Formal Semantics of Hardware Compilation Framework

Samira Ait Bensaid

université PARIS-SACLAY

Formal Semantics of Hardware Compilation Framework

Sémantique Formelle d'une Infrastructure de Compilation Matériel

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580 : Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique
Référent : Faculté des sciences d'Orsay

Thèse préparée à **l'Institut LIST** (Université Paris-Saclay, CEA), sous la direction de **Mathieu JAN**, Directeur de recherche, et le co-encadrement de **Mihail ASAVOAE**, Ingénieur-chercheur.

Thèse soutenue à Paris-Saclay, le 28 novembre 2023, par

Samira AIT BENSAID

Composition du jury

Membres du jury avec voix délibérative

Laurent PAUTET Professeur des universités, Télécom Paris	Président
Stephan MERZ Directeur de recherche, INRIA & LORIA, Université de Lorraine	Rapporteur & Examinateur
Steven DERRIEN Professeur des universités, Université de Rennes 1	Rapporteur & Examinateur
Jean-Luc BECHENNEC Chargé de recherche, CNRS, Université de Nantes	Examinateur
Isabelle PUAUT Professeure des universités, Université de Rennes 1	Examinatrice

THESE DE DOCTORAT
NNT : 2023UPASG085

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Sémantique Formelle d'une Infrastructure de Compilation Matériel
Mots clés : Modélisation des processeurs, Langages de Construction de Matériel (HCLs), Chemin de données du pipeline, Anomalies temporelles, Vérification de modèle

Résumé : Les analyses statiques de pire temps d'exécution sont utilisées pour garantir les délais requis pour les systèmes critiques. Afin d'estimer des bornes précises sur ces temps d'exécution, ces analyses temporelles nécessitent des considérations sur la (micro)-architecture. Habituellement, ces modèles de micro-architecture sont construits à la main à partir des manuels des processeurs. Cependant, les initiatives du matériel libre et les langages de description de matériel de haut niveau (HCLs), permettent de réaborder la problématique de la génération automatique de ces modèles de micro-architecture, et plus spécifiquement des modèles de pipeline. Nous proposons un workflow qui vise à construire automatiquement des modèles de chemin de données de pipeline à partir de conceptions de processeurs décrites dans des langages de contruction de matériel (HCLs). Notre workflow est basé sur la chaine de compilation matériel Chisel/FIRRTL. Nous construisons au niveau de la représentation intermédiaire les modèles de pipeline du chemin de données. Notre travail vise à appliquer ces modèles pour prouver des propriétés liées à la prédictibilité temporelle. Notre méthode repose sur la vérification formelle. Les modèles générés sont ensuite traduits en modèles formels et intégrés dans une procédure existante basée sur la vérification de modèles pour détecter les anomalies de temps. Nous utilisons le langage de modélisation et de vérification TLA+ et expérimentons notre analyse avec plusieurs processeurs RISC-V open-source. Enfin, nous faisons progresser les études en évaluant l'impact de la génération automatique à l'aide d'une série de critères synthétiques.

Title : Formal Semantics of Hardware Compilation Framework
Keywords : Modeling of processor design, Hardware Construction Languages (HCLs), Pipeline datapath, Timing anomalies, Model checking

Abstract : Static worst-case timing analyses are used to ensure the timing deadlines required for safety-critical systems. In order to derive accurate bounds, these timing analyses require precise (micro-)architecture considerations. Usually, such micro-architecture models are constructed by hand from processor manuals. However, with the open-source hardware initiatives and high-level Hardware Description Languages (HCLs), the automatic generation of these micro-architecture models and, more specifically, the pipeline models are promoted. We propose a workflow that aims to automatically construct pipeline datapath models from processor designs described in HCLs. Our workflow is based on the Chisel/FIRRTL Hardware Compiler Framework. We build at the intermediate representation level the datapath pipeline models. Our work intends to prove the timing properties, such as the timing predictability-related properties. We rely on the formal verification as our method. The generated models are then translated into formal models and integrated into an existing model checking-based procedure for detecting timing anomalies. We use TLA+ modeling and verification language and experiment with our analysis with several open-source RISC-V processors. Finally, we advance the studies by evaluating the impact of automatic generation through a series of synthetic benchmarks.

Contents

1 Introduction **13**
- 1.1 Scientific Contributions . . . 15
- 1.2 Outline . . . 16

2 Background **17**
- 2.1 Hardware Construction Languages and their Compilation . . . 17
- 2.2 Pipelined Processors . . . 22
  - 2.2.1 Pipelining Concepts . . . 23
  - 2.2.2 HDL Designs: RISC-V Processors Overview . . . 25
- 2.3 Formal Methods Overview . . . 29
  - 2.3.1 Formal Notions . . . 30
  - 2.3.2 Model Checking Formal Verification Method . . . 32
  - 2.3.3 TLA+ Language . . . 32
- 2.4 Conclusion . . . 34

3 State of the Art **37**
- 3.1 Building Pipeline Models from Hardware Designs . . . 37
- 3.2 Semantics of High/Low-Level Hardware Languages . . . 43
- 3.3 Code Analysis Approaches . . . 48
- 3.4 Model Checking for Timing Properties . . . 51
- 3.5 Synthesis & Conclusion . . . 55
- 3.6 Problem Statements . . . 57

4 Workflow of Timing Models Derivation **59**
- 4.1 Proposed Workflow . . . 59
- 4.2 Chisel/FIRRTL High-Level HDL Designs . . . 61
  - 4.2.1 Chisel Hardware Description Language . . . 61
  - 4.2.2 FIRRTL - The Intermediate Representation . . . 64
- 4.3 Pipeline Construction in the Chisel/FIRRTL Framework . . . 65
  - 4.3.1 Choice of FIRRTL Forms . . . 65
  - 4.3.2 Chisel to FIRRTL Compilation Issues . . . 67
  - 4.3.3 Description of Chisel-Based Processor Pipelines . . . 69
- 4.4 Conclusion . . . 77

5 Register Analysis of Pipeline Designs **79**
- 5.1 Register Analysis Formalization . . . 79
  - 5.1.1 Register Analysis Algorithm . . . 81
- 5.2 Application to RISC-V Processors . . . 85
  - 5.2.1 Mono-module Datapath Pipeline Design . . . 85

5.2.2 Multi-module Datapath Pipeline Design . . . 88
5.3 Experimental Results . . . 92
5.4 Conclusion . . . 94

6 Automatic Generation of Abstract Datapath Pipeline Models **95**
6.1 Construction of Abstract Datapath Pipeline Models . . . 95
6.2 Application to RISC-V Processor Designs . . . 100
6.3 Experimental Results and Synopsis . . . 106
6.4 Conclusion . . . 111

7 Automatic Construction of Formal Models From Abstract Models **113**
7.1 Formal Modeling of Processor Pipeline Designs . . . 113
7.1.1 TLA+ Specification from Datapath Pipeline Designs . . . 114
7.1.2 Control and ISA Modeling TLA+ Specification . . . 119
7.2 Application to RISC-V Processors Case Studies . . . 121
7.3 Integration of Formal Pipeline Models in a Timing Anomaly Detection Procedure . . . 125
7.3.1 Model Checking for Timing Anomalies . . . 125
7.3.2 Experimental Results . . . 129
7.4 Conclusion . . . 134

8 Conclusion and Perspectives **137**

9 Annexe: Résumé Substanciel **141**

List of Figures

2.1 Chisel/FIRRTL hardware compilation framework. . . . 21
2.2 Transformations and passes in Chisel/FIRRTL framework. . . . 21
2.3 Classical 5-stage pipeline with forwarding. . . . 23
2.4 Pipeline diagram of Sodor 5-stage, from [1]. . . . 27
2.5 Rocket Core pipeline, from [2]. . . . 27
2.6 Fetch, Pcgen stages of Rocket core pipeline, from [2] 28
2.7 Four stages of Rocket core pipeline, from [2]. . . . 28

