..., e_k \in in(v)$ of the vertex v associated to i. Thus, given a node $v \in V$, the activation function $AF(in(v))$, or simply $AF(v)$, can be described as the signal obtained properly combining the done signals associated to edges belonging to $in(v)$ (see Figure 3.9). Because of the two-way correspondence between them, as previously done for nodes and operations, in the next sig-

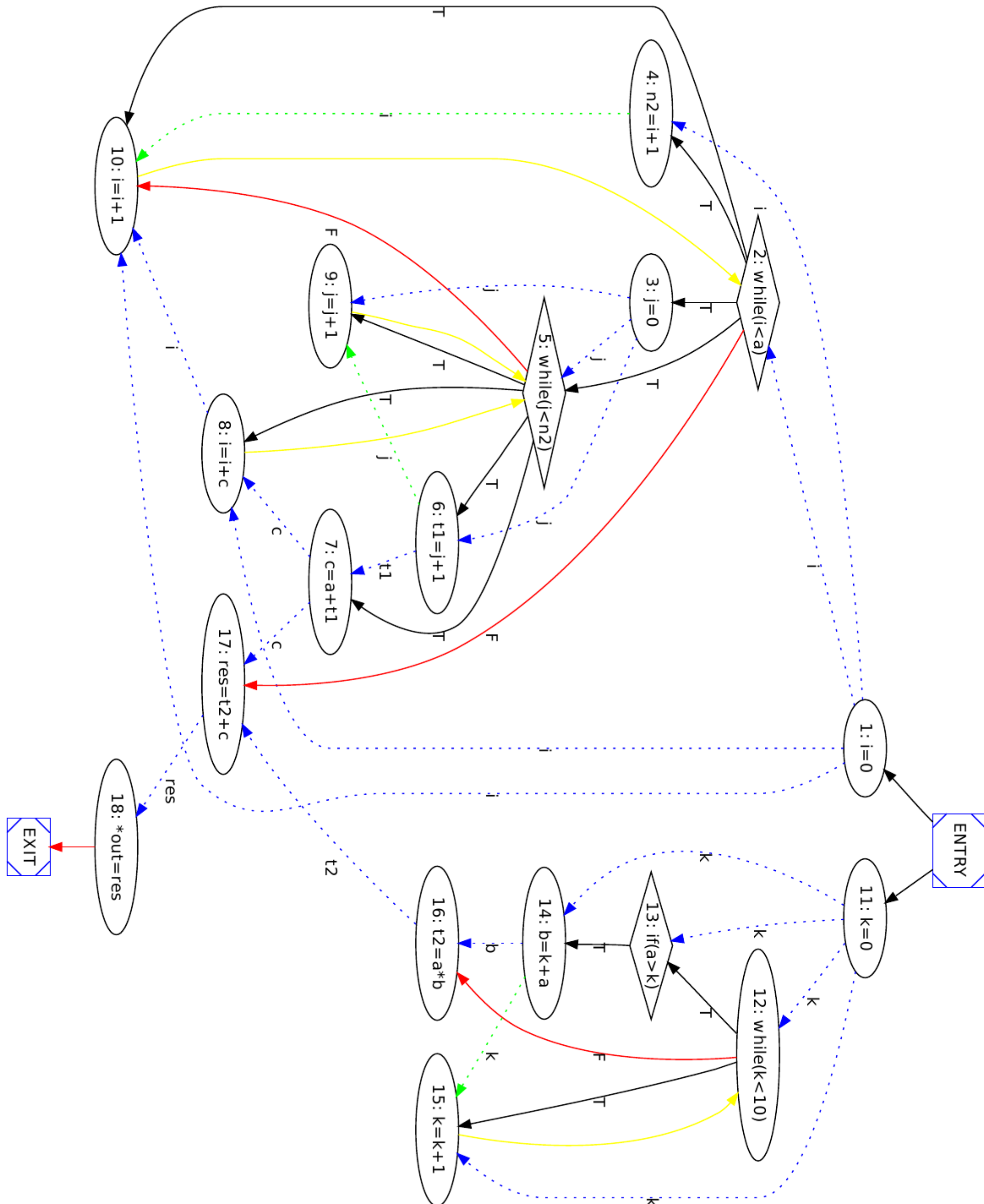

Figure 3.8: *Complete version of the PCG for the motivational example.*

nals and the corresponding edge in the PCG will be alternatively used.

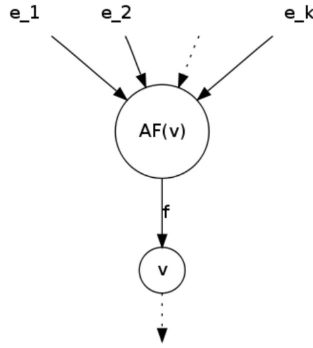

Figure 3.9: *Activation function graphical representation*

The activation function is obtained by combining the done signals through two basic operators:

- the *join* operator $\wedge$, corresponding to the logic operator *and*;
- the *or* operator $\vee$, corresponding to the logic operator *or*.

This connection with Boole's algebra is meaningful since the boolean events of signal productions/receptions are considered as operands. In the physical implementation, a signal will be assumed as generated when it has a high value. Given two signals s_1, s_2, whose generation denotes the occurred execution of operations v_1 and v_2 respectively:

- $s_1 \wedge s_2$ returns 1 as a result, thus generating a signal, iff both s_1 and s_2 have already been generated, i.e. both v_1 and v_2 have already been executed;
- $s_1 \vee s_2$ returns 1 as a result, thus generating a signal, iff at least one between s_1 and s_2 has already been generated, i.e. at least one between v_1 and v_2 has already been executed.

In addition to the two basic operators, another one is needed to be introduced, to handle with labeled edges belonging to the sets C_L and F_L. Given a node v, the presence of an edge $e = (v_1, v, L) \in in(v) \cap \{C_L \cup F_L\}$ denotes that v_1

is a conditional operation and that v depends on the result of the condition evaluation performed in v_1. Thus, execution of v is subject not only to the execution of v_1, but also to the result of its evaluation.
For this reason, the unary operator $\wedge_{cond}$, whose operand is a labeled edge belonging to $\{C_L \cup F_L\}$, is defined. Indicating with $res(v_1)$ the outcome of the condition evaluation in v_1, $\wedge_{cond}(s)$, with s the signal associated to $e = (v_1, v, L)$, returns a high value iff both these conditions hold:

- s has already been generated, i.e. v_1 has already been executed;
- $res(v_1) = L$.

Moreover $\wedge_{cond}(s_1)$, with $s_1 \notin \{C_L \cup F_L\}$, returns the signal s_1 itself.

Property 1. For any conditional node v_c, the same signal results from the appliance of the $\wedge_{cond}$ operator to any of its outgoing edges $e = (v_c, v_j, L)$ having the same label L, i.e. denoting with V_c the set of the conditional nodes:
$\forall\ v_c \in V_c$, $\forall\ s_1, s_2, ..., s_j$ associated to its outgoing edges $e_1 = (v_c, v_j, L_1)$, $e_2 = (v_c, v_j, L_2)$, ..., $e_j = (v_c, v_j, L_j)$, with $L_1 = L_2 = ... = L_j$:

$$\wedge_{cond}(s_1) = \wedge_{cond}(s_2) = ... = \wedge_{cond}(s_j)$$

Hence, it is possible to adopt the following notation:

$$\wedge_{cond}(s_j) = \wedge_{cond}(v_c, v_j, L) = \wedge_{cond}(v_c, L)$$

since the resulting signal does not depend on the target of the control edge.

Given a set of signals S, the following terms are introduced:

- $\prod_{s \in S} s = \prod S$, the result of the appliance of join operator to all the signals belonging to S;
- $\sum_{s \in S} s = \sum S$, the result of the appliance of or operation to all the signals belonging to S;

- $\prod_{cond} S$, with $S \subseteq \{C_L \cup F_L\}$, the set $\{s_{cond} = \wedge_{cond}(s), s \in S\}$

Obviously, if $|S| = 1, S = \{s\}$, then $\prod S = \sum S = s$.
Remarking that the set E of edges characterizing the PCG can be partitioned as

$$E = D_T \cup D_{AO} \cup C_L \cup C_{BE} \cup F_U \cup F_L,$$

some preliminary definition will be next introduced to formulate the activation function.

3.4.1 Control Path and Control Path Signal

Consider the node 14 in the PCG obtained from the motivational example, shown in Figure 3.8, focusing on control dependencies. The operation 14 is control dependent on the operation 13. Hence it has to be executed only if the variable a assumes a value greater than the one assumed by the variable k. The operation 13 is in turn control dependent on the operation 12. Hence it has to be executed many times while the variable k assumes a value less than 10. In this case it is possible to state that instruction 13 will be executed exactly ten times since it is known the number of iterations for loop3, but this information is not available in general. Indeed, a general approach should take into account that loop3 may not be executed at all, since the condition test can result false before the begin of the first iteration. Finally the operation 12 is in turn control dependent on the *entry node*, since all the instructions not having incoming control edges depend on the *entry node*. In conclusion, node 14 has to be executed while k is less then 10 and only if a is greater than k. From such considerations it is possible to infer that a complete formulation of the control part of the activation function should not consider just the activation signals associated with the immediate predecessors in the PCG. It should consider, instead, the presence of a control-edges path, starting from the entry node and terminating into the considered node. Consider, now, node 16 in the same example. It is connected to node 12 by means of a false-labeled control flow edge, expressing the condition that the

execution of 16 must follow the execution of 12 when the condition $k < 10$ is false. Hence, also conditions coming from labeled control flow edges must be considered. For this reason, in the next, conditions deriving from both control dependencies and control flow edges will be handle.
Given a vertex $v \in V$, an *activation edge* for v is defined as $e_{act} \in \{C_U \cup C_L \cup F_L\}$ such that e_{act} must be produced in order to execute e. The set of all the activation edges for v is indicated with $E_{act}(v)$. The set E_{act_i} of edges belonging to $E_{act}(v)$ which form a directed path in the PCG from the entry node to node v, define a *control path* $cpath_i(v) \subset \{C_U \cup C_L \cup F_L\}$. Remember that, if a control-edges path is found starting from a node v_i without incoming control edges, an edge from the entry node to v_i is added in the corresponding control path, since it means that such control dependency is implicit in the PCG. Hence, each control path will start from the entry node. Each node v can be reached by different control paths, whose set is identified by $CP(v)$. It is the case, for example, of node 10 in the PCG of the motivational example shown in Figure 3.8. In the proposed example it is possible to recognize two control paths for node 10:

- $cpath_1(10) = (ENTRY, 2, T) \rightarrow (2, 5, T) \rightarrow (5, 10, F)$,
- $cpath_2(10) = (ENTRY, 2, T) \rightarrow (2, 10, F)$.

It is assumed that each node has at least one control path: if no path is found in the PCG, then $CP(v) = \{(ENTRY, v, T)\}$, remembering again that building the PCG a transitive reduction for edges with source the entry node has been performed. Notice that in the control-path computation, edges outgoing from the entry node are considered as labeled edges $e = (ENTRY, v, L = true)$. Given a control path, it is possible to define the *control path signal* as follows.

Definition 5. *Given a control path $cpath_i(v)$ for a node v, a* control path signal *$cps_i(v)$ is defined as:*

$$cps_i(v) = \prod cpath_i(v),$$

i.e. as the signal obtained applying the join operator to all the edges composing the control path $cps_i(v)$.

Starting from the properties of boolean algebra, the following property for control path signals hold:

Property 2. $$\overline{cps_i(v)} = \sum_{j \in cpath_i(v)} \overline{j}$$

where $\overline{j}$ represents the *negation* of signal $j = (v_1, v_2, L) \in \{C_U \cup C_L \cup F_L\}$, defined as follows:

$$\overline{j} = (v_1, v_2, L') \text{ with } L' \neq L.$$

Edges coming from the entry node are ignored in this computation. It is important to underline that if there exists an edge $j = (v_1, v_2, L)$ in the PCG, then it will not exist an edge $j = (v_1, v_2, L')$ for sure: it is just a notational fiction used to denote that the outcome of condition evaluation in node v_1 is different from L. Considering the previous example, it results for node 10:

- $\overline{cps_1(10)} = (2, 5, F) \vee (5, 10, T)$,
- $\overline{cps_2(10)} = (2, 10, T)$.

3.4.2 Control Dependencies Activation Signal

Each node $v \in V$ can exhibit both control and data dependencies, that contribute together with loop backedges, to form the activation function for node v. From the considerations made in the previous section, it can be inferred that control dependencies must be considered exploiting the concept of control path. Hence, the subsequent definition follows.

Definition 6. *The* control dependencies activation signal $s_{control}(v)$ *is defined as*

$$s_{control}(v) = \sum_{i=0}^{|CP(v)|} cps_i(v),$$

i.e. as the signal obtained applying the or operator to all the control path signals $cps_i(v)$ associated to each control path $cpath_i(v) \in CP(v)$ reaching node v.

The correctness of such definition is justified by the fact that only one signal $cps_i(v)$ can be produced in a given time, since only one control path reaching a specific node may be followed in one of the possible execution flows. An interesting property of control dependencies activation signals is defined as following.

Property 3.
If $|in(v)| = 1$, $in(v) = \{s = (v_s, v) \in \{C_U \cup C_L \cup F_L\}\}$, then $s_{control}(v) = s$.

Justification of this property is trivial: if $|in(v)| = 1$, then $|CP(v)| = 1$, i.e. there exist a unique control path reaching node v. Thus activation of v will depend on its predecessor v_s execution only, and there is no need to consider the entire control path.
The definition given for control dependencies activation signals covers also the case of labeled control flow signals, since these edges has been included in the definition of control path. Despite the concepts of control dependency and control flow edge are very different, the cases concerning the prior aspect will be included in the ones concerning the latter. Hence, with an abuse of notation, signal concerning both this kind of informations will contribute to obtain control dependencies activation signals. The only case remaining not considered is the case of the exit node, that is the only node having unlabeled control flow incoming edges. This case management is closer to the case of data dependencies. For this reason, it will be introduced at the end of the next section.

3.4.3 Data Dependencies Activation Signal

Consider the node 16 in the PCG obtained from the motivational example, shown in Figure 3.8, considering now data dependencies. Node 16 is data dependent on node 14. This means that 16 cannot start before 14 is executed,

but 14 has to be executed only if the condition test for the instruction 13 is true, while 16 has to be executed regardless of the outcome of such condition evaluation. To handle with such situation activation signals for data dependencies must take into account the activation path of the instruction that is source of the dependency. In other words, it must be expressed the concept that a data dependency need to be considered only when the source of such dependency has to be executed, otherwise it must be ignored.
Data dependencies constraining the execution of node v are denoted by edges in the PCG belonging to $D(v) = (D_T \cup D_{AO}) \cap in(v)$. The *data dependencies activation signal* is introduced to determine if data dependencies constraints are satisfied; given that if $d = (v_s, v) \in D(v)$, then v_s denotes the source node of d, a first definition for data dependencies activation signal follows.

Definition 7. *A* data dependencies activation signal $s_{data}(v)$*, for a node* v*, is defined as*

$$s_{data}(v) = \prod_{d \in D(v), i \in \{o,\dots,|CP(v_s)|\}} (\sum(\{d\} \cup \overline{cps_i(v_s)})).$$

i.e. as the signal obtained applying the join operator to the signals obtained by applying the or operator between each data dependency edge d and each control path negated associated to the source of the data dependency v_s*.*

From the Property 2, an alternative formulation for data dependencies activation signals can be obtained as:

$$s_{data}(v) = \prod_{d \in D(v), i \in \{o,\dots,|CP(v_s)|\}} (\sum_{j \in cpath_i(v_s)} (\{d\} \cup \overline{j})).$$

For node 16 in Figure 3.8, having only one data dependency edge d, whose source instruction, 14, has in turn only one control path, it will result:

- $cpath(14) = (ENTRY, 12, T) \to (12, 13, T) \to (13, 14, F)$
- $s_{data}(16) = \sum_{j \in cpath(14)} (\{d\} \cup \overline{j}) = d \vee (12, 13, F) \vee (13, 14, F).$

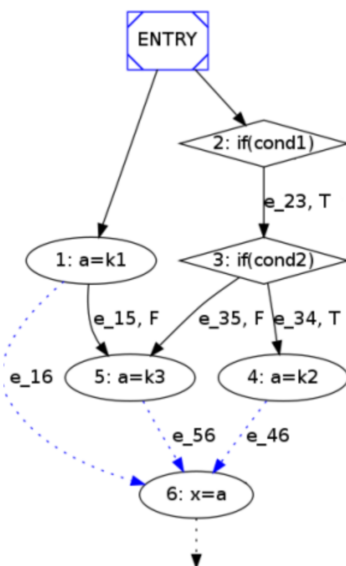

Figure 3.10: *Example showing the need of considering control paths in data dependencies activation signal computation.*

This formulation is able to handle also mutual exclusive dependencies, that will never be simultaneously satisfied. This situation should be clear considering the example provided in Figure 3.10, in which a name has been assigned to each edge just to clarify the formulation:

- node v_6 exhibits data dependencies coming from v_1, v_4 and v_5;
- signals e_46 and e_56 will never be simultaneously produced, since v_4 and v_5 are mutual exclusive operations; node v_1 indeed, will be always be executed, but the generation of signal e_16 will still not allow execution of v_6 until there is evidence that v_4 and v_5 have not to be executed; this situation is managed considering the control paths of nodes v_1, v_4 and v_5, i.e. the nodes on which v_6 is data dependent on;
- denoting with e_ijL the edge $e = (v_i, v_j, L)$ and with v_0 the entry node, it results:
  - $cps(v_1) = e_01$,
  - $cps(v_4) = e_02 \wedge e_23T \wedge e_34T$,

– $cps(v_5) = e_02 \wedge e_23T \wedge e_35F$;

then:

$$\begin{aligned} s_{data}(v_6) &= (e_16 \vee \overline{cps(v_1)}) \wedge (e_46 \vee \overline{cps(v_4)}) \wedge (e_56 \vee \overline{cps(v_5)}) = \\ &= (e_16 \vee \overline{cps(v_1)}) \wedge (e_46 \vee \overline{cps(v_4)}) \wedge (e_56 \vee \overline{cps(v_5)}) = \\ &= e_16 \wedge (e_46 \vee e_23F \vee e_34F) \wedge (e_56 \vee e_23F \vee e_35T) \end{aligned}$$

The proposed general formulation, in many cases will be penalized by the presence of redundant or not necessary operations. Additional optimizations will be introduced in Section 3.10. Even in this case, as it was for control dependencies activation signals, it results that the following property holds.