3.1 Structure of a VHDL-based static analysis framework, from [3]. . . . 39
3.2 VHDL timing model abstractions, from [4]. . . . 40
3.3 Chronos pipeline modeling overview. . . . 42
3.4 A control flow graph and its control dependence subgraph, from [5]. . . . 49

4.1 Formal verification of timing properties from hardware designs workflow. . . . 59
4.2 FIRRTL Abstract Syntax Tree (AST). . . . 65
4.3 Datapath pipeline module of Kyogen processor. . . . 71
4.4 Datapath pipeline modules of Rocket core. . . . 71
4.5 Datapath pipeline modules of Fuxi. . . . 74

5.1 Register analysis for mono-module datapath pipeline design. . . . 85
5.2 Datapath pipeline registers in RISC-V Mini. . . . 86
5.3 RISC-V Mini contexts. . . . 87
5.4 RISC-V Mini registers context. . . . 88
5.5 `Process Module Function` for RISC-V Mini processor. . . . 88
5.6 Register analysis for multi-module datapath pipeline design 89
5.7 Outputs context in RISC-V Sodor-3stages. . . . 91
5.8 IO interface for register precedence relation 92
5.9 Intermediate representation graph for RISC-V Sodor 3-stage 93

6.1 Simple application of min case 1 ($C1\_min$). . . . 97
6.2 Simple application of max case 1 ($C1\_max$). . . . 98
6.3 Inaccurate application of `min/max_case 1`. . . . 99
6.4 Simple application of case 2. . . . 100
6.5 Intermediate representation graph for RISC-V Sodor 5-stage 101
6.6 RISC-V Sodor 5-stage pipeline datapath model. . . . 102
6.7 Partial representation of the abstract pipeline model of the KyogenRV processor. . . . 105
6.8 Partial representation of the abstract pipeline model of the Rocket processor. . . . 106
6.9 Execution of the register assignment algorithm, i.e., Algorithm 4. . . . 108

7.1 Example of counter-intuitive timing anomalies 126

7.2 Timing anomalies intuition, from [6] 127

List of Tables

2.1 Syntax and semantic of Chisel constructs. 19

5.1 Register analysis experimental results on RISC-V processor designs. 93

6.1 Experimental results on KyogenRV processor. 104
6.2 Experimental results on Rocket chip processor. 107
6.3 Experimental results on RISC-V processor designs. 108

7.1 Statistics on TLA+ specification of RISC-V processor designs 124
7.2 Impact of stalling checks on TLA+ pipeline models. 125
7.3 Experimental results for the absence of timing anomalies for RISC-V Sodor processor. . 130
7.4 Experimental results for the absence of timing anomalies for Rocket processor. 130
7.5 Experimental results for timing anomalies detection with multiple program depth for Sodor 5-stage processor. 132
7.6 Experimental results for timing anomalies detection with multiple program depth for Rocket processor. 132
7.7 Experimental results for timing anomalies detection with multiples latencies for Sodor 5-stage processor. 133
7.8 Experimental results for timing anomalies detection with multiples latencies for Rocket processor. 133
7.9 Experimental results for timing anomalies detection with various processor designs. . 134

Dedications

I dedicate this humble work with great love and pride

To my dearest mother Khadija EL GAREH

To my dearest mother, may you find here the homage of my gratitude, which will be equal to your sacrifices and prayers for me, however great it may be. You represent the source of tenderness and the example of devotion that never ceased to encourage and pray for me. No dedication is eloquent enough to express what you deserve for all the sacrifices you have constantly made during my childhood and even adulthood. I dedicate this work to you as a token of my deep love. May God, the Almighty, preserve you and grant you health, long life, and happiness.

To my dear father Mohammad AIT BENSAID

To the good Lord and you, the man in my life, for being able to live this day. The world could compensate for all the sacrifices you made for my education and well-being so that I could devote myself to my studies. A thousand thanks for all your sacrifices, your Alas, no dedication can express my love for you. May Almighty God preserve you and grant you health, long life, and happiness.

To my brothers Noureddine, Youssef, and Zakaria

For their love and undeniable support. Words cannot express my attachment, love, and affection for you. I wish you a future full of joy, happiness, and success.

To my friends, for the moments we've shared and for your support.

To my supervisors and everyone who has helped me, I dedicate this work to them in recognition of their invaluable support.

Acknowledgment

This experience spent within CEA List and the Paris-Saclay University would never have been so fruitful without the contribution of some people I would like to thank.

I am particularly grateful to my three supervisors, Mathieu Jan, Mihail Asavoae, and Farhat Thabet, for their continuous support of my research. They ensured this project's progress and provided rigorous and professional seriousness in a friendly, relaxed climate with disconcerting virtuosity. I am very grateful for their wise advice, immense knowledge, and guidance.

I would like to thank my jury members, Dr. Stephan Merz, Prof. Steven Derrien, Prof. Jean-Luc Bechennec, Prof. Isabelle Puaut, and Prof. Laurent Pautet, for having accepted to evaluate my work and for their perceptive comments.

I also thank all my labmates; it has been great sharing the laboratory with you during these past three years.

Finally, I thank my family and friends who supported me with moral and emotional support during my thesis years.

1 - Introduction

Embedded systems are designed with specific functionalities in mind and integrated into more complex systems. Embedded systems have a hardware infrastructure (e.g., processor-based) and are capable of executing applications that address those specific functionalities. These systems are used in various application contexts, ranging from portable devices such as phones to safety-critical medical, aerospace, and transport systems. Safety-critical systems induce computing constraints, which depend on several factors. More specifically, we consider safety-critical systems as real-time embedded systems whose correctness depends not only upon their functional correctness but also upon the time in which they are performed.

The design and implementation of safety-critical systems rely on regulations and standards that state functional and non-functional constraints to be satisfied. Functional requirements are essential so that the system can operate correctly. However, these systems are also subjected to non-functional aspects such as hard real-time requirements, and their design is standardized to help identify and address potential hazardous events [7]. For example, events like missing timing deadlines would be deemed unacceptable since respecting these deadlines is mandatory for safety-critical systems.

Specialized timing analyses (i.e., worst-case) are required to derive adequate timing bounds and to characterize, in this way, the timing behavior of a system under consideration. The timing properties must be handled at different steps of the design process. Thus, the hardware designers have started to integrate these properties in the early design stages, from modeling to execution. The failure to meet deadlines and the timing vulnerabilities may be as harmful as producing inaccurate outputs or causing system damage. We characterize the timing behavior of these systems with adequate timing bounds (worst-case execution time considerations) [6] [8].

Static worst-case timing analysis [9–11] is one of the approaches used to estimate these deadlines and can compute safe and precise bounds on the timing behavior of the system. Such analyses consider the executions of an input program on an underlying architecture in order to account for accurate results. In this setting, both aspects of the program and the architecture considerations become equally important. Whereas the program-level infrastructure required by the WCET analysis relies on the control-flow graph (i.e., the input program is the binary), at the architecture level, the WCET analysis infrastructure is more diverse. For example, the same control-flow graph is used for cache analyses [12] whereas more specialized, cycle-accurate models are necessary for pipeline analyses [13].

Thus, the need to model the design architecture is paramount to verify

the required timing constraints. Several static timing analyzers exist, for example, the industrial-strength tool aiT [14] and the academic WCET analyzers Otawa [9], Heptane [10] and Chronos [11]. All these propose handcrafted models of (micro)-architecture (i.e., caches and pipelines), relying on static analysis to characterize their timing behaviors, and validated against hardware simulators for conformance (e.g., for aiT [15] or Chronos [16]). A closer code inspection of their pipeline models exposes their common attributes. First, these pipelines present a "flat" structure. More precisely, the pipeline stages are represented with simple state configurations (i.e., a single variable), reducing a pipeline stage to an identification attribute. The pipeline components are separated into smaller units with inner states. This separation reduced the complexity and made the implementation easier. Second, these models focus on how an instruction progresses through the pipeline, not the actual instruction semantics. Indeed, the correctness of program execution is assumed in the context of static WCET analysis, and the worst-case upper bounds are defined through the execution of the input program on the pipeline model. Furthermore, various models are built from low-level hardware designs such as VHDL and Verilog [3].