Property 4.
If $|in(v)| = 1$, $in(v) = \{s = (v_s, v) \in D\}$, then $s_{data}(v) = s$.

The definition introduced for data dependencies activation signals, however, cannot manage the following situation. Consider node 18 in the PCG obtained from the motivational example specification, shown in Figure 3.8. It is data dependent on node 17, hence, according to the formulation given above, its data dependencies activation signal will be obtained as:

$$s_{data}(18) = (17, 18) \vee (\wedge_{cond}(2, T))$$

This means that instruction 18 is activated as soon as the execution of loop1 starts, when the value for the variable res has not been computed yet, or else the instruction 18 is never activated if the loop1 test condition is false since the first iteration, and thus none iteration of the loop has not to be executed. In conclusion, either 18 starts too soon, or it does not start at all. In both cases the behavior is wrong. To handle such situations, in the next section, some considerations about the dominance property will be presented, and the definition of data dependencies activation signals will be slightly modified.
Finally, it remains to manage the case of unlabeled control flow edges, i.e. the case of the exit node. As above mentioned, this case has to be handle

in way similar to data dependency. Considering the PCG in Figure 3.8, it is clear that instruction 18 has to be executed to activate the exit node, and thus terminate the program execution. The exit node has only one predecessor, node 18, that in this case dominates it. In other words, all the paths from entry to exit pass through node 18. There may be, however, situations in which the exit node is not dominated by any of its predecessors. In such situations, the exit node may need only a subset of its predecessors execution to start. Indeed, some of the predecessors can belong to mutual exclusion regions. For this reason, in computing the activation signal for the exit node control paths of the instructions it is data dependent on must be considered. With an abuse of notation, the activation signal for the exit node will be called $s_{control}(EXIT)$. Denoting with v_{fs} the source of the unlabeled control flow edge, it will be defined as:

$$s_{control}(EXIT) = \prod_{f \in F_U, i \in \{o,\ldots,|CP(v_{fs})|\}} (\sum(\{f\} \cup \overline{cps_i(v_{fs})}))$$

Also this definition, however is afflicted by the above described problem, and thus will be slightly modified after the considerations that will be made in the next section.

3.4.4 Preserving the Dominance Property

At this point, all the elements needed to make some considerations about the dominance property have been introduced. This topic will be faced for two reason. First of all, a formal justification will be given for the introduction of a new kind of edges, i.e. the control flow edges; then, the definition of data dependencies activation signals will be modified to manage the situations described in the previous section.

Introducing false-labeled control flow edges, instead of control dependency edges, it has been avoided to incorrectly add a control dependency not present in the original specification.

To clarify this concept, suppose that false-labeled control dependency edges would been added. Then, a control dependency would been added between

source and target of such kind of edges, since a control dependency edge, in a graph representation, is a means to represent control dependencies. Remember that, given two nodes v_i and v_j, v_j is control dependent on v_i if and only if there exists a path from v_i to the exit node not traversing v_j.
Consider, for example, the false-labeled control edge $(2, 17, F)$, that would been added to build the PCG associated to the motivational example specification, shown in Figure 3.8. The consequence of such insertion is that node 17 becomes dummy control dependent on node 2.
Moreover, considering the PDG from which the PCG has been obtained, shown in Figure 3.3, it is clear that node 17 dominates node 18. Indeed, given two nodes v_i and v_j, v_j is dominated by v_i (or equivalently v_i dominates v_j) if and only if all the paths from the entry node to v_j pass through v_i. Such property must hold in PCG too, since dominance relations must be preserved.
However, it is possible to state that, after false-labeled control edges insertion, the PCG would contain a paradox. Indeed, considering again the consequences of edge $(2, 17, F)$ addition, it would result:

1. 17 dominates 18 by construction, i.e. the dominance relation is preserved. Indeed, all the paths from the entry node to node 18 pass through node 17.

2. Since in a graph representation control edges indicate the presence of a control dependency, node 17 become control dependent on node 2.

3. Node 18 is not control dependent on node 2.

4. Since node 17 is control dependent on node 2 and 18 is not control dependent on node 2, then 17 does not dominate 18. Indeed, if 17 is control dependent on node 2, there will be a path from node 2 to the exit node not traversing 17, and if 18 is not control dependent on 2, all the paths from node 2 to the exit node will traverse node 18. This implies the presence of a path from the entry node to 18 not traversing node 17. Obviously, it is an absurd, since point 4. is in contrast with

point 1. Indeed, such a path does not really exists, but its presence is formally inferred by the false control dependency induced by the false-labeled control edge added a posteriori.

For this reason false-labeled *control flow* edges has been added. Indeed, such edges have been added to extract informations about the control flow, hence they can be interpreted only as control flow informations.
Interpreting control flow edges only as control flow informations, however, means modify the definitions provided for data dependencies activation signals and for $s_{control}(EXIT)$. Hence, control flow edges are used in two different ways in such cases and in the case of control dependencies activation signals computation.
In the case of control dependencies, they are used to compute the control paths that can enable an instruction execution. Since an instruction can be activated when it is reached by the control flow, in this case both the informations about control dependencies and control flow are needed.
In the case of data dependencies (and in the case of the exit node), control flow edges are still used to compute control paths, but this time control paths negated are used to understand if the instruction source of the data dependency has to be executed, and as a consequence, if the target of the data dependency has really to wait for its execution. Hence, in this case, the information needed concern only control dependencies, since it is an implicit way to differentiate the instructions according to the possible mutual exclusion region they belong. In other words, in such situations the information about the flow must be excluded from the computation of control paths. According to such considerations, the final definitions for data dependencies activation signals and for $s_{control}(EXIT)$ follow. Denoting with v_s the source node of a data dependency, and with $CP_F(v_s) = \{cpath_F(v_s)\}$ the set of control paths for the node v_s containing at least a control flow edge belonging to the set F_L, it is possible to define the set of the control paths for the node v_s which not contain any control flow edge as:

$$CP'(v_s) = CP(v_s) \setminus CP_F(v_s)$$

Definition 8. *A* data dependencies activation signal $s_{data}(v)$*, for a node* v*, is defined as*

$$s_{data}(v) = \prod_{d \in D(v), i \in \{o,\ldots,|CP'(v_s)|\}} (\sum(\{d\} \cup \overline{cps_i(v_s)})).$$

i.e. as the signal obtained applying the join operator to the signals obtained by applying the or operator between each data dependency edge d and each control path negated associated to the source of the data dependency v_s*, not containing any control flow edge.*

Definition 9. *The control dependencies activation signal for the exit node is define as*

$$s_{control}(EXIT) = \prod_{f \in F_U, i \in \{o,\ldots,|CP'(v_{fs})|\}} (\sum(\{f\} \cup \overline{cps_i(v_{fs})}))$$

This formulation is general and ensures the correctness of the activation function for any kind of dependencies.

3.4.5 Back Edges Activation Signal

For each $loop_k$, each node belonging to $first(loop_k)$ has incoming backedges, introduced to manage subsequent loop iterations.

Definition 10. *The* back edges activation signal $s_{be}(v)$ *for a node* $v \in first(loop_k)$ *is defined as*

$$s_{be}(v) = \prod C_{be}(v)$$

where $C_{be}(v)$ *denotes the set* $C_{BE} \cap in(v)$ *of incoming back edges.*

This formulation is justified by the fact that operation v at iteration $i + 1$ can be executed only if all operations in the previous iteration i have already been executed. Anyway, especially in the case the front-end does not perform loops analysis and optimizations, such as loop invariants detection or loop hoisting, some instruction belonging to $i + 1$ could be executable even if

not all the instructions of the previous iteration have been executed. Thus, a proper analysis of loop-carried dependencies will detect these situations, thus reducing the cardinality of the set $C_{be}(v)$ and then simplifying back edges activation signals computation.

3.4.6 Activation Function Formulation

Once introduced control dependencies, data dependencies and back edges activation signals, it is now possible to determine a proper formulation for the activation function $AF(v)$:

$$AF(v) = (s_{control}(v) \wedge s_{data}(v)) \vee s_{be}(v).$$

This formulation is justified this way:

- data dependencies and control dependencies must be both satisfied to enable operation v, thus a join operation between their associated activation signals is needed;
- only one between control/data dependencies activation signals and back edges activation signal is needed to start a loop execution, so an or operator between them is needed. This should be clear considering the instruction 12 in Figure 3.8: when loop3 must be executed the first time, the back edge activation signal has surely not been yet produced, since the instruction 15 has not been yet executed. However, when dependencies are satisfied, i.e. after instruction 11 execution, the loop must be executed the first time, consuming the corresponding signal. Then, when the back edge signal will be produced, there will no longer be the already satisfied dependency signals, since they will be already consumed. Hence, they must not be considered for a loop to re-start. However, this is not a limitation since there is no need for subsequent iterations to care about dependencies already satisfied when the loop is first entered.

NODE	*ACTIVATION FUNCTION*
1	$AF(1) = s_c(1) = TRUE^a$
2	$AF(2) = s_d(2) \vee s_{be}(2) = (1,2) \vee (10,2)$
3	$AF(3) = s_c(3) = \wedge_c(2,T)$
4	$AF(4) = s_c(4) \wedge s_d(4) = (\wedge_c(2,T)) \wedge (1,4)$
5	$AF(5) = (s_c(5) \wedge s_d(5)) \vee s_{be}(5) = \{(\wedge_c(2,T)) \wedge [(3,5) \vee (\wedge_c(2,F))]\} \vee [(9,5) \wedge (8,5)]$
6	$AF(6) = s_c(6) \wedge s_d(6) = (\wedge_c(2,T)) \wedge (\wedge_c(5,T)) \wedge [(3,6) \vee (\wedge_c(2,F))]$
7	$AF(7) = s_c(7) \wedge s_d(7) = (\wedge_c(2,T)) \wedge (\wedge_c(5,T)) \wedge [(6,7) \vee (\wedge_c(2,F)) \vee (\wedge_c(5,F))]$
8	$AF(8) = s_c(8) \wedge s_d(8) = (\wedge_c(2,T)) \wedge (\wedge_c(5,T)) \wedge (1,8) \wedge [(7,8) \vee (\wedge_c(2,F)) \vee (\wedge_c(5,F))]$
9	$AF(9) = s_c(9) \wedge s_d(9) =$
	$= (\wedge_c(2,T)) \wedge (\wedge_c(5,T)) \wedge [(3,9) \vee (\wedge_c(2,F))] \wedge [(6,9) \vee (\wedge_c(2,F)) \vee (\wedge_c(5,F))]$
10	$AF(10) = s_c(10) \wedge s_d(10) =$
	$= \{[(\wedge_c(2,T) \wedge (\wedge_c(5,F))] \vee (\wedge_c(2,T))]\} \wedge [(4,10) \vee (\wedge_c(2,F))] \wedge [(8,10) \vee (\wedge_c(2,F)) \vee (\wedge_c(5,F))] \wedge (1,10)$
11	$AF(11) = s_c(11) = TRUE^a$
12	$AF(12) = s_d(12) \vee s_{be}(12) = (11,12) \vee (15,12)$
13	$AF(13) = s_c(13) \wedge s_d(13) = (\wedge_c(12,T)) \wedge (11,13)$
14	$AF(14) = s_c(14) \wedge s_d(14) = (\wedge_c(12,T)) \wedge (\wedge_c(13,T)) \wedge (11,14)$
15	$AF(15) = s_c(15) \wedge s_d(15) = (\wedge_c(12,T)) \wedge (11,15) \wedge [(14,15) \vee (\wedge_c(12,F)) \vee (\wedge_c(13,F))]$
16	$AF(16) = s_c(16) \wedge s_d(16) = (\wedge_c(12,F)) \wedge [(14,16) \vee (\wedge_c(12,F)) \vee (\wedge_c(13,F))]$
17	$AF(17) = s_c(17) \wedge s_d(17) = (\wedge_c(2,F)) \wedge [(7,17) \vee (\wedge_c(2,F)) \vee (\wedge_c(5,F))] \wedge (16,17)$
18	$AF(18) = s_d(18) = s_d(18) = (17,18)$
EXIT	$AF(EXIT) = (18.EXIT)$

Table 3.2: *Activation Functions in extended form for the Operations in the PCG obtained from the motivational example.*

[a]The operations with a TRUE value for the activation function can start their execution when the program starts to run.

In conclusion, considering once again the PCG obtained from the motivational example, shown in Figure 3.8, the activation functions for each node are shown in Table 3.2. Notice that control edges from entry have been omitted, since the program will always start. Indeed, a control edge $(ENTRY, v_i, F)$ indicate the condition in which the program does not start, while a control edge $(ENTRY, v_i, T)$ indicate a condition in which the program start. Since it is known that the program starts such two kind of edges can be both omitted. For brevity, the set $s_{control}$ will be denoted as s_c, the set s_{data} will be denoted as s_d, and the unary operator $\wedge_{cond}(s)$ will be denoted as $\wedge_c(s)$.
Finally, as will be described in the next of this work, such formulation for the activation function can be easily optimized in different ways. For example, it can be reduced by means of the Boolean Algebra rules, since it can be viewed as a logic function.

3.5 Controller Synthesis Process Overview

One of the main objective of the analysis phase has been the construction of a Parallel Controller Graph representing the specification to be implemented: starting from the PCG, each instruction v of the source code has been an activation function $AF(v)$ associated with, which indicates if v's dependencies have been satisfied. In this case, execution of v can start. Given this scenario, a controller structure design is proposed in order to exploit the analysis phase results. The most natural way to obtain the desired behavior, is to associate to each instruction a control module which manages its execution, exploiting the concept of activation function. Activation functions should thus be hardwire implemented, defining activation function modules. Control modules will then receive the signals produced by the associated activation function module as input; when the signal is turned high than execution can start. Moreover, once the instruction is executed, a signal must be sent to the other activation function modules associated to the dependent instructions. The resulting controller design, from a behavioral point of view, consists then in a

set of pairs control/activation function module associated to each instruction (i.e. nodes in the PCG). In following sections, the synthesis process of such a controller is presented, starting from the definition of the target architectural model (Section 3.6). Then a proper design for the control (Section 3.8) and activation function (Section 3.7) modules is proposed, and finally, in Section 3.9 the definition of a synthesis step to be included in the high level synthesis flow is presented.

3.6 Architectural Model

The proposed architectural model is composed of a controller and a datapath, as usual. The controller is not a centralized FSM-based one, as in the state-of-the-art FSMD model, indeed it follows the design concepts previously presented.