However, the manufacture and production of these hardware designs are evaluated according to several criteria, and the design time is one of the primary criteria. Furthermore, the low-level hardware description languages describe in more detail the designs and reduce the production speed in exchange. Since their micro-architecture models are usually huge, generating pipeline analysis that covers the timing behavior of the design would increase the state space, making the WCET determination a challenging task. Thus, we need to opt for high-level abstraction languages and generate these timing models automatically. With an automatic procedure, we speed up the process and reduce the model's size by pruning out all parts that do not contribute to the timing behavior. Moreover, we avoid losing information, and we guarantee that the timing analysis of the design is feasible with a precise computation of upper bounds. So, developing such models at a proper abstraction level is essential to obtain correct and precise WCET bounds.

Automatically deriving architecture models [3, 4] directly from the code of hardware designs can also be possible. Unsurprisingly, the accuracy of the resulting architecture models depends on how the hardware design is coded. For example, in the case of a pipeline design, a modular code with a clear separation between datapath and control facilitates the construction of the architecture model. Recent trends in hardware design led to more processor code being made available, i.e., open-source. Thus, it can be used to generate models, replacing the standard manual reference, where only specific design details are provided. This progress is facilitated by the emergence of open hardware initiatives [17]. Such initiatives propose software-like development workflows, from complex high-level Hardware Description Languages

(HDLs) [18] down to circuits while using sophisticated compilation chains.

High-level hardware description languages (e.g., Chisel [19] or SpinalHDL [20]) provide new features to hardware designers while, at the same time, speeding up development by being able to generate fast simulations of the design. While low-level, mainstream languages like VHDL and Verilog are still almost ubiquitous in the industry, their age and original intention put them behind current high-level languages regarding productivity and flexibility. High-level design languages come with their associated compilation framework, enabling the use of highly parameterized generators and advanced module systems to facilitate the hardware design [18] [21] [20]. These compilation chains come with configurable optimizations and a pass infrastructure, which facilitate the analysis of hardware designs. For instance, when targeting timing predictability, a transformation would mainly focus on the sequential logic to generate pipeline-level models. The main goal of our thesis work is to automatically generate pipeline models from high-level hardware processor designs so as to prove timing properties.

To prove these requirements of safety-critical systems, various methods can be used, from testing to formal verification, offering different degrees of confidence. Testing is a standard method used for validation, but it is not exhaustive. Formal verification is another method used for validation; it could be exhaustive but requires building formal models. We adopt formal verification as our approach, with the goal of generating formal micro-architecture models as pipeline models from open-source processor designs. Then, we can formally verify the properties related to timing predictability, such as the detection of timing anomalies [6, 22].

1.1 . Scientific Contributions

Considering the context and the main motivations presented in the previous section, our proposed work leads us to define two main goals to get around the above-mentioned motivations. The first goal is the automatic generation of pipeline models. The second goal is to integrate these pipeline models into a formal framework for the verification of timing properties, for example, the detection of timing anomalies. As a result, we summarize these goals in four main contributions.

As a first contribution, we propose a **general workflow** to achieve our goals. The workflow targets the automatic generation of pipeline models for the formal verification of timing properties. We specify each component with a highlight of the hardware compilation framework as the first brick of the workflow.

The second contribution concerns the **register analysis** and the construction of data structure representation of the abstract datapath pipeline models. In this context, we rely on the high-level hardware processor designs de-

veloped with Chisel language [19] and its associated compilation framework Chisel/FIRRTL [23] that provides the possibility to integrate passes. Thus, the problem of deriving convenient datapath pipeline models is backed by a comprehensive hardware compilation infrastructure. The approach proposed to derive register analysis is applied to various processor designs.

The third contribution concerns the **pipeline datapath construction** based on the results of the register analysis. In this phase, the registers are assigned to pipeline stages through several algorithms. The aim is to build the abstract datapath models as a succession of pipeline registers assigned in their identified pipeline stages. We report our proposed approach and evaluate its effectiveness in several case studies as Chisel-based RISC-V processors.

Finally, the last contribution intends to generate automatically the required formal models to prove formally the timing predictability properties and then to integrate them into a procedure for the detection of timing anomalies. We automatically generate the **formal datapath pipeline models**. Then we manually specify the control signals to guide the pipeline execution. In this direction, we combine both the software (ISA representation) with the hardware (pipeline models) aspects. Therefore, we formally verify the absence/presence of timing anomalies on the resulting models with the procedure presented in [6, 22]. The objective is to evaluate the impact of semi-automatically generated models through a series of synthetic benchmarks.

1.2 . Outline

The remainder of this manuscript is organized as follows:

- **Chapter 2:** We present the background of our work, including the hardware construction languages, the RISC-V processor designs used to evaluate our work and the formal verification framework. We also highlight our chosen hardware language and its compilation framework.
- **Chapter 3:** We review state of the art works that build the micro-architecture models from hardware designs. We detail each axis of our contributions. We introduce then the problem statement of our work.
- **Chapter 4:** We introduce our general workflow that aims to verify timing properties on hardware designs, formally.
- **Chapter 5:** We describe in this chapter the register analysis as well as its evaluation.
- **Chapter 6:** We describe the pipeline construction algorithm through an assignment of registers to pipeline stages. We report the application and the experimental results of several processor designs.
- **Chapter 7:** We present the generation of formal models followed by the integration of these formal models into the procedure to detect timing anomalies. We rely on several benchmarks to evaluate our results.
- **Chapter 8:** We describe conclusions and some perspectives of our work.

2 - Background

In this chapter, we introduce the background concepts on which this thesis work is based. In Section 2.1, we present a general background on hardware construction languages (HCLs) and their compiler frameworks, then elaborate on our chosen hardware language Chisel, and its compilation framework Chisel/FIRRTL. In Section 2.2, we continue with an overview of the processor designs considered as case studies in this work. Then, Section 2.3 introduces notions related to formal verification methods and tools.

All these elements introduced in this chapter allow us to emphasize the problems addressed in the next chapters precisely.

2.1 . Hardware Construction Languages and their Compilation

Overview of HCLs. When we discuss about hardware designs, there is a need to find a method to ease communication and understand the functionality of each design component. As such, the hardware designs are developed with hardware description languages (HDLs) to present the structure and functionality at different abstraction levels. When hardware designers intended to increase design productivity and efficiency with shorter design cycles, they relied on hardware generators and design reuse in particular libraries reuse, as in software programming. Thus, hardware designers use high-level languages called hardware construction languages (HCLs), which embed HDL-like hardware primitives in existing programming languages.

These languages enable the use of software programming features such as object orientation, polymorphism, and higher-order functions, which encourage and enable code reuse. Furthermore, these expressive language features make designs more parameterizable and modular and enable the use of domain-specific language constructs, and advanced module systems to facilitate the hardware design [18, 20, 21]. These languages are supported by hardware compiler frameworks (HCFs) which can put hardware development on an important evolution by enabling new hardware libraries. Furthermore, HCFs transform a high-level purpose code into an RTL level by formalizing various transformations. These transformation passes enable the reuse and optimization of the HDL (Verilog or VHDL) code they generate for later use as input in classical commercial (FPGA or ASIC) hardware design flows. As in software compilers, hardware designers can also insert specific transformation passes in those compilation chains to manipulate the designs. This feature enables to deploy, within these chains, analyses to construct abstract processor models automatically. For instance, when targeting timing predictability, a transformation would mainly focus on the sequential logic to generate pipeline-level

models.

Several hardware construction languages are introduced and developed for hardware designs. Usually, they are based on software programming languages such as C++, Python, and Scala. Chisel [19], BlueSpec [24], SpinalHDL [20], and DFiant [25] are the most modern hardware construction languages that can be seen as domain-specific languages, where they leverage host language ideas and software engineering techniques. We focus in our work on an HCL called Chisel, supported by the compilation framework Chisel/FIRRTL.