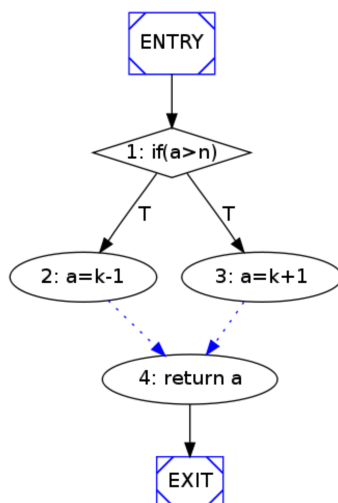

Figure 3.11: *Example PCG.*

Starting from the example PCG in Figure 3.11, Figure 3.12 shows such an architectural model. Each node in the PCG has a control module associated to, which manages its execution. Such control modules:

- take as input signals produced by the associated activation function

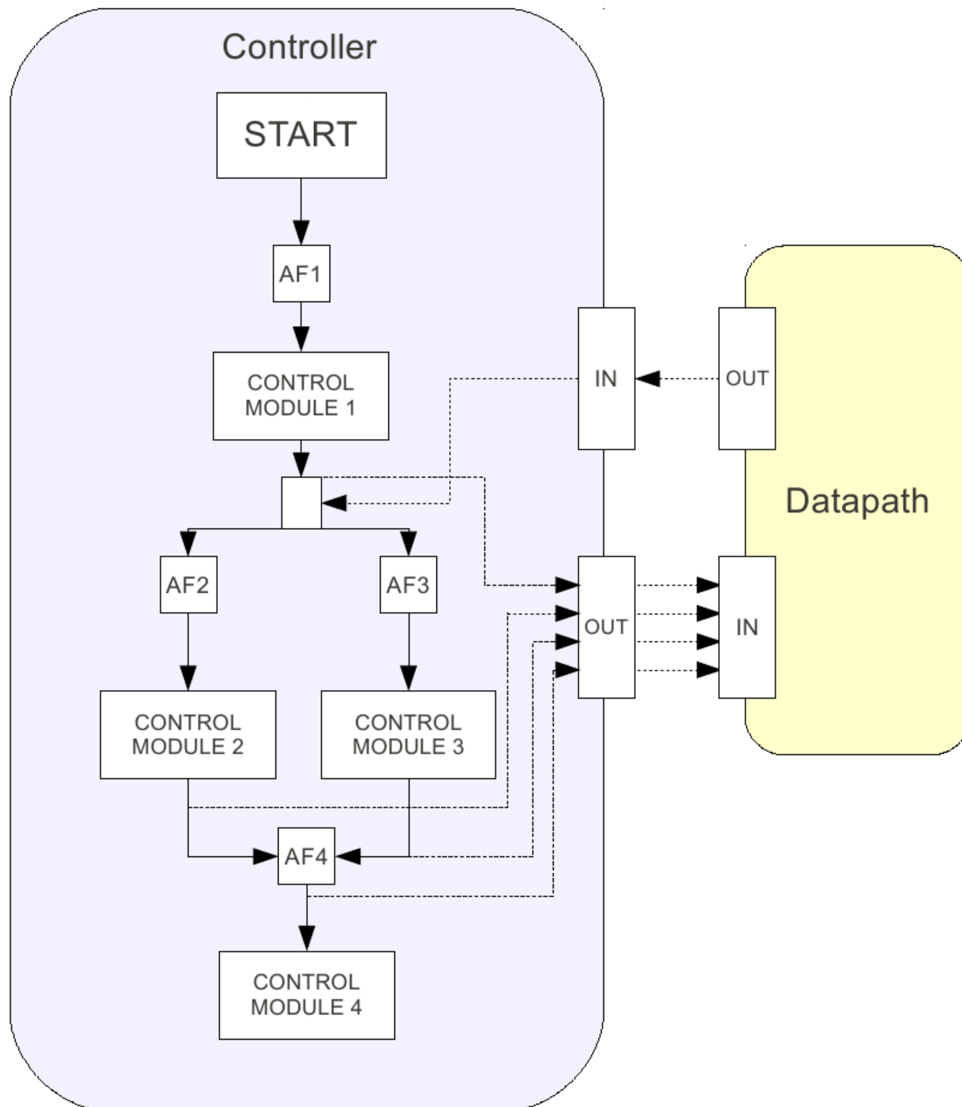

Figure 3.12: *Synthesized architecture scheme.*

modules;

- when the activation function module fires, than it means that the execution can start. In this case they produce control signals to be sent to the datapath through the controller out ports, activating the corresponding functional units.

Communication occurs even from the datapath to the controller, as in the case of activation functions that must care of the result of a conditional instruction. For example, node 1 in Figure 3.11 is a diramation node: activation functions of instructions 2 and 3, depending on the result of condition evaluation, receive this result from the datapath, as in Figure 3.12.

3.7 Activation Function Module

In Section 3.4 the activation function for a node v has been defined as:

$$AF(v) = (s_{control}(v) \wedge s_{data}(v)) \vee s_{be}(v)$$

where

- $s_{control}(v) = \sum_{i=0}^{|CP(v)|} cps_i(v)$,
- $s_{data}(v) = \prod_{d \in D(v), i \in \{o, \ldots, |CP(v_s)|\}} (\sum(\{d\} \cup \overline{cps_i(v_s)}))$,
- $s_{be}(v) = \prod C_{be}(v)$.

In order to obtain a proper implementation of the activation function module, each introduced operator involved in the activation function formulation, i.e. join operator $\wedge$, or $\vee$ operator and conditional join operator $\wedge_{cond}$ have been designed, as described in the following. As a result, the activation function module will consist in a set of interconnected modules, each implementing one of the above mentioned operators.

3.7.1 Join Module

The *join module* implements the join operator previously defined. It is a sequential module taking as inputs the signals to be joined, in addition to clock and reset signals. In its basic implementation, it produces as output a high valued signal iff all the inputs signals have been received, i.e. they have assumed a high value. Since input signals may be received in different clock cycles, when a input turns high the join module must store this information, i.e. the corresponding signal reception. Thus, the join module is designed as a finite state machine. In Figure 3.13 the FSM representing a 3-inputs join

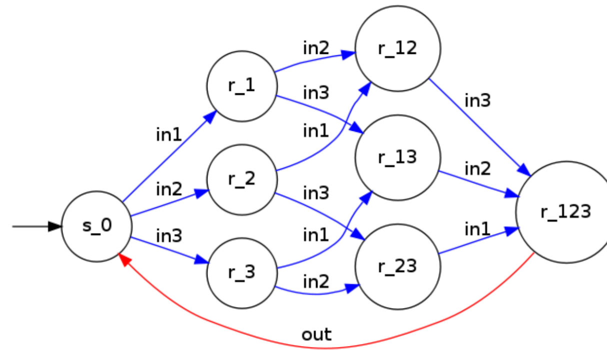

Figure 3.13: *Finite state machine representing 3-inputs join module*

module is shown:

- s_0 represents the initial state, in which no input signal has been received;
- states r_i, r_{ij}, r_{ijk} represent the occurred reception of signals i, j, k;
- starting from s_0, if input i turns high then a transition to state r_i is performed; a transition due to signal i reception is represented by edges in_i, starting from the current state and going to the corresponding next;
- if the current state is r_i, if the signal i turns high again, no transition is performed; instead if the signal $j \neq i$ turns high, a transition to state r_{ij} occurs. The same happens if current state is r_{ij};

- for a 3-input join module, the state s_{123} denotes that all input signals have been received: if the state s_{123} is reached, than the output signal must be produced, and a transition made to the initial state s_0.

In the proposed example, signals are received one at a time, but this is only to make explanation more clear: in the real implementation more than one signal can occur simultaneously; for example, starting from the initial state s_0, if signals s_1 and s_2 turn high at the same time than transition is made to state r_{12}. Notice that the proposed FSM model does not have a final state: when state s_{123} is reached and output produced, the FSM returns in the initial state. This way the join module is constantly monitoring input signals and in case, it turns high the output signal again. This feature is surely needed to state if a loop body must be executed again or not. This

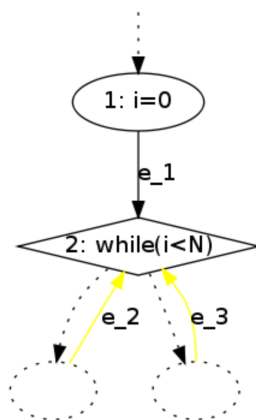

Figure 3.14: *Portion of a PDG showing incoming edges of a loop condition node*

issue will result more clear considering the example proposed in Figure 3.14. Activation function for vertex v_2 is given by

$$f(v_2) = e_1 \vee (e_2 \wedge e_3) \ ,$$

thus a join between signals e_2 and e_3 is needed, and this operation must be computed at each iteration of the loop. It is important to notice that in some situations, in the presence of loop constructs, signals to be joined could be produced once for different subsequent iterations of the loop; then the join

module, as just proposed, will turn high the output signal only at the first iteration, showing an erroneous behavior. An example is provided by Figure

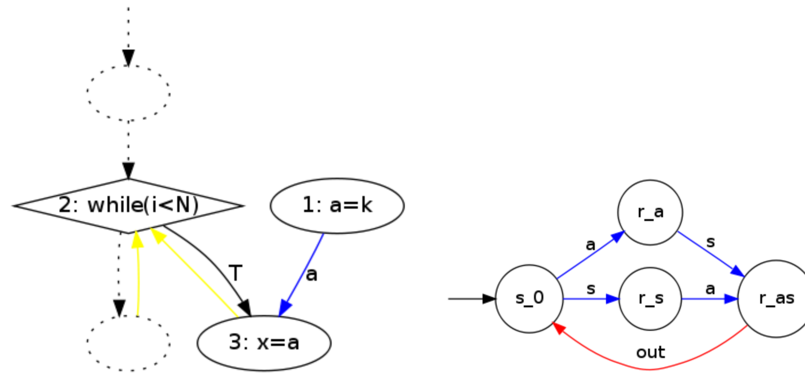

Figure 3.15: *Example of inadequacy of standard join module*

3.15, where the activation function for the node v_3 is given by

$$f(v_3) = \wedge_{cond}(e_{23}) \wedge e_{13} \text{ ,where } e_{23} = (v_2, v_3, T) \text{ and } e_{13} = (v_1, v_3, a).$$

Writing $s = \wedge_{cond}(e_{23})$ and $a = e_{13}$, s is high-valued only when the loop is executed the first time: at the second iteration, after the reception of signal a, the FSM representing the join module reaches state r_a; since it surely will not receive signal s again, no other transition will occur, and thus the execution flow halts. To handle with this problem, a modified version of join module is introduced. Its associated FSM, for the previous example, is shown in Figure

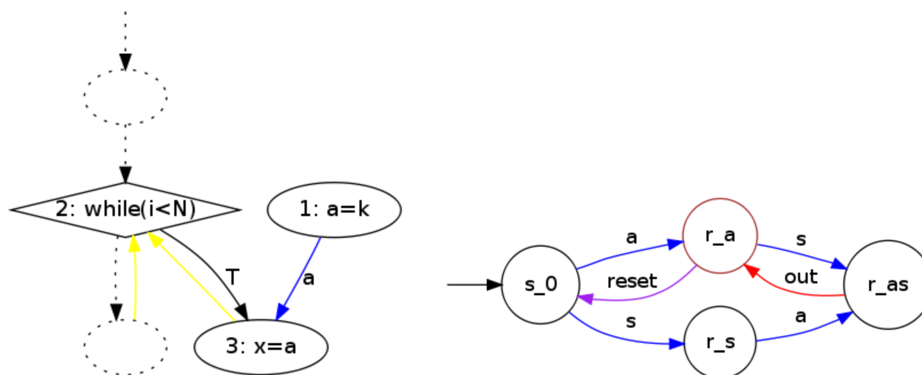

Figure 3.16: *Modified 2-input join module*

3.16: at the first iteration, when both a and s has been received, the FSM goes to state r_{as}, it produces the output signal and than returns in state r_a instead of state s_0. This way, for the subsequent iterations, signal a will be assumed as already received. The FSM returns to the initial state s_0 through the reception of an explicit reset signal, starting from state r_a. Signals, as a in the example, that requires the modified version of the join module, are identified in the PCG building process, or in a preliminary analysis phase. Notice that some front-end optimization, as loop hoisting, can avoid many of these situations to occur.

3.7.2 Or Module

The *or module* produces an high valued signal if any of the signals taken as input turns high. Since there's no need for this module to store informations about signals already received, the natural way to implement it is using a simple *or* gate.

3.7.3 Condition Join Module

Condition join module (Figure 3.17) implements $\wedge_{cond}$ operator; it is remarked that, given that node v_1 is a diramation node and $s = (v_1, v_2, L = cond_i)$, $\wedge_{cond}(s)$ produces a signal iff both

- v_1 has already been executed, generating the signal s_{v_1},
- condition evaluation in v_1 has $cond_i$ as outcome.

Even if $\wedge_{cond}$ operator has been defined as an unary one, from a behavioral point of view it should be viewed as a join between the signal s_{v_1} taken as input, and the signal *cond* representing the result of vertex v_1 condition evaluation. This is implemented through condition join module: it takes as input signal s and signal *cond*, that must be forwarded from the datapath, more precisely from the functional unit, e.g. a comparator, that computes condition evaluation. If diramation node v_1 has N branches, than condition

join module has N output ports: the signal resulting from $\wedge_{cond}(s)$, $s = (v_1, v_2, L = cond_i)$ with $i = 1, 2, ..., N$, is given by the i-th out port of the module. Thus a single condition join module can implement at least N

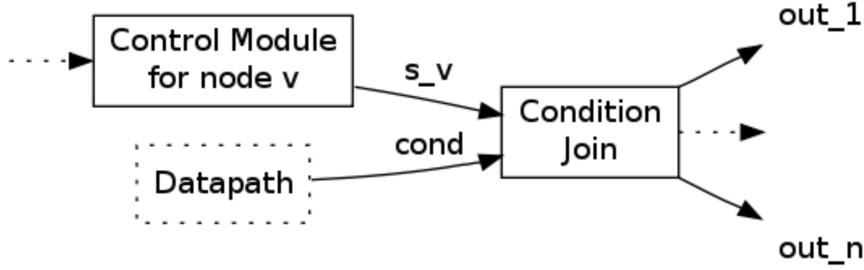

Figure 3.17: *Condition join module schematic representation.*

different $\wedge_{cond}$ operations; their precise number N' is given by

$$N' = |C_L(v_1)|, \text{ with}$$
$$C_L(v_1) = out(v_1) \cap C_L = \{e = (v_1, t, L = cond_i), t \in V, i \in \{1, ..., N\}\}.$$

It is then implied that condition join modules are associated with dirama-tion nodes in the PCG, and that all operations $\wedge_{cond}(e), e \in C_L(V)$ are implemented sharing the same conditional join module. Since signals s and *cond* are produced simultaneously, there is no need of storage elements in the module, whose physical implementation requires only combinational circuitry.

3.8 Control Module

The control module has the role of managing the execution of an instruction. It takes as input the signals produced by the associated AF module, and when execution can start it activates the functional unit on which the instruction is bound sending control signals to the datapath. When the execution is performed, it notifies this to the AF modules associated to dependent instructions. Anyway, the AF signal reception itself does not always lead to

the immediate activation of the instruction: if it is mapped on a shared resource, concurrency must be handled. For this reason the control module has been designed as a *control element* receiving the activation function signals, connected to a priorities manager, introduce to manage concurrency.

3.8.1 Control Element

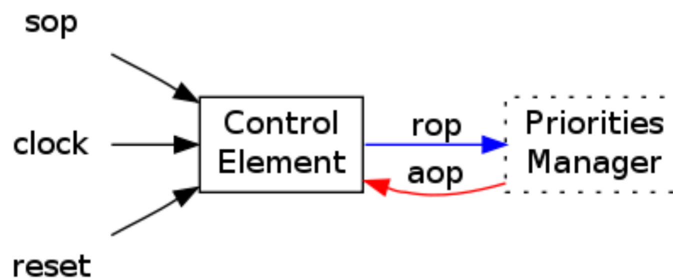

Figure 3.18: *Control Element module schematic representation.*

The control element schematic representation is shown in Figure 3.18:

- clock and reset denotes clock and reset signals taken as input by the module;
- input signal *sop* assumes an high value when the operation associated to the control element can start its execution, i.e. it is turned high when the activation function for node v turns high;
- output signal *rop*, is turned high when the execution of the operation can start (which means that *sop* signal has already been high-valued and all dependencies satisfied); it can be viewed as a request to the priorities manager, that will establish if the functional unit on which the given operation is associated to is available;
- input signal *aop* represents the answer of the priorities manager to the control element's request: it assumes a high value if the functional unit

on wich the operation is bound is available; together with the *aop* signal reception, the associated operation is executed.

Notice that if the functional unit on which the operation is bound is not shared among different operations, than there is no need for the control element to send (and to receive) any signal from the priorities manager. In absence of concurrency over the functional units, the execution of an operation can start as soon as the control element receives the sop signal. The behavior of the control element module can be described by a finite state

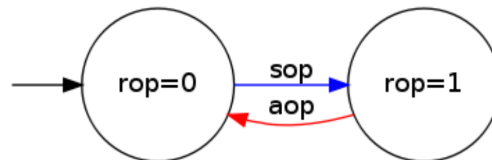

Figure 3.19: *Control Element module FSM representation.*

machine, presented in Figure 3.19 :

- in the initial state no request is forwarded since the dependency constraints are not yet satisfied;
- when the *sop* signal is received, then execution can start and a request is sent to the priorities manager, bringing the machine to the state labeled as "$rop = 1$";
- while *aop* signal has not yet come, no transition is made: *rop* signal maintains its high value, since request for execution must be sent again to the priorities manager until the requested functional unit becomes available and thus the *aop* signal is received;
- when *aop* signal turns high, the operation is executed and the FSM returns to the initial state. If indeed the control element is associated to an operation belonging to a loop, then the operation can have to be executed more than once, and the control element must manage this situation. For this reason, even in this case, there is not a final state.

3.8.2 Priorities Manager

During the binding phase each instruction is associated with a particular instance of a functional unit able to execute it. If the same module is bound to more than one instruction, than it is shared among them. It is clear that if in a given cycle step multiple instructions sharing the same functional unit are ready to be executed (their associated control elements have received the *sop* signal), a choice between them must be made. This task is performed by the priority manager, introduced to handle with concurrency. The choice is made according to operation priorities, that are obtained starting from the instruction scheduling. Given a set V_M of operations that share the same functional unit M, an instructions schedule S induces a binary relation $\leq_p$ over V_M such that:

$$v_1 <_p v_2 \text{ iff } c_s(v_1) > c_s(v_2) \text{ with } v_1, v_2 \in V_M,$$

where $c_s(v)$ is the cycle step in which instruction v is scheduled according to S. For each feasible scheduling it results $c_s(v_1) \neq c_s(v_2)$, since two instructions scheduled in the same cycle stop cannot be bound to the same functional unit. Thus the schedule S defines a total ordering over the set V_M: the priority $pt(v)$ of instruction v is then defined as its position in such a ordering. It follows that instruction scheduled in earlier cycle step has

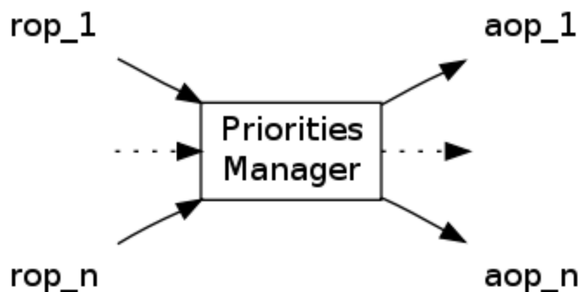

Figure 3.20: *Priorities manager schematic representation.*

greater priority, and vice versa. Therefore the priority manager, given two operations v_1 and v_2 belonging to V_M both ready to be executed, will enable

the execution start of the one with greater priority. The priority manager module can be schematically represented as in Figure 3.20. If it is associated with a functional unit M such that $|V_M| = n$, than it has n inputs and n outputs, each corresponding to a different operation i. When the i-th input rop turns high it means that the control element associated to the i-th operation $v_i \in V_M$ has required the instruction execution. Indicating with $ROP(t)$ the set of rop signals received at control step t, the output signal aop_i activating operation i, is generated if:

- $rop_i \in ROP(t)$;
- $i = argmax_j(pt(v_j))$.

Obviously at each cycle step t, only one of the n outputs is turned high. To clarify how control elements and priorities managers actually interact,

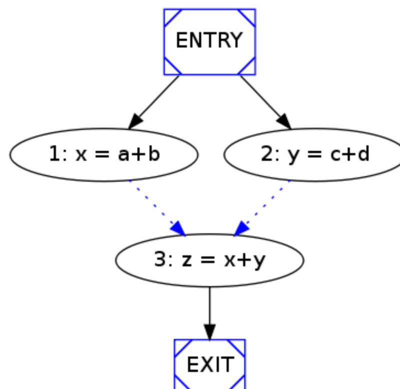

Figure 3.21: *Example parallel controller graph.*

consider the parallel controller graph in Figure 3.21. Assuming only one adder available, the three instructions must be scheduled in subsequent cycle steps, and bound to the unique functional unit. For example, a feasible scheduling according to the dependency constraints associates v_1 to cycle step c, v_2 to $c + 1$ and v_3 to $c + 3$. Hence it results $v_3 <_p v_2 <_p v_1$. Figure 3.22 shows how control elements and the unique priorities manager behave at the beginning of code execution, e.g. at cycle step $t = 1$:

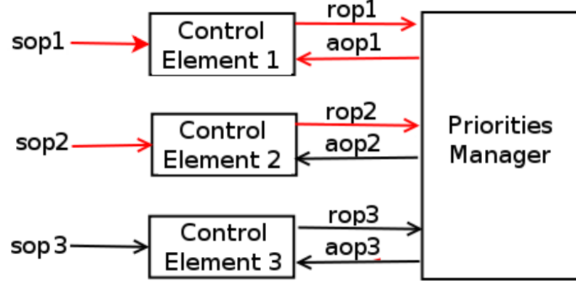

Figure 3.22: *Interaction between priorities manager and control elements at control step t=1; red edges denotes high-valued signals.*

- v_1 and v_2 are dependent only to the entry point, so they can be both immediately executed; the start module produces done signals that satisfy their dependencies, than sop_1 and sop_2 are high-valued;
- sop_2 has not yet been generated, since v_3 depends on both v_1 and v_2;
- received the corresponding sop signals, control element 1 and control element 2 request the activation of v_1 and v_2, turning high $rop1$ and rop_2;
- the priorities manager receives in the same cycle step more than one activation requests, thus a choice must be made. It results $ROP(t = 1) = \{rop_1, rop_2\}$; since $v_1 >_p v_2$ or alternatively $pt(v_1) > pt(v_2)$, then operation v_1 is selected and aop_1 turned high.
- in the following control step $t = 2$, it will be
  $sop_1 = sop_3 = 0, sop_2 = 1 \rightarrow$
  $\rightarrow rop_1 = rop_3 = 0, rop_2 = 1 \rightarrow ROP(t) = \{rop_2\} \rightarrow$
  $\rightarrow aop_1 = aop_3 = 0, aop_2 = 1;$
- in control step $t = 3$, it will be
  $sop_1 = sop_2 = 0, sop_3 = 1 \rightarrow$
  $\rightarrow rop_1 = rop_2 = 0, rop_3 = 1 \rightarrow ROP(t) = \{rop_3\} \rightarrow$
  $\rightarrow aop_1 = aop_2 = 0, aop_3 = 1.$

The resulting schematic representation of the control module, is shown in Figure 3.23.

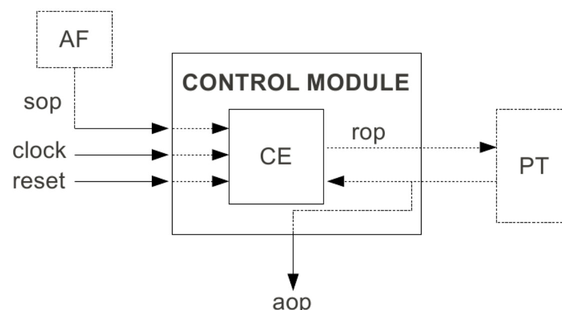

Figure 3.23: *Control module schematic representation.*

3.9 Controller Synthesis Flow

The previously defined controller architecture design, must be automatically obtained starting from the specification through a HLS flow. As a result, the general Controller Synthesis step, presented in Section 1.6, has been modified as shown in Figure 3.24. The introduced controller synthesis flow is composed of a set of sub-task: the first steps (1. and 2.) are not dependent from the performed analysis phase; in step 3. for each node in the parallel controller graph control modules, activation function modules and in the case of diramation nodes conditional join modules are allocated and interconnected. Finally the interconnections, where not, are defined. Notice that even the resource library has to be extended, including the introduced modules.

3.9.1 Common Ports Instantiation and Interfaces Creation

At the beginning of the synthesis phase, the controller is just an empty black box. As a first step common ports are added to the design; such ports include:

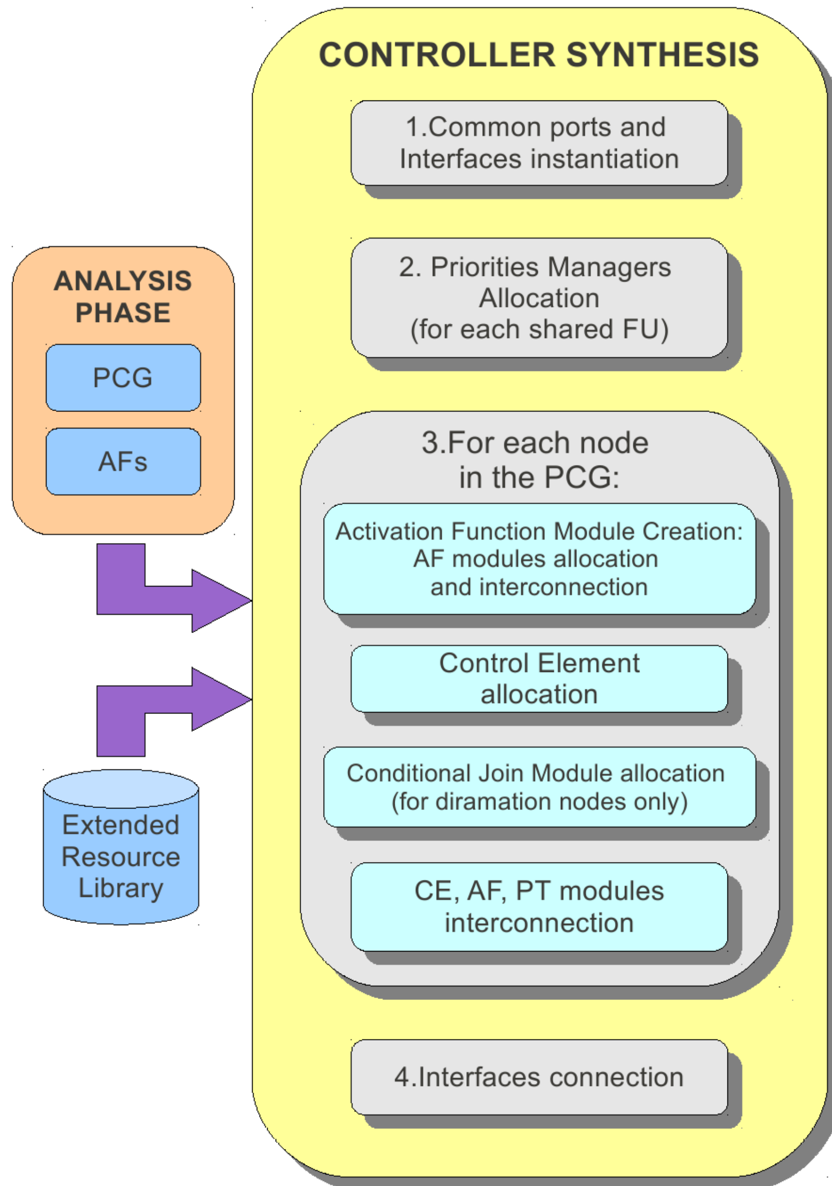

Figure 3.24: *Controller synthesis step added in the HLS flow.*

- clock signal port,
- reset signal port,
- start signal port,
- done signal port.

The start signal is introduced to make the implemented subprogram's execution start. In the parallel controller graph representation, the start signal is assumed to be produced by the entry node, representing the unique entry point of the source program. In the same way, the done signal denotes that execution has been performed, and in the PCG this is denoted by the activation of the exit node. The input/output interfaces with the datapath are then instantiated. Input ports are used to receive signals from the datapath, output ones to send control signals in order to manage the datapath itself.

3.9.2 Priorities Managers Allocation

Once the functional unit binding and instruction scheduling task have been performed, enough information is available to correctly perform priority managers allocation:

- the FU binding provides information about the priorities manager to be instantiated, identifying the functional units shared among multiple; to each of them is associated a priorities manager;
- the instruction scheduling provides informations to properly construct the priority ordering, which concurrency management performed by the priorities manager is based on.

3.9.3 Activation Function Modules, Control Modules and Condition Join Modules Creation

In this synthesis step the results obtained in the analysis phase are exploited. For each node in the PCG, except the entry node, each of the following tasks

is performed:

- Activation function module creation: since the AF module is a compound one, each module composing it must be instantiated, and properly connected; connection of the inputs to the corresponding ports is also performed;
- Control module creation: for each node, the corresponding control element is instantiated and connected to the associated functional unit module, and, if needed, to a priorities manager;
- if the considered node is a diramation node, then a condition join module is also allocated. The input signal, representing the outcome of the condition evaluation, is taken through the controller input ports.

The entry node is treated differently with respect to the other nodes in the PCG, since it does not represent an operation to be performed, but it simply denotes the entry point of the implemented subprogram.

3.9.4 Interfaces Connection

Interconnections are defined from the allocated control modules to the controller output ports, in order to allow the sending of control signals to the datapath. Notice that these phase may be embedded in the control modules creation step.

3.10 Optimizations

One of the most important factors impacting on the area overhead in the proposed architectural model is the need of activation function (AF) modules allocation. Each AF module is composed of a set of modules, each implementing the introduced operations. Since the proposed formulation of the activation function has the aim of being as most general as possible, in many cases activation functions include redundant operations that increase

the number of physical resources to implement them. Thus, in order to improve the performance in terms of area, some optimization algorithms are proposed to:

- reduce the number of modules needed to implement the AFs;
- reduce the number of inputs of the allocated modules, since, considering the proposed implementations, described in Section 3.7, modules area occupancy increases according with the number of inputs.

The two following approaches have been identified:

- *reduction of the number of edges in the PCG*; this is obtained through SSA form transformation of the source code;
- *activation functions simplification*; this is obtained exploiting the boolean algebra properties.

In the following such optimization approaches will be presented.
Remember that in the proposed formulation there exist two kinds of edges, since signals can come from either a control module or a conditional join module. In the former case they are expressed as (v_i, v_j), where v_i and v_j are operations, while in the latter case they are expressed as (v_i, L), where v_i is a conditional operation and L is the label indicating the result of the associated condition evaluation. However, using the above described formulation for the activation function, edges (and thus signals) of both kinds can be *homogeneously treated as literals*, with the only difference that their high/low value has a different meaning. When a signal (v_i, v_j), coming from a control module, assumes an high-value it means that the corresponding operation v_i has been executed, while when it assumes a low value, v_i has not been executed yet. Conversely, when a signal (v_i, L), coming from a conditional join module, assumes an high value it means that the corresponding conditional operation v_i has been executed giving L as result of the condition evaluation, while when it assumes a low value, it may be because either v_i has not been executed yet, or it has been executed giving $\overline{L}$ as result of the condition evaluation.

3.10.1 Static Single Assignment Form Transformation

The transformation of the source code into static single assignment (SSA) form, in many cases could drastically reduce the number of edges in the PCG (leaving unchanged the number of nodes), thus simplifying the computation of activation functions.

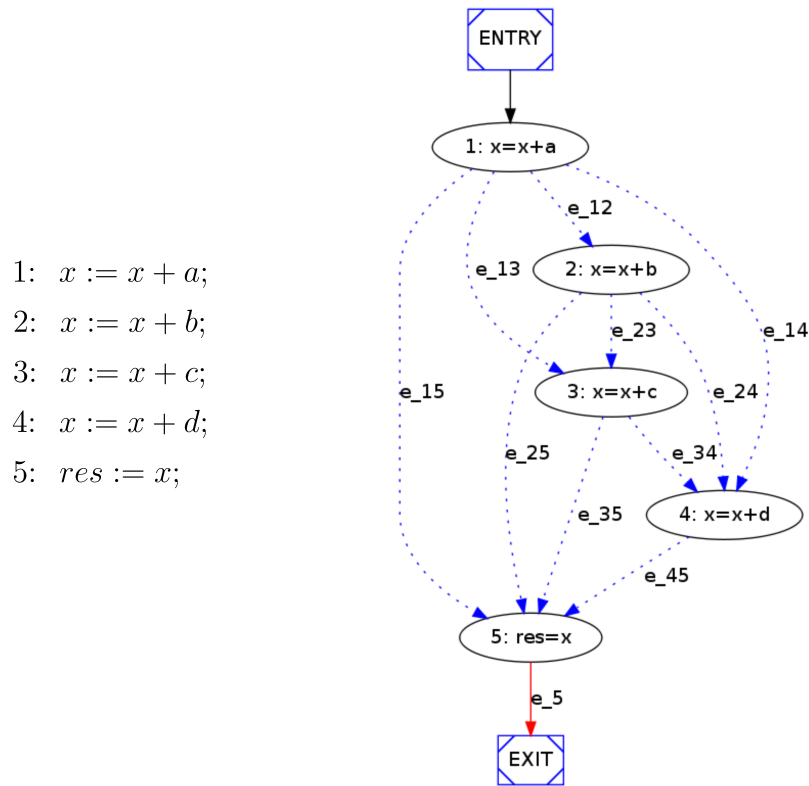

Figure 3.25: *Example code and corresponding PCG.*

Figure 3.25 shows a simple purely data-flow subprogram, not in SSA form: the variable x has several definition points, in which it is also used; as a result, it is possible to recognize in the corresponding PCG a chain of data dependencies, most denoting irrelevant relationships between instructions. Table 3.3 reports the resulting activation functions for each node of the PCG. To implement such AFs, four activation function modules must be allocated, each composed of a join module having from two to four inputs.

If indeed the source code is translated into the SSA form, as shown in Figure 3.26, the set of data dependencies becomes greatly reduced, leading to simpler formulations of activation functions. Referring to Table 3.4, activation functions for the code translated in SSA form shows no need for the AF modules to be allocated. In the case of complex programs, this can result in a great reduction in terms of area overhead.