Chisel/FIRRTL Hardware Compiler Framework. Chisel [19] is a hardware construction language (HCL) developed at UC Berkeley, embedded in the Scala language and thus, implemented as a package of Scala class libraries. Chisel provides powerful constructs for writing hardware generators. We take, as examples, parameterization and higher-order functions.

Table 2.1 summarizes the primary Chisel constructs, specifying their syntax and semantics. We distinguish between basic elements such as types, operators, etc. state elements corresponding to registers and memory components, and finally high-level constructs such as interfaces. Chisel provides three data types: `Bits`, `UInt`, and `SInt`. All three types represent vectors of bits. Here is the definition of different such types: a plain 6-bit, an 6-bit unsigned integer, and a 10-bit signed integer, with `W` specifying the width: `Bits(6.W)`, `UInt(6.W)`, `SInt(10.W)`.

Interfaces are presented in Chisel by `DecoupledIO` and `ValidIO` language constructs. Chisel defines the `DecoupledIO` bundle in the following way: `bits` is used for the data and is parameterized with the data type. `ValidIO` is similar to `DecoupledIO` except that it only has `valid` and `bits` fields.

Listing 2.1 – Parameterization in Chisel.

```
class DecoupledIO [T <: Data](gen: T) extends Bundle {
   val ready = Input(Bool())
   val valid = Output(Bool())
   val bits = Output(gen)
}
class ValidIO [T <: Data](gen: T) extends Bundle {
   val valid = Output(Bool())
   val bits = Output(gen)
}
```

We detail other constructs with particular processor designs in Section 4.2.

The Listing 2.2 illustrates the parameterization features of the Chisel language. Chisel uses parameterized types to specify generic functions and classes. The function `Mux`, line 11 in Listing 2.2 describes the parameterized function and semantically it defines a generic multiplexer function. We define this function as taking a boolean condition, `con`, and `alt` arguments (corresponding to

Table 2.1 – Syntax and semantic of Chisel constructs.

Chisel_Constructs	*Syntax*	*Semantics*
`Types`	Bool(), UInt(), SInt()	Boolean, Unsigned/Signed integers
`Aggregate types`	class MyBundle extends Bundle { val a = Bool () val b = UInt ()}	Define Data types indexed by name
	Vec(elts:Iterable[Data])	An indexable vector of Data types
`Operators`	!x, x̃, x + y	Logical NOT, Bitwise NOT, Addition
`Memory`	Mem(n:Int, out:Data)	Addressable memory with n as a depth
`Registers`	Reg(type)	Create type register
	RegInit()	Register with initialization value
	RegNext()	Register updated every clock cycle
`Wires`	val x = UInt()	Allocate wire of type UInt
	x := y	Assign wire y to x
	x <> y	Bulk connect x and y
`Interfaces`	DecoupledIO(gen: Data)	gen: Chisel Data to wrap with a ready valid interface. **Interface:** (in) .ready: ready Bool (out) .valid: valid Bool (out) .bits: data
	ValidIO(gen: Data)	Wrap gen with a valid interface. **Interface:** (out) .valid: valid Bool (out) .bits: data

`then` and `else` expressions) of type T which is required to be a subclass of Bits. The assignment to `mux1`, in line 15, specifies the parameter values.

Like parameterized functions, we can also parameterize classes to make them more reusable. For instance, we can generalize the `Adder` class, lines 3-12 in Listing 2.2 to add any type T which is a generic type defined in the package `chisel3.Data`. We define two instances of this class `add1` and `add2`, lines 13-14, to take two arguments `genIn` and, `genOut` of type T.

Listing 2.2 – Parameterization in Chisel, inspired from [26].

```
1  import chisel3.Data
2  ...
3  class Adder[T <: Data](genIn: T, genOut: T) extends Module {
4      val io = IO(new Bundle {
5          val a = Input(genIn)
6          val b = Input(genIn)
7          val out = Output(genOut)
8          ...
9      })
10     io.out := io.a + io.b
11     def Mux[T <: Bits](c: Bool, con: T, alt: T): T = { ... }
12 }
13 val add1 = new Adder(UInt(4.W), UInt(6.W))
14 val add2 = new Adder(SInt(4.W), SInt(6.W))
15 val mux1 = add1.Mux(true.B, UInt(10), UInt(11))
16 ...
```

Listing 2.3 describes high-order functions in Chisel that take functions as arguments. These are powerful constructs that encapsulate a general calculation model, allowing one to concentrate on the application logic instead of the control flow. These functions include the `map` and `reduce` functions. Function `map`, line 2 in Listing 2.3, invokes a function (successor) on each element of the list and returns a list of the function's return value. Function `reduce`, line 3 in Listing 2.3, takes two arguments: the first is the current accumulation and the second represents the list element. These are given by the two underscores in the parentheses. It returns a value with a type of the list elements, unlike `map`, which returns a list.

Listing 2.3 – Higher-order functions in Chisel.

```
1 ...
2 List(1, 2, 3, 4).map(_ + 1) /* return List(2,3,4,5) */
3 List(1, 2, 3, 4).reduce(_ + _) /* return 10 */
4 ...
```

Figure 2.1 illustrates the Chisel/FIRRTL hardware compiler framework. Once a Chisel design is compiled, a Verilog file is generated via an intermediate representation FIRRTL [23], which can then be processed with any standard RTL design flow.

An important part of any compiler is its intermediate representation (IR), upon which all transformations operate. FIRRTL transformations always take as input and produce a well-defined AST circuit and are easily connected one after the other. These transformation passes provide simplification, optimization, analysis, specialization, and instrumentation and the resulting representation can either be simulated directly or passed to one of many Verilog back-

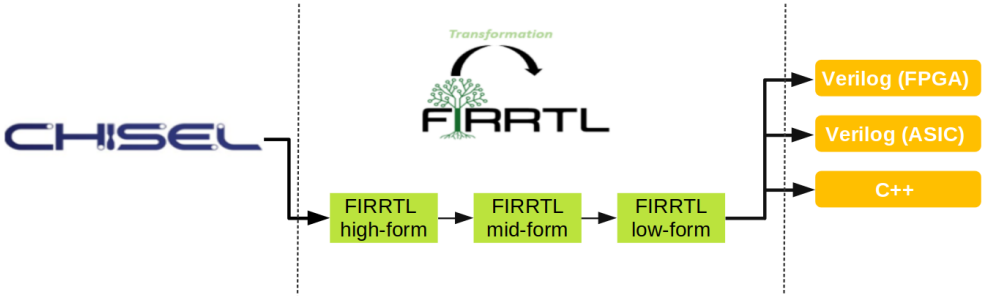

Figure 2.1 – Chisel/FIRRTL hardware compilation framework.

ends tailored for simulators, FPGAs, or ASIC technology processes.

Figure 2.2 describes the Chisel/FIRRTL compiler framework, where the Chisel design is compiled into FIRRTL representations through simplification transformations. Simplification transformations take a FIRRTL circuit and simplify it to a lower form. First, Chisel is compiled into FIRRTL high form through various passes, which generally check and resolve the high form constructions, for instance, the `CheckTypes` and `CheckWidths` passes, where the types and components widths must be checked to verify the coherence between Chisel constructs. The `InferBinaryPoints` pass verifies the type match in the assignment constructs. Furthermore, for all circuit components declared with unspecified width or precision, the FIRRTL compiler, through the `InferWidths` pass, will attempt to infer the strictest possible width value out of all possible values of its incoming connections.

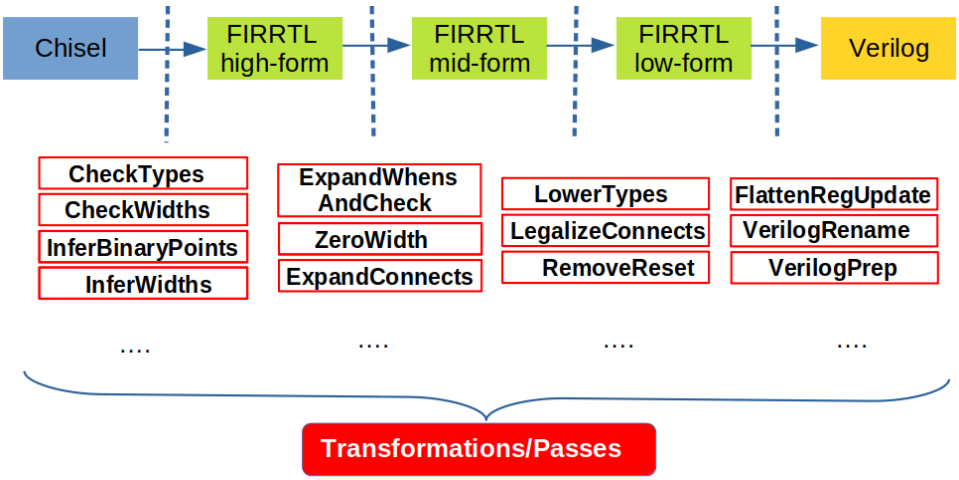

Figure 2.2 – Transformations and passes in Chisel/FIRRTL framework.

Afterward, transformations go through two simplifications, from high form to mid-form and from mid-form to low form. In the mid-form, the conditional statements (`when`) are not used; thus, the pass `ExpandWhensAndCheck` addresses this point. Also, all widths must be explicitly defined, and fields in

connections must be detailed using the `ExpandConnects` pass. The low form gives a direct correspondence to a circuit netlist. For instance, the partial connect statements are not used in this form, thus the pass `LegalizeConnects` is used. Moreover, the synchronous `Reset` is removed with the `RemoveReset` pass. Finally, the low form is directly compiled into Verilog. Therefore, various passes are implemented to make changes to the FIRRTL AST to make Verilog emission easier. We mention the `FlattenRegUpdate` pass, which flattens register updates into a single expression, and `VerilogPrep` pass, which adds wires to connect port interfaces between instances.

Furthermore, writing a transformation saves effort for all future uses, reducing the cost of the development over time. By making transformations easy to write and integrate into a compiler framework, the upfront development cost of a transformation is reduced, and the number of interesting automatable tasks increases. Writing a FIRRTL pass usually requires writing functions that walk the FIRRTL data structure. The intermediate representation (IR) of FIRRTL is a tree, where each IR node can have the children nodes. The recursive walk collects information or replaces IR nodes with new IR nodes.

Next, in Listing 2.4, we present an example of the transformation that expresses a recursive walk to collect the names of every register declared in the design. We apply a custom map function for each node and then to the subset of children whose node type matches the function's input and return node type, line 7 in Listing 2.4. When the node is a register, we collect its name into a list structure.

Listing 2.4 – Simple pass in Chisel/FIRRTL compiler framework.

```
1  ...
2  def walkMod(m: DefModule): DefModule = { m.map(walkStmt) }
3  def walkStmt(s: Statement): Statement = {
4   def fct(regNames: mutable[String])(s: Statement): Statement=
5        s match {
6          case r: DefRegister => regNames += r.name; r
7          case _ => s map fct(regNames)
8        }
9  }
10 ...
```

2.2 . Pipelined Processors

We present next the various pipelined hardware designs evaluated and used as case studies in this work. We start with an overview of the pipeline features that are considered in the hardware designs and which impact our analysis, in Section 2.2.1. Then, we present the primary use case studies of hardware designs, in Section 2.2.2.

2.2.1 . Pipelining Concepts

Pipelining is a process of arrangement of hardware elements of the processor such that its overall performance is increased. The process of executing one instruction can be divided into several sequential sub-operations, with each sub-operation corresponding to a pipeline stage. The processor can execute instructions in the order that they appear in the program, as is the case of in-order pipelines. The pipeline stages for the classic in-order pipeline, shown in Figure 2.3, are:

- Instruction Fetch (IF): In this stage, an instruction is read from the instruction memory. The address of memory is contained in the program counter. The fetch stage is the first stage to initiate the instructions traversal of the pipeline. We can notice that the instruction may be retrieved from the memory or directly from the instruction cache.
- Instruction Decode (ID): In this stage, an instruction is decoded, and control signals and combinatorial logic are used for the opcode and operands according to each instruction type.
- Execute (EX): The ALU (arithmetic logic unit) operations are performed at this stage. It takes the two operands to perform the ALU operation. In this stage, the branch target can also be computed.
- Memory Access (MEM): In this stage, the memory accesses are performed with read and write to/from memory instructions. The stage is also responsible for stalls due to cache misses.
- Write-Back (WB): In this stage, the computed value is written back to the register file. It writes the results of the arithmetic operations or the data read from memory.

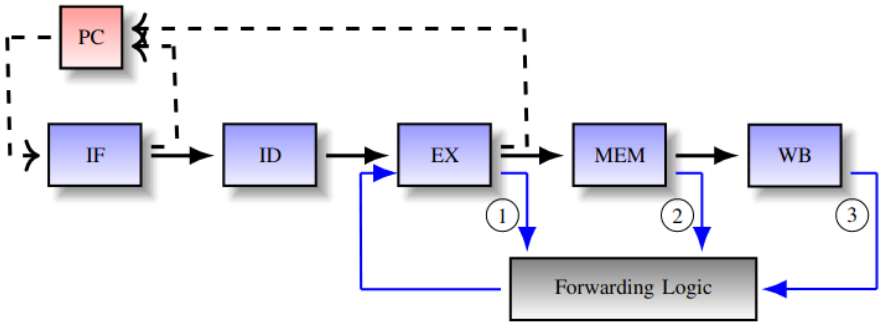

Figure 2.3 – Classical 5-stage pipeline with forwarding.

A pipeline consists of the datapath and control path. The datapath can be defined as the path that the instructions must fulfill within the processor to finish their execution. Different instructions may need different datapaths and an instruction generally requires more execution time to fulfill a longer

path within the datapath. The control path ensures the correct transmission of data and controls the execution status of the instructions at each pipeline stage. The control path defines the pipeline status by identifying stalling and forwarding mechanisms and establishes signals for instructions decoding.

Figure 2.3 shows the stages of a standard in-order 5-stage pipeline, i.e., from the Instruction Fetch (*IF*) stage to the register Write-Back (*WB*) stage. The pipeline control, represented with dotted lines, captures how instructions advance (i.e., the next *PC* value) through the pipeline. The stalling logic is, however, not explicitly represented, but it can be inserted in any pipeline stage. The datapath, presented with solid lines, considers the usual left-to-right advancement through the pipeline stages, with optimization in the form of register forwarding, represented with blue lines. The forwarding mechanism depends on the architecture and in which stage an instruction requires/produces the dependencies. The processors employed in our work and whose pipelines strictly adhere to this representation are shown in Fig. 2.3.

The pipeline execution of the processors divides the execution of instructions into several steps without affecting the performance. In contrast, the step duration can be reduced by increasing the number of steps and therefore, reducing the processor's cycle time. However, there are various cases where pipeline performance is degraded, called pipeline hazards. Hazards are pipelined execution problems caused when an instruction is unable to be executed in its designated cycles. There are three types of hazards: data, control, and structural hazards.

Structural hazards result from sharing resources between instructions. They occur when more than one instruction needs to use the same datapath resource simultaneously. There are two main causes of structural hazards:

- Register File: The simultaneous access to the register file leads to structural hazards. The register file is accessed both during ID when it is read and during WB when it is written. We can solve this by having separate read and write ports. To account for reads and writes to the same register, processors usually write to the register during the first half of the clock cycle and read from it during the second half. This is also known as double pumping.
- Memory: Memory is accessed for both instructions and data. The IF and MEM stages both transfer data over the memory bus. Having a separate instruction memory and data memory solves this hazard.