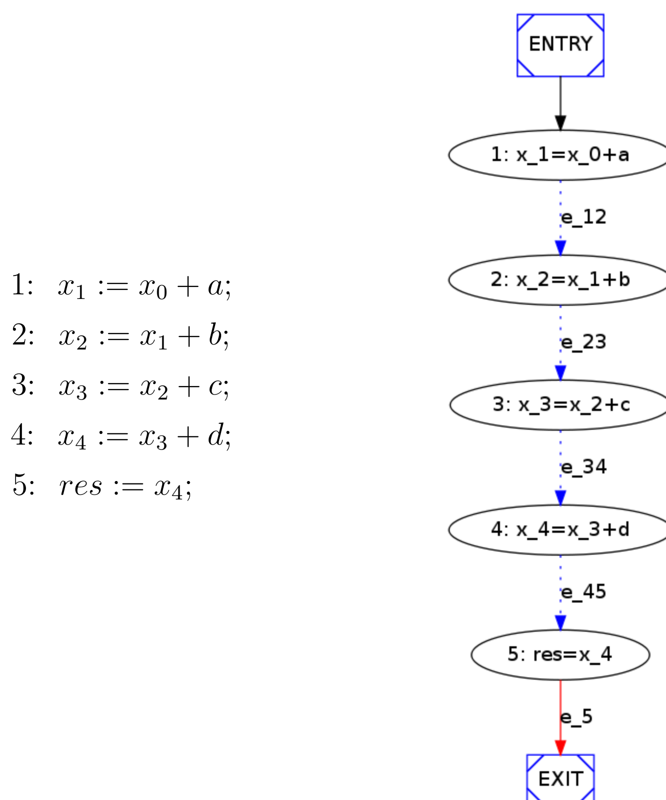

Figure 3.26: *SSA translation of previous example's code and corresponding PCG.*

3.10.2 Activation Functions Simplifications

As the introduced operators used to define activation functions are strictly connected to boolean operators, some properties of Boole's algebra holds for join, or and conditional join operators too. Thus, activation function can be simplified exploiting these properties.

NODE	*ACTIVATION FUNCTION*
1	$AF(1) = s_c(1) = TRUE$
2	$AF(2) = s_d(2) = e_{12}$
3	$AF(3) = s_d(3) = e_{23} \wedge e_{13}$
4	$AF(4) = s_d(4) = e_{34} \wedge e_{14} \wedge e_{24}$
5	$AF(5) = s_d(5) = e_{45} \wedge e_{15} \wedge e_{25} \wedge e_{35}$
EXIT	$AF(EXIT) = e_5$

Table 3.3: *Activation Functions in extended form for the nodes of the PCG in Figure 3.25.*

NODE	*ACTIVATION FUNCTION*
1	$AF(1) = s_c(1) = TRUE$
2	$AF(2) = s_d(2) = e_{12}$
3	$AF(3) = s_d(3) = e_{23}$
4	$AF(4) = s_d(4) = e_{34}$
5	$AF(5) = s_d(5) = e_{45}$
EXIT	$AF(EXIT) = e_5$

Table 3.4: *Activation Functions in extended form for the nodes of the PCG in Figure 3.26.*

Algebraic Simplification

Algebraic simplification of activation functions can be achieved exploiting the following properties of boolean algebra:

1: $a \vee 1 = 1$
2: $a \wedge 0 = 0$
3: $a \vee a = a$
4: $a \wedge a = a$
5: $a \vee \overline{a} = 1$
6: $a \wedge \overline{a} = 1$
7: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
8: $a \vee (a \wedge b) = a$
9: $a \wedge (a \vee b) = a$
10: $a \vee (\overline{a} \wedge b) = a \vee b$
11: $a \wedge (\overline{a} \vee b) = a \wedge b$

Table 3.5: *Common properties of boolean algebra useful to simplify activation functions.*

As an example consider the activation functions of the motivational example listed in Table 3.2; for node 7 it results:

$$
\begin{aligned}
AF(7) &= s_c(7) \wedge s_d(7) = \\
&= (\wedge_c(2,T)) \wedge (\wedge_c(5,T)) \wedge [(6,7) \vee (\wedge_c(2,F)) \vee (\wedge_c(5,F))].
\end{aligned}
$$

This formulation of $AF(7)$ requires a 4-inputs join module and a 3-inputs or module to build the corresponding activation function module. Since $\wedge_c(2,F) = \overline{\wedge_c(2,T)}$ and $\wedge_c(5,F) = \overline{\wedge_c(5,T)}$, applying property 11 in Table 3.2, it results:

$$AF(7) = (\wedge_c(2,T)) \wedge [(6,7) \vee (\wedge_c(2,F)) \vee (\wedge_c(5,F))] \wedge (\wedge_c(5,T)) =$$
$$= (\wedge_c(2,T)) \wedge [(6,7) \vee (\wedge_c(5,F))] \wedge (\wedge_c(5,T)) =$$
$$= (\wedge_c(2,T)) \wedge (6,7) \wedge (\wedge_c(5,T))$$

The formulation of $AF(7)$ obtained through algebraic simplification requires indeed only a 3-inputs join module. Moreover this result can be formally explained, beyond the mathematical correctness, considering the PCG in Figure 3.8: node 7 corresponds to the instruction $c = a + t1$,

- operand a is given as input
- operand $t1$ has a single definition point, in node 6.

Thus node 7 is data dependent only on node 6, and there is no need in this case to consider the control path computing the data dependencies activation signal $s_d(7)$. Anyway, still reasoning about dependencies, it is possible to notice that the obtained formulation is not the minimal one. In fact, the join operator takes as input both $(\wedge_c(2,T))$ and $(\wedge_c(5,T))$, even if the generation of signal $(\wedge_c(5,T))$ implies that $(\wedge_c(2,T))$ has already been produced. Thus the minimum formulation for $AF(7)$ is given by:

$$AF(7) = (6,7) \wedge (\wedge_c(5,T))$$

that can by obtained exploiting flow informations about dependencies. This kind of optimizations will be presented in the following.

Simplification through flow analysis

The formulation obtained for the activation functions may be not minimal, also after applying the above described optimizations. This is because it may contain input combinations that will never occur. Consider, for example, the activation function of node 7 in the PCG shown in Figure 3.8. For sake of simplicity, the reduced version of such activation function has been

$(2,T)$	$(6,7)$	$(5,T)$	$AF(7)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 3.6: *Truth table for the reduced activation function of node 7 in the PCG shown in Figure 3.8.*

considered, i.e. $AF(7) = (\wedge_c(2,T)) \wedge (6,7) \wedge (\wedge_c(5,T))$. Table 3.6 reports the corresponding truth table.
Though *flow analysis* it is possible to infer the following logical implication:

$$(5,T) \Rightarrow (2,T)$$

Such implication represents the fact that, if the condition associated with node 5 has been evaluated as true, then the condition associated with node 2 has been necessarily evaluated as true too. Indeed, observing the PCG in Figure 3.8, it is clear that instruction 5 cannot be activated until instruction 2 condition results true. Hence, instruction 5 condition cannot be evaluated as true, since instruction 5 cannot be activated at all. From this implication it is possible to infer one consideration consideration about the output of the activation function $AF(7)$:

- the combination of signals $(2,T) = 0$ and $(5,T) = 1$ will never occur, hence the corresponding value of the activation function $AF(7)$ can be changed in a *don't care condition*.

The truth table for $AF(7)$, obtained after *don't care conditions* addition for the *output* values, is reported in Table 3.7.

$(2,T)$	$(6,7)$	$(5,T)$	$AF(7)$
0	0	0	0
0	0	1	$\times$
0	1	0	0
0	1	1	$\times$
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 3.7: *Truth table for the activation function of node 7 in the PCG shown in Figure 3.8, after don't care conditions addition on the output values.*

Now, *don't care conditions* addition for the *input* values can be obtained by minimization techniques, such as Karnough Maps method. For example, considering the Karnough map for $AF(7)$, after don't care conditions addition

(2,T)(6,7) \ (5,T)	1	0
00	x	0
01	x	0
11	1	0
10	0	0

Figure 3.27: *Karnough Map for the activation function of node 7 in the PCG shown in Figure 3.8, after don't care conditions addition on the output values.*

on the output values, shown in Figure 3.27, the minimum formulation of the activation function for node 7 can be obtained:

$$AF(7) = (6,7) \wedge (\wedge_c(5,T))$$

Indeed, it can be inferred that when the signal $(5, T)$ assumes value 1, the signal $(2, T)$ has necessarily value 1. Hence, the optimization removed the information of $(2, T)$ from the activation function since it is effectively redundant.

This simplification is general and maintains the correctness of the activation function also in the case of mutual exclusion. Consider, for example, node 6 of the PCG shown in Figure 3.10. Its activation function in extended form can be computed as:

$$AF(6) = (1, 6) \wedge [(4, 6) \vee (2, F) \vee (3, F)] \wedge [(5, 6) \vee (2, F) \vee (3, T)]$$

The corresponding truth table is shown in Table 3.8. By means of flow analysis the following logical implications can be inferred:

$$(3, T) \Rightarrow (2, T)$$

Table 3.8: *Truth table for the extended activation function of node 6 in the PCG shown in Figure 3.10.*

(1, 6)	**(4, 6)**	**(2, F)**	**(3, F)**	**(5, 6)**	**(3, T)**	**AF(6)**
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	0
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	0
0	0	0	1	1	1	0
0	0	1	0	0	0	0
0	0	1	0	0	1	0
0	0	1	0	1	0	0
0	0	1	0	1	1	0
0	0	1	1	0	0	0
0	0	1	1	0	1	0

Table 3.8: (continued)

(1, 6)	**(4, 6)**	**(2, F)**	**(3, F)**	**(5, 6)**	**(3, T)**	**AF(6)**
0	0	1	1	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	0	0
0	1	0	0	0	1	0
0	1	0	0	1	0	0
0	1	0	0	1	1	0
0	1	0	1	0	0	0
0	1	0	1	0	1	0
0	1	0	1	1	0	0
0	1	0	1	1	1	0
0	1	1	0	0	0	0
0	1	1	0	0	1	0
0	1	1	0	1	0	0
0	1	1	0	1	1	0
0	1	1	1	0	0	0
0	1	1	1	0	1	0
0	1	1	1	1	0	0
0	1	1	1	1	1	0
1	0	0	0	0	0	0
1	0	0	0	0	1	0
1	0	0	0	1	0	0
1	0	0	0	1	1	0
1	0	0	1	0	0	0
1	0	0	1	0	1	1
1	0	0	1	1	0	1
1	0	0	1	1	1	1
1	0	1	0	0	0	1
1	0	1	0	0	1	1
1	0	1	0	1	0	1

Table 3.8: (continued)

(1, 6)	**(4, 6)**	**(2, F)**	**(3, F)**	**(5, 6)**	**(3, T)**	**AF(6)**
1	0	1	0	1	1	1
1	0	1	1	0	0	1
1	0	1	1	0	1	1
1	0	1	1	1	0	1
1	0	1	1	1	1	1
1	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	1
1	1	0	0	1	1	1
1	1	0	1	0	0	0
1	1	0	1	0	1	1
1	1	0	1	1	0	1
1	1	0	1	1	1	1
1	1	1	0	0	0	1
1	1	1	0	0	1	1
1	1	1	0	1	0	1
1	1	1	0	1	1	1
1	1	1	1	0	0	1
1	1	1	1	0	1	1
1	1	1	1	1	0	1
1	1	1	1	1	1	1

$$(3, F) \Rightarrow (2, T)$$

$$(3, T) \Leftrightarrow \overline{(3, F)}$$

$$\overline{(3, T)} \Leftrightarrow (3, F)$$

$$(2, T) \Rightarrow (3, T) \vee (3, F)$$

$$(4, 6) \Rightarrow (3, T)$$

$$(5,6) \Rightarrow (3,F)$$

$$(2,T) \Leftrightarrow \overline{(2,F)}$$

Indeed, node 3 cannot be activated at all when the condition associated with node 2 is false, hence its associated condition cannot be evaluated neither true nor false. As a consequence, it results:

- the combination of signals $(3,T) = 1$ and $(2,T) = 0$, as well as the combination of signals $(3,F) = 1$ and $(2,T) = 0$ will never occur, hence the corresponding value of the activation function $AF(6)$ can be changed in a *don't care condition*.

- the combination of signals $(3,T) = 1$ and $(3,F) = 1$ will never occur, as well as the combination of signals $(3,T) = 0$ and $(3,F) = 0$. Hence the corresponding value of the activation function $AF(6)$ can be changed in a *don't care condition*.

- when $(2,T) = 1$ exactly one signal between $(3,T)$ and $(3,F)$ must have value 1. This is because node 2 has only not 3 as successor. Hence $AF(6)$ can be changed in a *don't care condition* when $(2,T) = 1$, $(3,T) = 0$ and $(3,F) = 0$ simultaneously.

- the value for the signals $(2,T)$ must be always different from the value for the signal $(2,F)$. Hence, when $(2,T) = 1$ and $(2,F) = 1$, or $(2,T) = 0$ and $(2,F) = 0$, the corresponding value of the activation function $AF(6)$ can be changed in a *don't care condition*.

Notice that in this case the literal $(2,T)$ is not present in $AF(6)$. However, through flow analysis it has been obtained the implication $(2,T) \Leftrightarrow \overline{(2,F)}$, allowing to say that in this case $(2,T) = 1$ is equivalent to $(2,F) = 0$. Hence, all the implications obtained can be transformed by substituting $(2,F)$ to $(2,T)$, $(2,F) = 0$ to $(2,T) = 1$ and $(2,F) = 1$ to $(2,T) = 0$. The truth table for $AF(6)$, obtained after *don't care conditions* addition, is reported in Table 3.9. For brevity, in this case only the rows for which the activation function

had value 1 before the minimization has been reported.
The resulting activation function after don't care conditions addition for node 6 is:

$$AF(6) = [(1,6) \wedge (5,6)] \vee [(1,6) \wedge (4,6)]$$

This formulation can be further reduced by applying the properties of Boolean Algebra. The resulting minimal formulation for $AF(6)$ is:

$$\begin{aligned}
AF(6) &= [(1,6) \wedge (5,6)] \vee [(1,6) \wedge (4,6)] = //(distributive) \\
&= \{[(1,6) \wedge (5,6)] \vee (1,6)\} \wedge \{[(1,6) \wedge (5,6)] \vee (4,6)\} = //(a \wedge (a \vee b) = a) \\
&= (1,6) \wedge \{[(1,6) \wedge (5,6)] \vee (4,6)\} = //(distributive) \\
&= (1,6) \wedge [(1,6) \vee (4,6)] \wedge [(5,6) \vee (4,6)] = //(a \wedge (a \vee b) = a) \\
&= (1,6) \wedge [(4,6) \vee (5,6)]
\end{aligned}$$

Minimization through flow analysis is a powerful optimization. Indeed, using the truth table after don't care conditions addition as input of a generic logic synthesis tool, the associated activation function can be automatically reduced. Moreover, in order to obtain the minimal form for activation functions, a flow analysis must be performed in conjunction with algebraic simplifications. Since data flow based optimization requires the construction of truth tables for the activation functions, a preliminar step of algebraic simplification could be performed, in order to reduce the number of literals, and thus the size of the associated truth table.
This optimization, however, has not been implemented in this thesis work. Hence, formulation and implementation of the flow analysis are left as further works.