Structural hazards can always be resolved by adding more hardware.

Control hazards could occur when instructions of type jump and branch are in execution. If a dynamic branch instruction occurs, either as a conditional branch or an indirect jump, the fetch stage (PC) cannot continue to fetch the next instruction (next sequential PC) after the branch, but the computation result of the argument to the branch completed in the EX stage. Thus, we should stall the pipeline for control hazards, but this decreases performance.

Branch prediction attempts to resolve a branch hazard by predicting which path to take.

Data hazards are due to data dependencies between instructions: an instruction requires an unavailable result of the previous instruction. Existing solutions to handle data hazards include forwarding mechanisms [27], code reordering and pipeline stalling. There are three situations in which a data hazard can occur:

- Write After Read (WAR): An anti-dependency hazards occur if an instruction writes a register that is read by a previous instruction. Here it must be guaranteed that the write occurs only after the first instruction has read the register. This type of hazard cannot occur in in-order processors since instructions are always executed in the primary fixed order.
- Write After Write (WAW): an output dependency, hazards occur if a subsequent instruction writes the same register as a previous one. It must be guaranteed that only the last write is performed to the register file. Again, for in-order pipelines with only a single write-back stage, this hazard cannot occur.
- Read After Write (RAW): A true dependency hazards occur if a subsequent instruction reads a register that is written by a previous instruction. This dependency is resolved by the forwarding mechanism to avoid additional delays.

Forwarding is an efficient method in solving the data dependencies between instructions, more precisely when a destination operand of an instruction is the source operand(s) of a subsequent instruction(s). Forwarding is implemented using combinational logic units (i.e., multiplexers and comparators) to test these dependencies. For instance, it uses multiplexers and structural conditions in order to test if there is a data dependence between the instruction's operands. Furthermore, the forwarding mechanism presents several possible paths. It depends on various criteria; the data dependence between instructions is the first. However, the addition of wires also depends on the architecture of the processors, in particular, the stage in which the instruction should be executed in order to produce/require the data.

A complete forwarding consists of several paths. The main possible cases include three paths. There are thus 3 forwarding paths illustrated in Figure 2.3 ① - ③, from the pipeline stages Execute (*EX*), Memory (*MEM*) and Write-Back (*WB*) to *EX* respectively.

2.2.2 . HDL Designs: RISC-V Processors Overview

RISC-V is a modern, open-source Instruction Set Architecture (ISA) that has gained significant momentum in recent years. Moreover, RISC-V is designed from the ground up to enable the integration of custom instruction set extensions in order to build highly application-specific solutions. Most Chisel-based processor designs are built to support RISC-V ISA. We present next several

hardware processor designs used to evaluate our work. They are developed in the Chisel language and compiled with the Chisel/FIRRTL hardware compilation framework. These processors' pipeline structure ranges from 3 to 6 pipeline stages. We present RISC-V Sodor [1], RISC-V Mini [28], KyogenRV [29], Rocket [2] and Fuxi [30] processors.

RISC-V Mini. RISC-V Mini [28] proposes a simple three-stage pipeline designed to serve as an initial test case when designing the last versions of Chisel. RISC-V-mini is implemented using Chisel, hardware design language, version '3.5'. It has been a crucial example in various project developments, including the languages Chisel3 and FIRRTL as well as their respective simulation and verification methodologies. It implements RV32I of the Base Integer User-level ISA Version 2.0. It also consists of an instruction cache and a data cache.

RISC-V Sodor. Sodor [1] is a family of processors coming with different pipeline depths. We consider its 3 and 5 stages versions in our work. It is an open-source RISC-V project for educational purposes as well as to enable a broad range of developers to learn and utilize RISC-V ISA quickly. It contains five 32-bit RISC-V CPUs to demonstrate simple RISC-V ISA written in Chisel. These CPUs are named from RV32-1-stage to RV32-5-stage. Furthermore, each core has its features. For example, the RV32-1-stage is used essentially as an ISA simulator, and the RV32-2-stage is mainly employed to demonstrate pipelining in Chisel. RV32-3-stage uses sequential memory, while RV32-5-stage can toggle between fully bypassed or fully interlocked. Figure 2.4 illustrates the RV32-5-stage of Sodor processor design. Its pipeline is a classic 5-stage pipeline where instructions pass through the following stages: IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MM (Memory), and WB (Write-Back).

KyogenRV. KyogenRV [29] is also an open-source five-stage pipeline processor targeting Intel FPGAs and developed for academic purposes. It is written in Chisel version '3.5'. It implements RV32I of the User-level ISA Version 2.2.

Rocket. Rocket is an in-order scalar processor originally developed at UC Berkeley, initially designed for ASICs but can be built for an FPGA and features OS support. The Rocket core is used as a component within the Rocket Chip SoC generator which is a Scala program that invokes the Chisel compiler in order to emit RTL describing a complete SoC. Rocket core is based on the RV64GC RISC-V Instruction Set Architecture (ISA) [31], and is written in Chisel language, version '3.5'. The Rocket core is a 5-stage pipeline, however, it is sometimes presented as a 6-stage pipeline with a separate stage `pcgen`, implementing the branch prediction. Figure 2.5 describes the Rocket

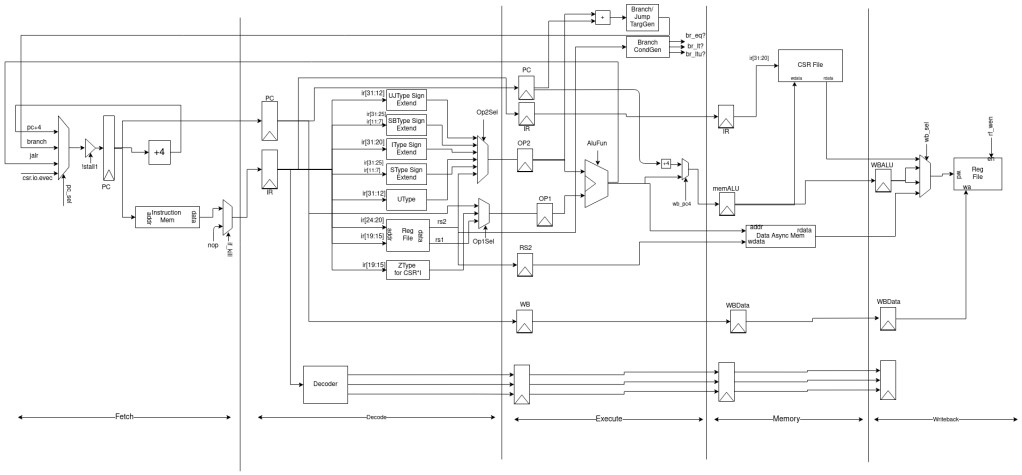

Figure 2.4 – Pipeline diagram of Sodor 5-stage, from [1].

core pipeline.

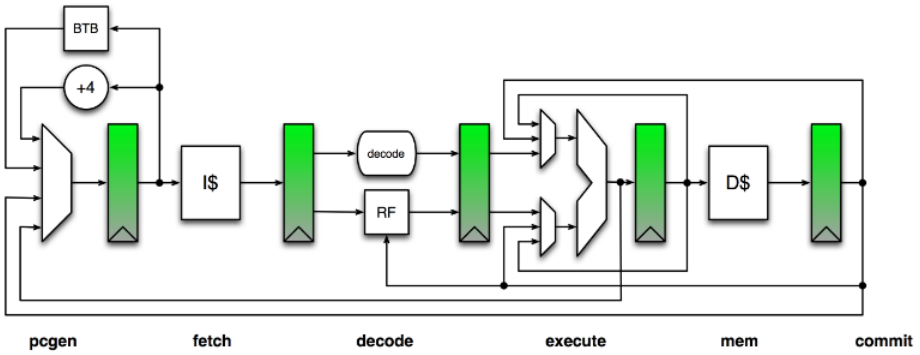

Figure 2.5 – Rocket Core pipeline, from [2].