Transformation in Product of Sum (POS) form

In many situations may result convenient having the activation function in POS form. Indeed, in the proposed architecture products corresponds to join operations; they need thus *join modules* to be performed. Sums, instead, corresponds to or operations, needing *or modules* to be performed. The

$(1,6)$	$(4,6)$	$(2,F)$	$(3,F)$	$(5,6)$	$(3,T)$	$AF(6)$	$match$
1	0	0	1	0	1	$\times$	$(3,T)=(3,F)=1$
1	0	0	1	1	0	1	$(5,6)=1,\ (3,F)=1$
1	0	0	1	1	1	$\times$	$(3,T)=(3,F)=1$
1	0	1	0	0	0	$\times$	$(3,T)=(3,F)=0$
1	0	1	0	0	1	$\times$	$(3,T)=1,\ (4,6)=0$
1	0	1	0	1	0	$\times$	$(3,T)=(3,F)=0$
1	0	1	0	1	1	$\times$	$(3,F)=0,\ (5,6)=1$
1	0	1	1	0	0	$\times$	$(2,F)=1,\ (3,F)=1$
1	0	1	1	0	1	$\times$	$(2,F)=1,\ (3,F)=1$
1	0	1	1	1	0	$\times$	$(2,F)=1,\ (3,F)=1$
1	0	1	1	1	1	$\times$	$(3,T)=(3,F)=1$
1	1	0	0	0	1	1	$(4,6)=1,\ (3,T)=1$
1	1	0	0	1	0	$\times$	$(3,T)=(3,F)=0$
1	1	0	0	1	1	$\times$	$(3,F)=0,\ (5,6)=1$
1	1	0	1	0	1	$\times$	$(3,T)=(3,F)=1$
1	1	0	1	1	0	$\times$	$(3,T)=0,\ (4,6)=1$
1	1	0	1	1	1	$\times$	$(3,T)=(3,F)=1$
1	1	1	0	0	0	$\times$	$(3,T)=(3,F)=0$
1	1	1	0	0	1	$\times$	$(2,F)=1,\ (3,T)=1$
1	1	1	0	1	0	$\times$	$(3,T)=(3,F)=0$
1	1	1	0	1	1	$\times$	$(2,F)=1,\ (3,T)=1$
1	1	1	1	0	0	$\times$	$(2,F)=1,\ (3,F)=1$
1	1	1	1	0	1	$\times$	$(2,F)=1,\ (3,T)=1$
1	1	1	1	1	0	$\times$	$(2,F)=1,\ (3,F)=1$
1	1	1	1	1	1	$\times$	$(3,T)=(3,F)=1$

Table 3.9: *Truth table for the extended activation function of node 6 in the PCG shown in Figure 3.10, after don't care conditions addition on the output values.*

join module is a sequential module, implemented as an FSM, while the *or module* is a simple *or gate*. Hence, join modules are more expansive than or modules. POS form can in same cases reduce the number of join modules, and the number of inputs needed for each join module.
Consider for example the activation function for node 5 of the PCG shown in Figure 3.8:

$$AF(5) = [(2,T) \wedge (3,5)] \vee [(8,5) \wedge (9,5)]$$

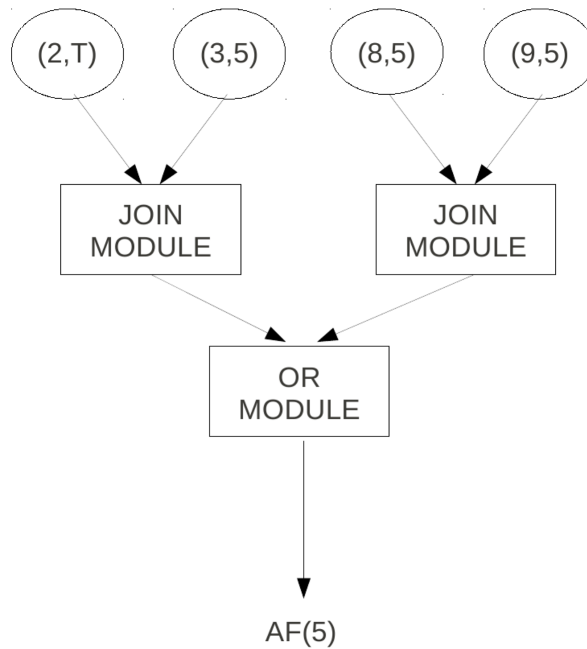

Figure 3.28: *Schematic representation of the circuit portion implementing the AF in SOP form for node 5 of the PCG in Figure 3.8.*

It is in SOP form and requires a single or module with two inputs and two join modules both with two inputs. Figure 3.28 shows a schematic representation of the circuit portion implementing $AF(5)$ in SOP form.
By applying De Morgan theorem, it results:

$$\overline{AF(5)} = [\overline{(2,T)} \vee \overline{(3,5)}] \wedge [\overline{(8,5)} \vee \overline{(9,5)}]$$

Now $AF(5)$ is in POS form. Such transformation lead to a formulation needing a single join module with two inputs and two or modules both with two inputs. Hence, it requires one join module less. However, implementing such formulation need to define the *negation operator* applied to signals. Indeed, in the proposed formulation, negation operator can be applied only to control paths, for which it has been defined. The negation of a control path has been introduced to identify the conditions indicating that a given instruction has

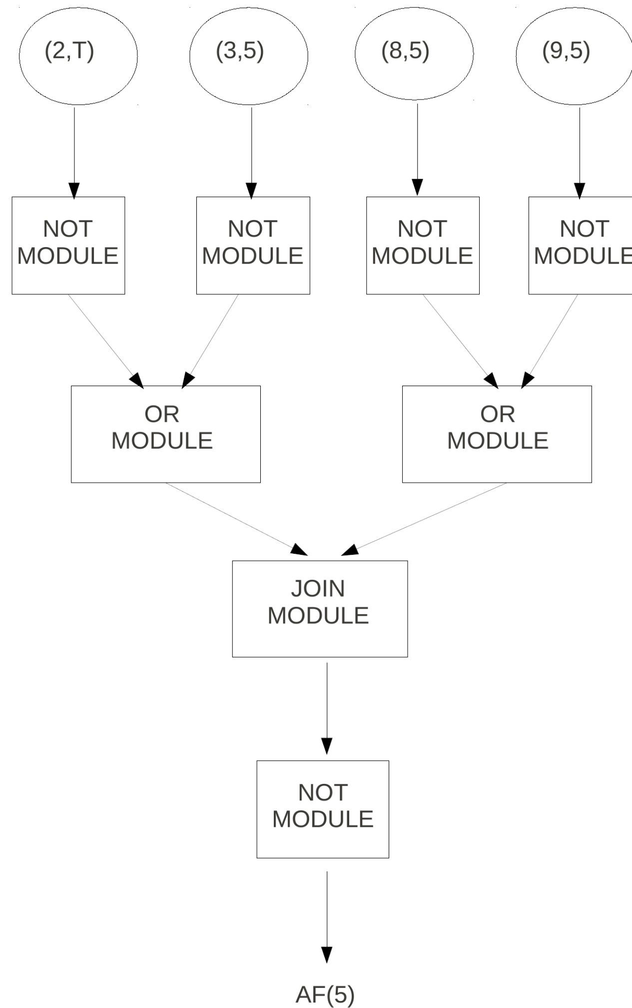

Figure 3.29: *Schematic representation of the circuit portion implementing the AF in POS form for node 5 of the PCG in Figure 3.8.*

not to be executed. When such conditions hold, the associated data dependency does not need to be satisfied. Applying De Morgan theorem, instead, lead to define the negation for the signals, in which case the negation has a different meaning. Considering conditional instruction outgoing edges, the negation applied to such edges may have different meanings. Indeed, it can mean either that the instruction has not to be executed, or that the result of the evaluation of its associated condition must be different from the label of the considered outgoing edge. Considering, instead, non-conditional instructions, the negation applied to their outgoing edges means that the instruction has not to be executed.

In any case, these are only observations, since the formalization for signals negation and for transformation in POS form is remanded to future works. For now, it is just possible to observe that such transformation needs, in the proposed example, five *not modules*. Figure 3.29 shows a schematic representation of the circuit portion implementing $AF(5)$ in POS form.

Chapter 4

Experimental Evaluation

In this chapter, the proposed methodology for the synthesis of the parallel controller (identified as PC in the rest of the chapter) has been validated in terms of performance and area occupation. In particular, the performance has been evaluated through the simulation on a set of benchmarks, while the area occupation has been evaluated through RTL synthesis targeting FPGA devices. In both cases, the results are compared with those obtained considering a target architecture whose controller is designed as a finite state machine (identified as FSM in the rest of the chapter).
Section 4.1 introduces the implementation details for the proposed methodology, while in Section 4.2 the experimental setup is presented. Finally, Section 4.3 and Section 4.4 present, compare and analyze the experimental results obtained in terms of performance and area occupation respectively.

4.1 High Level Synthesis Details

The proposed methodology has been implemented in a C++ and integrated into PandA [5] framework. PandA aims at providing an open-source framework (Figure4.1) covering different aspects of the hardware/software codesign of embedded systems, including methodologies to support:

- the research on high-level synthesis of hardware systems,

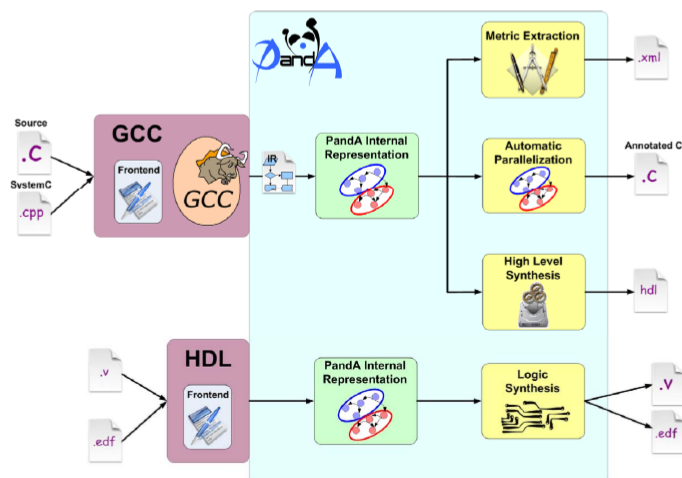

Figure 4.1: *Panda framework schematic overview.*

- parallelism extraction for software and hardware/software partitioning,
- the definition of metrics for the analysis and mapping onto multiprocessor architectures and on dynamic reconfigurability design flow.

The part for high-level synthesis receives as input a behavioral description of the algorithm, in C language, and generates the Verilog description of the corresponding RTL implementation as output, along with a testbench for the simulation and validation of the behaviour. This HDL description is then compatible with commercial RTL synthesis tools.

Internal Representation In order to construct the Parallel Controller Graph of the input specification, the C code has to be properly transformed into a graph-based internal representation.
For this purpose, PandA exploits the front-end capabilities of the GCC compiler [3] ver. 4.3.4, as shown in Figure 4.3). In particular, the source code is parsed, producing GENERIC trees: GENERIC is a language-independent representation, which interfaces the parser and the code optimizer. The GENERIC code is then translated into the GIMPLE IR, with the purpose of target- and language-independent optimizations. GIMPLE data structures

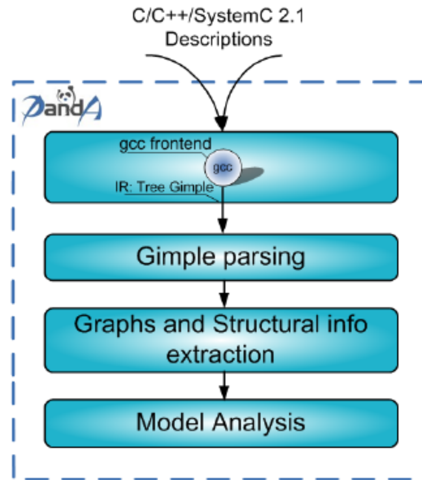

Figure 4.2: *PandA analysis flow.*

provide enough information to perform a static analysis of the specification, stored in ASCII files. Following the grammar of these files, a parser reconstructs the GIMPLE data structure in the PandA framework, thus allowing further analysis and the construction of additional internal representations, such as CFG, DFG and PDG. Once the PDG is obtained, the PCG is constructed using the algorithms proposed in the Chapter 3. Parallel Controller Graphs data structures and construction algorithms have been implemented on the top of the Boost library [1] ver. 1.40.

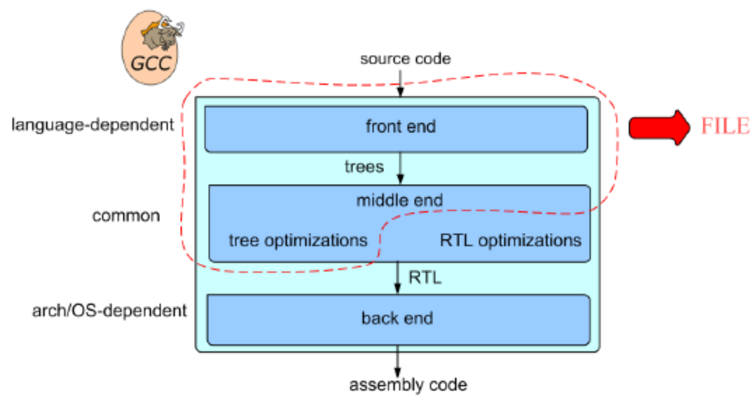

Figure 4.3: *GCC internal structure.*

Synthesis The PandA high level synthesis has been extended in order to produce the RTL descriptions of the proposed controller architectural model. Besides implementing the controller synthesis, the resource library has been also extended by introducing the modules needed to correctly implement the modules introduced in Section 3.6, such as control elements, priority managers and activation functions. The other synthesis sub-tasks (see Section 1.5 for additional details) have been accomplished by exploiting the existing algorithms already provided by the framework, as described in the following:

- **Resource allocation and binding** : these steps are performed without specifying resource constraints, allowing resource sharing; the resulting allocation and binding schemes are the same ones for both the PC and the FSM controllers. Only the register management is different, since, in PC, each variable is stored in a different storage element, following a conservative approach. On the contrary, in FSM, the register binding is performed exploiting a clique covering approach [25]. For this reason, a significant increase in the number of registers needed to implement the datapath is expected for the designs based on the parallel controller.

- **Instruction Scheduling** : scheduling is performed using a list-based algorithm [71] to compute the priority of each operation; anyhow, any of the existing scheduling algorithm that is able to define a similar priority ordering can be adopted as well. The analysis of such scheduling algorithms, or possibly the definition of proper scheduling methodologies, are postponed to further works. Moreover, chaining and multi-cycling are currently not supported, even if the extension is straightforward.

Finally, the datapath is based on a mux-based architecture. For this reason, the activation of the operations by the PC and the FSM needs to enable the proper paths from registers to functional units and viceversa by means of selectors.

4.2 Experimental setup

The proposed methodology has been validated on two set of benchmarks, classified depending on their available parallelism. To estimate the amount of parallelism in the specifications, *parallel regions* are defined as independent code portions where:

- straight line code instructions are assumed to belong to the same parallel region, even if there exist instructions that may be executed simultaneously;
- independent control constructs define different parallel regions; as an example two independent loops constructs belong to different parallel regions, while nested loops belong to the same one.

Thus parallel regions show the amount of available parallelism that common FSM-based approaches usually fail to exploit, due to the sequencing of control constructs, as discussed in Section 3.2. The two benchmark classes are characterized as follows:

1. *unique parallel region* - benchmark belonging to this class, listed in Table 4.1, are common high level synthesis benchmarks, characterized by data-intensive specifications. Except the Kim benchmark, which includes `if-then-else` constructs, all the other benchmarks are composed only of straight line sequential code. This set has been considered to verify that the proposed approach lead to the same results, in terms of clock cycles, with respect to the FSM approach, that in this case is able to statically exploit the available parallelism.
2. *multiple parallel regions and control intensive* - benchmarks belonging to this class, shown in Table 4.2, are synthetic control-intensive specifications, designed as follows:
   - Synth_lx1 is characterized by the presence of a simple loop construct, performing 50 iterations, and it is the only benchmark in this class having a unique parallel region;

Benchmark	**#source code instructions**	**# operations**	**# parallel regions**
ARF	36	33	1
Chemical	33	37	1
DCT	51	57	1
DCT_Wang	49	56	1
Dist	57	58	1
EWF	34	38	1
FIR	14	15	1
Kim	26	34	1
Paulin	11	14	1
Pr1	42	50	1
Simplebiquad	4	10	1

Table 4.1: *Unique parallel region class benchmarks characteristics.*

- Synth_lx2, Synth_lx3, Synth_lx4 are obtained combining respectively two, three and four parallel instances of Synth_lx1;
- Synth_FIR_lx1 is obtained combining FIR and Synth_lx1;
- Synth_FIR_lx2 is obtained combining FIR and Synth_lx2;
- Synth_2FIR_l_Biquad is obtained combining two instances of FIR and an independent loop whose body contains Simplebiquad;
- Synth_complex is obtained combining FIR, Kim and two independent loops, one of which is embedded in an `if` statement. This means that, for FSM, even if it would be possible to statically identify the parallelism, the scheduling performed at compile-time would increase the number of states, due to the presence of the `if` statement, whose branch condition result is unknown in advance. In fact, in such situation, all the admissible alternatives should be considered [27]. Moreover, this situation is further complicated by the fact that the number of instructions executed when the `then`

branch is very different from the number of instructions executed when the `else` branch is taken.

These benchmarks show an higher level of available parallelism, denoted by their number of parallel regions, thus a performance improvement adopting the parallel controller design is expected.

Benchmark	**#source code instructions**	**# operations**	**# parallel regions**
Synth_lx1	8	14	1
Synth_lx2	13	27	2
Synth_lx3	18	40	3
Synth_lx4	25	53	4
Synth_FIR_lx1	22	28	2
Synth_FIR_lx2	27	41	3
Synth_2FIR_l_Biquad	41	47	2
Synth_complex	66	90	3

Table 4.2: *Multiple parallel regions class benchmarks characteristics.*

Once the RTL descriptions of the selected benchmarks have been produced through the described HLS flow, the experimental evaluation is carried out in two phases:

1. performance evaluation,
2. area occupation evaluation.

RTL implementations have been simulated by means of Icarus Verilog [4] ver. 9.3, producing performance estimates in terms of clock cycles needed to perform the specification's execution. Area occupation have been extracted through RTL synthesis on FPGA, using Xilinx ISE ver. 11.1 [6], a RTL design suite that follows the typical synthesis flow illustrated in Figure4.4.