Rocket core contains L1 data and instruction caches, integer ALU and an optional floating-point unit (FPU). The Rocket core is written in Chisel language and arranged into modules (Core, Frontend, BTB, etc.). The first two stages, Pcgen, and Fetch, are implemented in the Frontend module, while the remaining four pipeline stages are contained within the core module.

Figure 2.6 describes the Fetch and Pcgen stages in detail. It includes a MMU that supports page-based virtual memory with a translation lookaside buffer (TLB), a non-blocking data cache, and a front-end with branch prediction which is provided by a branch target buffer (BTB), branch history table (BHT), and a return address stack (RAS). While Figure 2.7 presents the Decode, Execute, Memory, and Write-Back stages. The core includes the optional FPU, the configurable functional unit pipelines, and the privileged architecture implementation (i.e., the control and status register file). The connection between these two parts is described in more detail in Section 4.3.3.

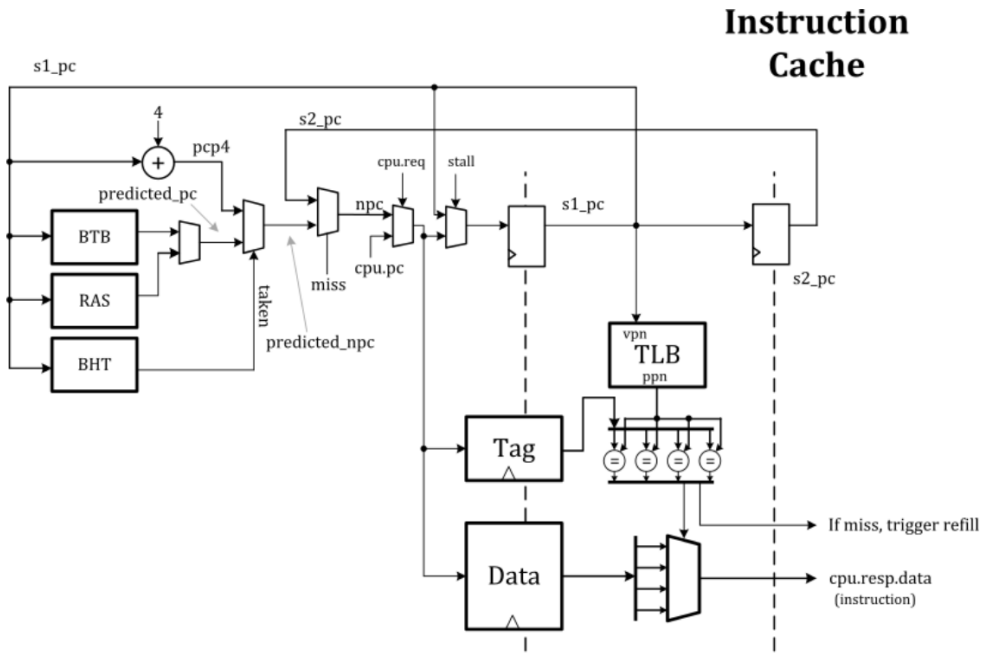

Figure 2.6 – Fetch, Pcgen stages of Rocket core pipeline, from [2]

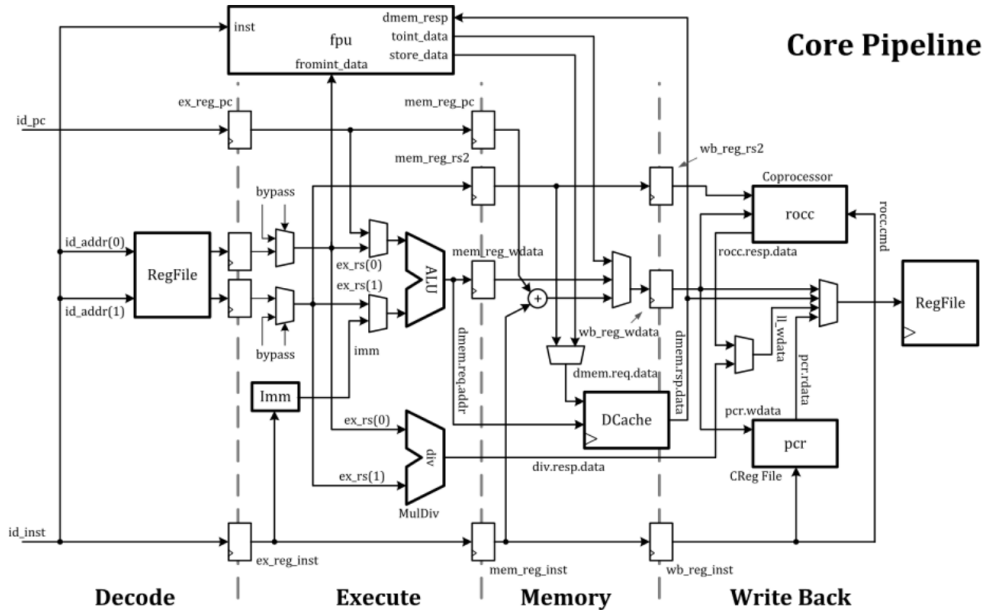

Figure 2.7 – Four stages of Rocket core pipeline, from [2].

Fuxi. Fuxi is a 32-bit pipelined RISC-V processor designed for running simple operating systems or bare-metal software. Fuxi implements RV32I ISA and is implemented in Chisel language, version '3.2'.

2.3 . Formal Methods Overview

The process of modeling and verification of properties of hardware designs considers two main phases. The first phase focuses on creating an abstract model and validating the abstraction. The second concerns verifying the properties of the abstract model. System modeling is carried out taking into account the level of abstraction required for properties to verify. Various abstract models can exist depending on the type of targeted properties and the level of granularity. Therefore, the abstract model must be detailed enough to capture the relevant behavior of the system and should be done with a high level of confidence.

Formal methods are used to ensure a high level of confidence in a system behavior. More precisely, formal methods are used to specify the systems by building an abstract representation of the real system, which focuses on the requirements that the system should satisfy. The formal modeling describes the system using mathematical and logical theories. Then, we deploy formal verification on the formal model to prove the target properties.

Several techniques are employed to verify properties, starting from testing to simulations before achieving now the formal proving. Both testing and simulation are widely used in practice, more precisely in industrial applications, and their use has proven to be very useful. However, it is usually not possible to simulate or test all behaviors of a given system. Thus, we can rely on formal verification to address this behavioral coverage issue and prove the absence/presence of errors.

Two types of proof techniques exist: deductive, e.g., theorem proving [32] and algorithmic, e.g., model checking [33] [34]. Theorem proving permits to establish the system's properties through mathematical reasoning. The theorem-proving techniques, while expressive, have limitations as they require extensive manual efforts in order to build proofs. Model checking permits to verify a system's properties by exploring all possible behaviors of a specification of this system with a limitation to face state explosion. We introduce next general formal notions before focusing on model checking, which we use to prove timing predictability-related properties over the hardware designs.

2.3.1 . Formal Notions

We present next some notions regarding formal modeling and verification.

Definition 1 (values, variables). *We denote by Val a set of values, where a value is a data item and by Var a set of variables.*

A logic consists of a set of rules for manipulating formulas. We need to define the semantics to understand what the formulas and their manipulation mean. The semantics of our mathematical logic is defined in terms of state using the notation from [35].

Definition 2 (state). *We define a state s as a mapping from state variables Var to values Val, and we denote this by $s(x) = v$ where $x \in Var$ and $V \in Val$. The set of all states named the state space is denoted as $\mathcal{S}t$.*

Definition 3 (transition system (TS)). *A transition system is a tuple $(\mathcal{S}t_t, I, \rightarrow)$, where:*

- *St_t is the set of states,*
- *$I \subseteq \mathcal{S}t_t$ is the set of initial states,*
- *$\rightarrow \subseteq \mathcal{S}t_t \times \mathcal{S}t_t$ is the transition relation, also called the next-state relation.*

The choice of states of a transition system, which describes how to pass from a state to another, depends on how much of that behavior needs to be observed in order to analyze a particular property of interest. The initial states correspond to the starting point on which we observe the system, and the next-relation includes all other possible transitions.