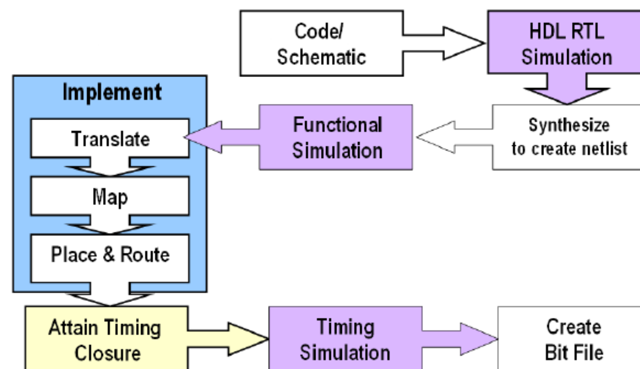

Figure 4.4: *Overview of ISE's design flow.*

The chosen metrics for the RTL synthesis have been the number of Look-Up-Tables (LUTs) and Flip-Flop (FF) registers needed to implement the design on the target device, that is an FPGA Virtex-5 XC5VLX330T (package ff1760, speed grade -2). This device has 51,840 available slices, where each of them contains four LUTs and four flip-flops. As a result, there are 207,360 LUT/flip-flop pairs available for the synthesis. In both the evaluation phases, the target frequency is fixed and, for this reason, performance can be evaluated just comparing the number of clock cycles resulting from the simulation.

4.3 Performance Evaluation

Table 4.3 shows the results of the simulations performed on the benchmarks belonging to the unique parallel region class, where the input values have been randomly assigned. The results show that adopting the PC, the designer is able to obtain the same performance of the FSM. In fact, there is a unique parallel region, which parallelism is fully detectable and exploitable at compile time. For this reason, the FSM-based approach is able to achieve the same performance of the methodology proposed in this work.
Moreover, it is worth noting that the proposed controller does not worsen the performance. This is a desirable feature, since it means that the implemented

Benchmark	#Clock Cycles (FSM)	#Clock Cycles (PC)
Arf	10	10
Chemical	9	9
Dct	13	13
Dct_Wang	12	12
Dist	12	12
Ewf	9	9
Fir	8	8
Kim	12	12
Paulin	5	5
Pr1	12	12
Simplebiquad	5	5

Table 4.3: *Clock Cycles for the execution of class1 benchmarks.*

token-driven approach which manages the activation of all operations does not introduce communication overheads, as it happens instead for designs based on more complicated communication protocols.
The results obtained for the second class of benchmarks, that are characterized by control-intensive specifications, are presented in the following.

Benchmark	#Clock Cycles FSM	#Clock Cycles PC	Speedup
Synth_FIR_lx1	162	158	1.02
Synth_FIR_lx2	318	162	1.96
Synth_2FIR_l_Biquad	163	158	1.03

Table 4.4: *Clock Cycles for the execution of Synth_FIR_lx1, Synth_FIR_lx2 and Synth_2FIR_l_Biquad benchmarks.*

Table 4.4 shows the performance obtained through the simulation of benchmarks Synth_FIR_lx1, Synth_FIR_lx2 and Synth_2FIR_l_Biquad. The first one is composed of two parallel regions, but their parallelism is not enough

to lead to a significant speedup. In fact, the parallel regions are *unbalanced*: there is an initial phase in which the FIR's code and the parallel loop are concurrently executed, exploiting parallelism and available resources, and leading to a gain of 4 clock cycles on the execution time. When the FIR's code is completely executed, parallelism is no longer available (there is only one parallel region to be executed), and the parallel controller offers the same performance of the FSM-based controller. The same situation is found for benchmark Synth_2FIR_1_Biquad. Results obtained for benchmark Synth_FIR_lx2 instead, show a good increase of the speed-up. In this case when the FIR's code has been executed, there are still two parallel loops that can be concurrently executed, and the parallel controller exploits this kind of parallelism.

simulation	**#Clock Cycles FSM**	**#Clock Cycles PC**	**Speedup**
RUN1	203	139	1.86
RUN2	153	83	1.84
RUN3	139	99	1.41
RUN4	169	135	1.25
RUN5	211	132	1.60

Table 4.5: *Clock Cycles for the execution of Synth_complex benchmarks.*

Table 4.5 provides the performance for the Synth_complex benchmark. The performance has been evaluated through five different runs of this benchmark, assigning different random values to input variables for each run. In this case, due to its control-intensive nature, input variables significantly affect the number of instructions to be performed, and random input values ensure to maintain the unpredictability of the execution flow. Thus it represents a good challenge for dynamic extraction and exploitation of parallelism. The parallel controller, in these simulations, effectively demonstrates the capability of extracting parallelism even in the case of unpredictable behavior at run-time.

All the following simulation results refer to benchmarks Synth_lx1, Synth_lx2,

Synth_lx3 and Synth_lx4, shown in Figure 4.6. Considering synth_1x1, even

Benchmark	#Clock Cycles FSM	#Clock Cycles PC	Speedup
Synth_lx1	156	156	1
Synth_lx2	312	159	1.96
Synth_lx3	467	162	2.88
Synth_lx4	623	315	1.98

Table 4.6: *Clock Cycles for the execution of Synth_lx's benchmarks.*

if the specification is control-dominated, there is a unique parallel region, and the simulations outcomes confirm the results obtained for the first class of benchmarks. The situation significantly changes when increasing the number of parallel regions from 1 to 2. In this case (benchmark synth_1x2) the number of clock cycles needed to perform the computation is almost halved adopting the parallel controller design instead of the FSM-based one. Since the relative weight of the sequential fraction of the code (i.e. the initialization instructions) is small with respect to the total number of instructions, the obtained speed-up is near to the maximum theoretical one (e.g., 2x), given by the number of parallel regions. Augmenting the parallelism degree to 3, the obtained speedup still increases, also in this case reaching a value near to the theoretical maximum (e.g., 3x). However, when increasing the number of parallel regions to 4 (i.e., benchmark synth_1x4), the results show a decrease of the speedup with respect to the previous case. This is due to the saturation of available resources: the first three loops exploit all the available resources, and the fourth one is sequentialized. However, resource availability is not the only aspect mitigating the speedup increase. Indeed, as above mentioned, in the current implementation of the parallel controller, the resources binding is given and performed by means of the algorithms mainly developed for the FSM. Such algorithms are mainly designed to address the parallism identified by the FSM. As a result, also independent instructions may be bound on the same resource, reducing the parallelism exploitation.

4.4 Area Evaluation

The first step performed to evaluate area occupation of the proposed implementation has been the analysis of some metrics extracted during the high level synthesis flow. In particular, Table Table 4.7 shows the resources needed to be allocated to implement the parallel controller for each benchmark, and the number of flip-flop registers expected on the basis of the such resources, leading to the following considerations:

- the number of Control Elements (#CE) is equal to the number of operations reported in Table 4.1 and Table 4.2, since each operation requires a Control Module to be instantiated;
- the number of Conditional Join Modules (#C-JOIN) is equal to the number of conditional operations;
- the number of Priority Managers provides information about the number of functional units that share at least two operations;
- the number of Join modules is larger than the number of Or modules for each benchmark, and this is due to the current formulation of the proposed activation function. This is a relevant issue, implementing a Join module requires a number of flip-flops equal to the number of its inputs.
- the estimated total number of Flip Flops (FF) is computed as the sum of the FF needed for both Join modules and Control Elements: in almost all the cases the number of Join FF is at least an half of the overall number, and this underlines the impact that join modules have on the area overheads for the parallel controller.

In conclusion, for the reduction of the Join modules, and thus the number of flip-flops, transformations from SOP to POS forms are definitively required, remarking the potential impact of the optimization algorithms proposed in Section 3.10.

	Allocated resources					Estimated #FF	
Benchmark	**#CE**	**#PT**	**#C-JOIN**	**#JOIN**	**#OR**	**#JOIN-FF**	**#FF**
ARF	33	7	0	14	0	28	61
Chemical	37	5	0	19	0	38	75
DCT	57	14	0	32	0	64	121
DCT_Wang	56	13	0	26	0	52	108
Dist	58	6	0	22	0	44	102
EWF	38	5	0	20	0	40	78
FIR	15	1	0	7	0	14	29
Kim	34	7	2	30	29	96	130
Paulin	14	5	0	5	0	10	24
Pr1	50	11	0	25	0	50	100
Simplebiquad	10	1	0	4	0	8	18
Synth_lx1	14	3	1	9	6	24	38
Synth_lx2	27	7	2	18	12	48	75
Synth_lx3	40	9	3	27	18	72	112
Synth_lx4	53	11	4	36	24	96	149
Synth_FIR_lx1	28	5	1	16	6	38	66
Synth_FIR_lx2	41	8	2	25	12	62	103
Synth_2FIR_lBq	47	6	1	23	3	49	96
Synth_complex	90	11	6	68	54	188	278

Table 4.7: *Number of resources needed to implement the parallel controller and estimation of the required flip-flops (Estimated #FF). #CE is the number of Control Elements, #PT of the Priority Managers, #C-JOIN of the Conditional Join Modules, #JOIN of the other Join Modules, #OR of the OR modules. Estimated #JOIN-FF is the number of estimated flip-flops for the Join Modules.*

	Controller Estimated # FF		DataPath Estimated #FF	
Benchmark	**FSM**	**PC**	**FSM**	**PC**
ARF	10	61	256	896
Chemical	9	75	544	1024
DCT	13	121	512	1536
DCT_Wang	12	108	320	1504
Dist	12	102	704	1664
EWF	9	78	544	1056
FIR	8	29	224	416
Kim	26	130	96	832
Paulin	5	24	160	320
Pr1	12	100	352	1312
Simplebiquad	5	18	128	256
Synth_lx1	8	38	48	108
Synth_lx2	16	48	192	608
Synth_lx3	23	112	256	928
Synth_lx4	31	149	320	1248
Synth_FIR_lx1	14	66	288	704
Synth_FIR_lx2	22	103	288	1024
Synth_2FIR_lBq	15	96	640	1280
Synth_complex	57	278	352	2112

Table 4.8: *Estimated number of Flip Flops for controller and datapath, in the case of both FSM and PC-based designs.*

The estimated number of needed Flip Flops has been computed even for the FSM controller and the Datapath, for both PC and FSM designs.
Results are compared in Table 4.8 and, from their analysis, it is possible to draw the following considerations:

- *Controller* - FSM requires in all cases a lower number of registers with respect to PC; however it must be considered that FSM optimizations to better exploits parallelism, such as the technique in [27], could lead to an exponential increase of the number of states and consequently of the number of needed flip-flops. Moreover, as previously discussed, optimizations of the activation functions could significantly reduce the number of flip-flops in PC designs. Anyway, the number of flip-flops required for the datapath implementations usually dominates the controller ones by at least one order of magnitude.

- *Datapath* - The estimated number of needed register to implement the datapath, is significantly higher in PC designs. This is due to the conservative approach adopted for register binding in the PC-based synthesis. These results clearly show that a proper register binding definition is very important, but it has been left as further work since it is out of the scope of this thesis.

Estimated results have been in the following compared with actual ones obtained through RTL synthesis. Table 4.9 compares the estimated total number of flip flops needed to implement FSM and PC designs, with the actual number of used register in the physical implementation on FPGA. The results show that the synthesis process can be effectively able to predict the requirements in terms of resources. Moreover, the proposed approach for PC increases the predictability of the designs, demonstrated by a lower average difference between estimated and actual values. This opens new possibilities for research in the exploration of the architecture (e.g., priority scheduling and operation binding) at higher level of abstraction, i.e. without performing the actual synthesis for each candidate solution.

	Total Estimated # FF		Total Actual #FF	
Benchmark	**FSM**	**PC**	**FSM**	**PC**
ARF	266	957	385	957
Chemical	553	1099	554	1099
DCT	525	1657	532	1653
DCT_Wang	332	1612	332	1602
Dist	716	1766	718	1766
EWF	553	1134	553	1134
FIR	232	445	232	446
Kim	122	962	122	1025
Paulin	165	344	165	344
Pr1	364	1412	365	1408
Simplebiquad	133	274	133	275
Synth_lx1	56	146	138	323
Synth_lx2	208	656	208	679
Synth_lx3	279	1040	279	1034
Synth_lx4	351	1397	351	1389
Synth_FIR_lx1	302	770	312	767
Synth_FIR_lx2	310	1127	322	1123
Synth_2FIR_lBq	655	1376	656	1375
Synth_complex	409	2390	432	2376
Avg. Difference			3,80%	1.00%

Table 4.9: *Total number of estimated and actually used Flip Flops.*

	Total Actual #LUT	
Benchmark	**FSM**	**PC**
ARF	1066	896
Chemical	1148	781
DCT	2232	1862
DCT_Wang	2330	1722
Dist	2045	1615
EWF	1296	845
FIR	399	295
KIM	832	1163
Paulin	457	381
Pr1	1918	1566
Simplebiquad	137	118
Synth_lx1	270	294
Synth_lx2	495	581
Synth_lx3	653	946
Synth_lx4	1013	1360
Synth_FIR_lx1	871	647
Synth_FIR_lx2	934	943
Synth_2FIR_lBq	1420	982
Synth_complex	2111	2484

Table 4.10: *Number of used LUTs.*

Finally, Table 4.10 show the requirements in terms of LUTs to implement the entire benchmarks, with both the controllers. The results show that the number of LUTs required by the PC is usually reduced with respect to the designs based on the FSM. In fact, when a larger number of registers is used (as in the case of PC), the number of multiplexers is usually reduced since there are less conflict between paths from/to functional units. For this reason, the number of registers and multiplexers have to be necessarily explored at the same time in order to reduce the number of the registers, without degrading the number of the multiplexers.

4.5 Concluding Remarks

In conclusion, the proposed approach is effectively able to exploit the parallelism by identifying at run-time the instructions that can be executed step by step. On the other hand, the results show that efficient algorithms have to be necessarily designed for determining the proper operation binding and the corresponding priority values in order to further improve the performance. From the analysis of the area occupation, additional observations can be made. First, since the implementation of a Join module requires a number of flip-flops equal to the number of its inputs and such modules have a large impact on the number of flop-flops required for the PC implementation, transformations from SOP to POS forms are definitively required for the reduction of the Join modules. Then, the results show that the proposed approach for PC increases the predictability of the designs, opening new possibilities for optimizations at higher level of abstraction, i.e. without performing the actual synthesis for each candidate solution.
In conclusion, concurrent explorations of scheduling priorities and operation bindings, as well as registers and multiplexers will be able to further improve performance and area occupation of the proposed controller. Finally, additional optimizations such as adding *don't care* conditions and transformations from SOP to POS would further reduce the number of required resources.

Chapter 5

Conclusions and Future Works

The main objective this thesis work aimed to achieve was to optimize parallelism exploitation in HLS. For such purpose, two main contributions have been proposed: a new parallel controller structure, and a methodology for the design of such architecture in high-level synthesis.
The guideline of the whole project is that the task of establish when an instruction can start its execution can be shifted on the controller. Making the controller able to dynamically select the set of instructions to execute in each control step eliminates most of the restrictions in parallelism extraction. The only limitation may come from binding and number of available resources. Indeed, in the proposed model, the binding task is performed in advance as well as resources allocation.
The proposed methodology started with parallelism representation. A new graph, namely the Parallel Controller Graph (PCG), has been proposed as Intermediate Representation (IR) for such purpose. The PCG has been constructed as an extension of the Program Dependence Graph (PDG) representing the minimum set of control flow informations, in addition to the minimum set of control and data dependencies already represented by the PDG. Such informations resulted in the insertion of back-edges, labeled control flow edges and unlabeled control flow edges. The algorithms for the insertion of these kind of edges, and thus for the PCG construction, have

been presented.
Representing the inherent parallelism of an application was not the only aim leading to the PCG construction. Such IR, indeed, was built in such a way to easily support the extraction of the informations about the minimum set of conditions enabling instruction execution. These conditions indicate what events must happen for an instruction can start its execution, thus expressing when the dependencies are satisfied. Hence, an analysis of the PCG has been performed at first to identify the parallelism; then, the identified parallelism has been formalized into activation functions, representing the above mentioned conditions. Activation functions has been implemented as combinations of signals. This allows the parallel controller to automatically understand and manage such information, addressing the parallelism formalization problem. Activation functions represent the basis to build an execution model. Indeed, they provide not only the information about the exploitable parallelism, but also a complete characterization of the control flow of the synthesized application.
The proposed architecture for the parallel controller has been obtained through a bottom-up approach. First, the set of modules composing the controller has been defined, describing their behavior. Then, such modules has been properly interconnected. The resulting architecture is event-driven following the producer/consumer paradigm. Each time an instruction is executed it fires, informing the instructions depending on it that the associated dependency is satisfied. More in detail, since each instruction depends on a set of instructions, properly combined according to the activation function, each time an instruction is executed the fired signal is collected by the AF module. The AF module in turn will fire when all the dependencies will be satisfied, informing the associated instructions that its execution can start.
The proposed methodology has been implemented integrating the C++ High-Level Synthesis framework into PandA [5]. It has been applied on a set of benchmarks, obtaining their RTL description targeting the parallel controller. Then, the same set of benchmark has been synthesized targeting a

centralized FSM. The performance obtained in these two cases have been compared in terms of clock cycles. This has been possible since the synthesis has been performed setting the frequency in advance. The parallel controller has shown good performance, especially for those specifications characterized by an high degree of available parallelism. However, area occupancy, evaluated through RTL synthesis, increased. One of the main reasons for this is that register allocation is optimized in the case of the FSM, while these optimizations has not been extended to the case of the parallel controller yet.
In conclusion, the main goal of this thesis work was to propose a parallel controller structure improving performance by parallelism exploitation. This objective has been achieved. However, although the obtained results are satisfying, parallel controller capabilities could be extended in future works, improving area occupation and giving even better performance. Some hints for further improvements are provided in the following.

- The optimizations described in Chapter 3, i.e. translation into SSA form, algebraic simplification, reduction trough flow analysis and translation into POS form, can be formalized and implemented. This would lead to a reduction of the logic needed to implement activation functions into hardware, possibly reducing the number of registers.

- Registers allocation can be improved by applying optimizations techniques. Standard optimization techniques, however, may result inadequate for the parallel controller, since concurrent run-time-established execution of multiple instructions, complicates the analysis for registers reuse. Hence, the proposal of novel algorithms may be needed.

- Resources allocation and binding are currently performed in the same way for the FSM and for the parallel controller. More in detail, the parallel controller uses the allocation and binding algorithms developed for the FSM. This means that they are performed starting from the information contained in the Control Flow Graph (CFG). Since the CFG

fails in recognize all the sources of parallelism, it can be allocated a lower number of resources of a certain kind with respect of the number of independent operations needing that kind of resource to be executed, thus restricting parallelism exploitation. Moreover, multiple independent operations can be recognized as dependent in the CFG. In such cases it may be bound on the same resources. For these reasons, the algorithms performing resources allocation and binding starting from the PCG should be developed.

Ringraziamenti

Per prima cosa, è nostro dovere ringraziare la deadline per aver posto fine al "*bagno di sangue*"... o almeno alla sua prima fase!
Ringraziamo il grande capo, il professor Fabrizio Ferrandi, perchè è riuscito a trasmetterci la sua grande passione per questo affascinante lavoro. Il suo continuo desiderio di fare meglio ci ha "costretti "alla ricerca di risultati sempre migliori, permettendoci di raggiungere di conseguenza importanti traguardi professionali e personali. Grazie capo, per i continui stimoli che ci fornisce.
Ringraziamo il nostro caro correlatore, Christian Pilato, per il suo continuo sostegno a tutte le ore del giorno e della notte. Il suo aiuto è stato decisamente indispensabile per la riuscita di questo progetto. Grazie Christian, perchè non eri costretto a farlo (o perlomeno non con così tanto entusiamo!) e invece non ti sei mai tirato indietro.
Ringraziamo le nostre famiglie, che anche a mille kilometri di distanza, non hanno mai smesso di darci il loro sostegno. Anche questo traguardo lo abbiamo raggiunto grazie a voi e con voi!
Ringraziamo gli amici, quelli vecchi e quelli nuovi. In particolare un grazie va a Marco, ormai detto il comasco, che sa sempre come risollevarti il morale, anche quando (dettaglio) non è sua intenzione!
Infine, ringraziamo il *panda* che è stato il simbolo del nostro percorso universitario, sin dagli esordi baresi, e che ci ha guidati fin qui. Non si sa come, in un modo o nell'altro il panda è sempre stato presente!

DAN4ER!

Bibliography

[1] Boost C++ Libraries. http://www.boost.org.

[2] "Celoxica Ltd., 'Handel-C Language Reference Manual, v3.0', 2002".

[3] CGG - GNU Compiler Collection. http://gcc.gnu.org.

[4] Icarus Verilog. http://www.icarus.com/eda/verilog.

[5] Panda framework official site. http://trac.elet.polimi.it/panda.

[6] Xilinx. Synthesis tools for FPGA devices. http://www.xilinx.com.

[7] A. Bardsley. *"Implementing Balsa Handshake Circuits"*. PhD thesis, Univerity of Manchester, 2000.

[8] A. Crews and F. Brewer. "Controller Optimization for Protocol Intensive Applications". *Proc. of EURO-DAC*, 1996.

[9] A. Seawright and F. Brewer. "Clairvoyant: A Synthesis System for Production-Based Specification". *IEEE Trans. on VLSI*, pages 172–185, June 1994.

[10] Alfred V. Aho. *"Algorithms for finding patterns in strings"*, chapter 5, pages 255–300. Elsevier, Amsterdam, 1990.

[11] A.R. Karlin, H.W. Trickey, and J.D. Ullman. "Experience with a Regular Expression Compiler". *ICCD*, pages 656–665, 1983.

[12] B. Akers Sheldon. "Binary Decision Diagrams". *IEEE Transactions on Computers*, pages 509–516, June 1978.

[13] B.A. Cota, D.G. Fritz, and R.G. Sargent. Control flow graphs as a representation language. *Proceedings of the Winter Simulation Conference*, pages 555–559, 1994.

[14] Berkeley Design Technology Incorporation. "FPGAs for DSP: An Independent Perspective". *Xilinx Workshop, Embedded Systems Conference*, April 3, 2007.

[15] I. B. By the staff of Berkeley Design Technology. "An Independent Evaluation of: High-Level Synthesis Tools for Xilinx FPGAs". Technical report, Berkeley Design Technology, Inc, 2010.

[16] R. Camposano and W. Wolf. *"High-Level VLSI Synthesis"*. Kluwer (now Springer), Dordrecht, 1991.

[17] Carl A. Petri. "Communication with automata". *DTIC Research Report*, 1966.

[18] Chris Papachristou and Yusuf Alzazeri. "A method of distributed controller design for RTL circuits". *Proc. of Design Automation and Test in Europe*, 1999.

[19] P. Coussy, D. Gajski, M. Meredith, and A. Takach. "An Introduction to High-Level Synthesis". *Design Test of Computers, IEEE*, 26(4):8–17, jul. 2009.

[20] M. Dhodhi, F. Hielscher, R. Storer, and J. Bhasker. "Datapath Synthesis Using a Problem-Space Genetic Algorithm". *IEEE Transac- tions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, August 1995.

[21] E. J. McCluskey. *"Introduction to the Theory of Switching Circuits"*, chapter 2. McGraw-Hill, 1965.

[22] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The program dependencies and its use in optimization". In *Proceedings of ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 9, Issue 3, pages 319–349, July 1987.

[23] Fredrick J. Hill and Gerald R. Peterson. *"Introduction to the Theory of Switching Circuits"*. McGraw-Hill, 1965.

[24] G. Berry and G. Gonthier. "The ESTEREL synchronous programming language: design, semantics, implementation". *Science of Computer Programming*, 19(2):87–152, 1992.

[25] G. De Micheli. *"High-Level Synthesis: From Algorithm to Digital Circuits"*. Springer, 2008.

[26] G. De Micheli. *"Synthesis and Optimization of Digital Circuits"*. McGraw-Hill, New York, 1994.

[27] G. Lakshminarayana, K.S. Khouri, and N.K. Jha. "Wavesched: a novel scheduling technique for control-flow intensive designs". *IEEE Trans. on CAD*, 18(5):505–523, May 1999.

[28] D. Gajski et al. *"High Level Synthesis: An Introduction to Chip and System Design"*. Kluwer (now Springer), 1992.

[29] Geiger and Muller-Wipperfurth. "FSM decomposition revisited: algebraic structure theory applied to MCNC benchmark FSMs". *28th ACM/IEEE Design Automation Conference (DAC '91)*, pages 182–185, 1991.

[30] W. Geurts et al. *"Accelerator Data-Path Synthesis for High-throughput Signal Processing Applications"*. Kluwer Academic Publishers, 1996.

[31] G.G. de Jong. "data flow graphs: system specification with the most unrestricted semantics". pages 401–405, Feb. 1991.

[32] R. K. Gupta and F. Brewer. *"A Retrospective in High-Level Synthesis"*. Springer, 2008, chapter 2.

[33] S. Gupta. *"Coordinated Coarse-Grain and Fine-Grain Optimizations for High-Level Synthesis"*. PhD thesis, PhD thesis, University of California, Irvine, 2003.

[34] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. *"SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits"*. Kluwer, Dordrecht, 2004.

[35] H. P. Zeiger. *"Loop-free synthesis of finite-state machines"*. PhD thesis, MIT, Department of Electrical Engineering, Cambrige, Mass., September 1961.

[36] C. Hitchcock and D. Thomas. "A method of automatic data path synthesis". *Design Automation Conference*, 1983.

[37] Hugo De Man. "Cathedral-II: A Silicon Compiler for Digital Signal Processing". *IEEE Design and Test*, vol. 3, no. 6, 1986, pp. 13-25.

[38] Hyunuk Jung, Kangnyoung Lee, and Soonhoi Ha. "Efficient hardware controller synthesis for synchronous dataflow graph in system level design". *IEEE Trans. on VLSI Systems*, Aug. 2002.

[39] J. Eppling and D. Thomas. "Multiple Controller Synthesis for Reducing Control Path Delays". *SRC TECHCON*, Sept. 1993.

[40] J. Hartmanis. "Symbolic analysis of a decomposition of information processing". *In Inform. Control*, June 1960.

[41] J. Hartmanis and R. E. Stearns. *"Algebraic Structure Theory of Sequential Machines"*. Prentice-Hall, Englewood CIiffs, N. J., 1966.

[42] James E. Smith and Andrew R. Pleszkun. "Implementing Precise Interrupts in Pipelined Processors". *Proceedings of the 12th annual international symposium on Computer architecture*, 1985.

[43] James Lyle Peterson. *"Petri Net Theory and the Modeling of Systems"*. Prentice Hall, 1981.

[44] João M. Fernandes, M. Adamski, and A.J. Proença. "VHDL generation from hierarchical petri net specifications of parallel controllers". *Proc. of IEEE Computer and Digital Techniques*, 1997.

[45] John E. Hopcroft and Jeffrey D. Ullman. *"Introduction to Automata Theory, Languages, and Computation"*. Addison-Wesley Publishing, 1979.

[46] Jörg Desel and Gabriel Juhás. *"What Is a Petri Net? Informal Answers for the Informed Reader"*, pages 1–25. Hartmut Ehrig et al., 2001.

[47] K. Bilinski, E. Dagless, and J. Mirkowski. " Synchronous parallel controller synthesis from behavioural multiple-process VHDL description". *Proc. of European Design Automation Conference*, Sept. 1996.

[48] K. Thulasiraman and M. N. S. Swamy. *"Acyclic Directed Graphs"*, chapter 5, Section 7. John Wiley and Son, 1992.

[49] D. Knapp. *"Behavioral Synthesis: Digital System Design using the Synopsys Behavioral Compiler"*. Prentice-Hall, Englewood Cliff, NJ, 1996.

[50] T. Kowalski and D. Thomas. "The VLSI design automation assistant: what's in a knowledge base". *Design Automation Conference*, 1985.

[51] Krzysztof Biliński, E.L. Dagless, J.M. Saul, and M. Adamski. "Parallel controller synthesis from a Petri net specification". *Proc. of European Design Automation Conference*, 1994.

[52] G. Lakshminarayana, A. Raghunathan, and N. Jha. "Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions". *Design Automation Conference*, 1998.

[53] M. Yoeli. "The cascade decomposition of sequential machines". *In IRE Trans. Electronic Computers*, April 1961.

[54] G. Martin and G. Smith. "High-Level Synthesis: Past, Present, and Future". *IEEE Design & Test of Computers*, December 2009, vol. 26(4), pp. 18-25.

[55] M. C. McFarland, A. C. Parker, and R. Camposano. "Tutorial on High-level Synthesis". In *Conference, in Proc. of 25th ACM/IEEE Design Automaton*, pages 330–336, 1988.

[56] M. Meredith. "A Look Inside Behavioral Synthesis". *EE Times*, Aug. 2004.

[57] Milind Girkar and Constantine D. Polychronopoulos. "The hierarchical task graph as a universal intermediate representation". *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[58] M.O. Rabin and D. Scott. "Finite Automata and their Decision Problems". *IBM Journal of Research and Development*, pages 115–125, 1959.

[59] Nancy Wu, Gary Smith EDA. "ESL Synthesis: Tips for Implementing a Viable ESL-Synthesis Flow". *EDN Magazine*, July 30, 2010.

[60] Noam Chomsky. "Three Models for the Description of Language". *IRE Transactions on Information Theory*, pages 113–123, 1956.

[61] N. Ntlatlapa. "Verified High-Level Synthesis Front-End and Simulator Using Dependence Flow Graphs". Technical report, Auburn University (Auburn, Alabama, USA), 1999.

[62] N. Ntlatlapa. "High-Level Synthesis using Dependence Flow Graphs as the Intermediate Form". Technical report, Auburn University (Auburn, Alabama, USA), January 20, 1998.

[63] P. Ashar, S. Devadas, and R. Newton. "Optimum and heuristic algorithms for an approach to finite state machine decomposition". *IEEE Trans. on CAD*, 10:296–310, 1991.

[64] A. Parker, J. Pizarro, and M. Mlinar. "MAHA: A program for datapath synthesis". *Proc. 23rd IEEE/ACM Design Automation Conference*, pp. 461-466, Las Vegas NV, June 1986.

[65] P. Paulin and J. Knight. "Scheduling and binding algorithms for high-level synthesis". *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.

[66] P. G. Paulin and J. Knight. "Force-Directed Scheduling for the Behavioral Synthesis of ASICs". *IEEE Trans. Computer-Aided Design of Integrated Circuits and Sys- tems*, 1989, vol. 8, no. 6, pp. 661-679.

[67] P. G. Paulin and J. Knight. "Force-Directed Scheduling in Automatic Data Path Synthesis". *Proc. of the 24th Design Automation Conference*, June 1987, pp. 195-202.

[68] O. Penalba, J. Mendias, and R. Hermida. "Maximizing conditional reuse by pre-synthesis transformations". *Design Automation and Test in Europe*, 2002.

[69] M. Pinedo. *"Scheduling Theory, Algorithms and Systems"*. Prentice Hall, 1995.

[70] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[71] R. L. Graham. "Bounds on multiprocessing timing anomalies". *Journal of Applied Mathematics*, 17:416–429, 1969.

[72] I. Radivojevic and F. Brewer. "A New Symbolic Technique for Con- trol-Dependent Scheduling". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 1, January 1996.

[73] Reinaldo A. Bergamaschi. " Browse ¿ Conferences¿ Design Automation Conference, ... Behavioral network graph unifying the domains of high-level and logic synthesis". *Proc. of the 36th Design and Automaton Conference*, pages 213–218, June 1999.

[74] M. Rim, Y. Fann, and R. Jain. "Global Scheduling with Code-Motions for High-Level Synthesis Applications". *IEEE Transactions on VLSI Systems*, vol. 3, no. 3, pp. 379-392, September 1995.

[75] R.W. Floyd and J.D. Ullman. "The Compilation of Regular Expressions into Integrated Circuits". *Jo. ACM*, 1982.

[76] S. Amellal and B. Kaminska, . "scheduling of a control data flow graph". pages 1666–1669 vol.3, May 1993.

[77] S. Sarkar, S. Dabral, P. K. Tiwari, and R. S. Mitra. "Lessons and Experiences with High-Level Synthesis". *IEEE Design & Test*, Vol. 26 , Issue 4, pp 34-45, July 2009.

[78] S.Devadas and R. Newton. "Decomposition and Factorization of Sequential Finite State Machines". *IEEE Trans. on CAD*, 8:1206–1217, 1988.

[79] M. Sipser. *"Introduction to the Theory of Computation"*. PWS Publishing Co., Boston 1997.

[80] Taylor L. Booth. *"Sequential Machines and Automata Theory"*. John Wiley and Sons, 1967.

[81] Taylor L. Booth. *"Digital Networks and Computer Systems"*. John Wiley and Sons, 1971.

[82] A. Timmer and J. Jess. "Exact Scheduling Strategies based on Bipartite Graph Matching". *Proceedings of the European Design & Test Conference*, pp. 42-47, 1995.

[83] C. Tseng and D. Siewiorek. "Automated synthesis of data paths in digital systems". *Proceedings of the 20th Design Automation Conference*, July 1986.

[84] D. Verkest, J. Kunkel, and F. Schirrmeister. "System Level Design Using C++". *Proc. of Design, automation & Test in Europe*, Paris, France, pp. 74-83, 2000.

[85] K. Wakabayashi and T. Yoshimura. "A resource sharing and control synthesis method for conditional branches". *IEEE International Conference on Computer-Aided Design*, 1989.

[86] T. Yamashita, K. Matsuzaki, T. Toyoyama, M. Satoh, A. Adachi, and F. Sugawara. "Using high-level synthesis for FPGA development. Applying a C-based design methodology using Catapult C Synthesis at FUJITSU". Technical report, Design Platform Development Center Corporate Product Technology Unit, FUJITSU LIMITED, 2007.