Definition 4 (execution in a TS). *We define the execution (also called behavior) of a transition system as a sequence of states $(s_1, s_2, s_3, ...) \in \mathcal{S}t_t \times \mathcal{S}t_t \times \mathcal{S}t_t \ldots$ such that two successive states are in the transition relation: $\forall i \in \mathbb{N}*, (s_i, s_{i+1}) \in \rightarrow$. It is an* `initial` *execution if $s_1 \in I$.*

The diameter of a TS is the least number of steps to reach all reachable states (i.e., the states that belong to at least one initial execution).

Definition 5 (labelled transition system (LTS)). *We define a labelled transition system, the tuple $(\mathcal{S}t_t, I, \Lambda, \rightarrow_\Lambda)$, where:*

- *St_t is the set of states,*
- *$I \subseteq \mathcal{S}t_t$ is the set of initial states,*
- *Λ is a set of labels,*
- *$\rightarrow_\Lambda$ is the transition relation that includes the set of labelled transitions.*

Executions of an LTS are of the form $(s_1, a_1, s_2, a_2, s_3, \ldots)$, where the successive labelled transitions are in the transition relation: $\forall i \in \mathbb{N}*, (s_i, a_i, s_{i+1}) \in \rightarrow_\Lambda$.

Definition 6 (state function, predicate). *We define state function as a non-boolean expression which maps states to constants.*
We define a state predicate as a boolean expression which includes variables and constant symbols.

States and new states are linked with a relation called action.

Definition 7 (action). *We define an action as a boolean-valued expression between unprimed variables and primed variables. We say that action A is true on step $s \Rightarrow t$, or that $s \Rightarrow t$ is an A step, if A assigns the value true to $s \Rightarrow t$*

As an example, $y = x' + 1$ asserts that the value of y in a state is one greater than the value of x in the new state.

Definition 8 (validity). *A validity of an action A, written as $\models A$ is that every step is an A step. Formally: $\models A \triangleq \forall s, t \in \mathcal{S}t : s[\![A]\!]t$.*

It is common to think of a system's execution as a series of actions. Thus reasoning about systems means reasoning about state sequences. Temporal logic is responsible for this type of reasoning.

Definition 9 (temporal formulas). *A temporal formula is made from elementary formulas (predicates and transitions) using boolean operators and the unary operator $\Box$ (read always). For example, if E1 and E2 are elementary formulas, then $\neg E1 \wedge \Box(\neg E2)$ and $\Box(E1 \Rightarrow \Box(E1 \vee E2))$ are temporal formulas. Moreover, for any temporal formula F, let $\Diamond F$ (read eventually) be defined by $\Diamond F \triangleq \neg\Box\neg F$*

A linear-time property is a requirement imposed on the traces (observable behaviors occurring during system runs) of a transition system, denoted by example with the set Traces(TS). In order to express linear-time properties, we can adopt specialized temporal logic, for instance, Linear Temporal Logic (LTL) or one of its variants. LTL formulas are expressed on infinite sequences of states where each point in time (state) has a unique successor based on a linear-time perspective. We are interested in a class of LTL properties, which are called invariants.

Definition 10 (linear-time property (LT)). *We define a linear-time property over the set of atomic predicates denoted by AP, as a subset of $(\mathbb{P}(AP))^\omega$, i.e., of the set of sequences whose elements are in $\mathbb{P}(AP)$.*

Let $P = \{(A_1, A_2, A_3, ...) \in (\mathbb{P}(AP))^\omega\}$ is a linear-time property over AP.

Definition 11 (invariant). *A (linear-time) property is an invariant if there exists a state predicate φ such that $P = \{(A_1, A_2, A_3, ...) \in (\mathbb{P}(AP))^\omega | \forall i \in \mathbb{N}*, A_i \models \varphi\}$. Such a state predicate φ is called an invariant (condition).*

2.3.2 . Model Checking Formal Verification Method

Model checking [33] is an automated approach used to verify that a model of the system satisfies a specification expressed as a property. The property defines the requirements requested for the expected behavior of the system. Model checking verifies by exploring the model's states allowed by the specification. Achieving the system abstraction and specification is a crucial step that may require system mastery and expertise in the methods used.

Once the system is modeled and the property is also defined, an investigation of whether the model satisfies the specification is performed, by exploring all possible executions of the system from its initial states. Then, the model checking automatically generates a counter-example when the property is not satisfied as an execution trace starting from the initial state to the state violating the invariants. Thus, the model-checking approach is performed on two main phases (modeling and specification verification). Two categories of model checking exist, explicit model checking [36, 37] and symbolic model checking [38, 39]. [36] contains a description on explicit model checking as TLC.

Explicit Model Checking. Explicit model checking is often based on a state space exploration, where progress is made one state at a time, and each processed reachable state of the system is represented in memory explicitly. Explicit state model checking performs a depth-first or breadth-first search through the system's states, from a state to its successors. Therefore, all the states participating in the search are reachable from the initial state, which facilitates to evaluate the number of states encountered during the verification. The main drawback of explicit model checking is state space explosion: time and memory required to verify the system grows linearly with the number of its possible states. However, it scales better to harder verification problems than symbolic model checking [40], since its configuration is explicit.

Symbolic Model Checking. To avoid the state-explosion issue in the model checking method, the symbolic category mitigates this problem through representing the system state space symbolically instead of explicitly. Symbolic model checking considers a state space encoding which is implicit, based on Boolean formulas, and examines sets of states in each step. The compressed representation of the state space is done using the binary decision diagrams (BDD) [41] as one approach to symbolic model checking. BDDs are representations of Boolean functions that are compact. The reason why the state space explosion problem is mitigated is that space requirements for Boolean functions are exponentially smaller than for explicit representation.

2.3.3 . TLA+ Language

The abstract models that we aim to generate would be integrated into a procedure of the detection of timing anomalies. Thus, the need to model the cycle-accurate behavior of the generated micro-architecture models. Sev-

eral tools exist for modeling the hardware architectures and verifying timing properties. We generate the formal models of the pipeline modeled as TLA+ specifications, and we use TLC, the explicit model checking tool, to explore the models and prove properties about timing anomalies.

TLA+ is a modeling language that offers a module system, the untyped set theory and predicate logic, making it appropriate for the specification of complex computational systems like computer architectures. The choice for TLA+ language is not coincidental but because of several semantic factors. Untyped set theory (and predicate logic) are aspects of the modeling language that allow for the specification of extensive state data. A further advantage of TLA+ is its advanced module system, which is based on interfaces, parameters, local declarations, etc. This system enables the precise development of abstract architecture models from smaller components.

Abstraction in TLA+ is ensured by temporal existential quantification, which hides unnecessary state elements. However, this is not supported by the TLC model checker. Moreover, refinement is natural in TLA+ and supported by ensuring stuttering invariance (i.e., execution steps that do not change the values of state variables of interest), which allows reasoning about the specification paths on different levels of granularity and temporal existential quantification to slice away the unnecessary state elements.

All the aforementioned concepts establish TLA+ as a unified logical language designed to specify both systems and their properties and verify, using the same specification, both a system and its possible refinements. Abstraction and refinement characteristics of TLA+ are more important in our timing properties verification purpose when we specify the system as a single formal specification which is then systematically refined. TLA+ is supported by an explicit model checker called TLC, which forms a powerful formal specification and verification framework.

A TLA+ model is organized in specifications built on two levels. The first level contains state and state transition formulas (i.e., system specification) and the second level constitutes the property evaluated on a sequences of states (i.e., system properties) to be verified. At the core of a TLA+ specification is the transition predicate, called action, which captures, using the primed notation, the state change, say for a variable `x` as `x' = x + 1`. If `x` is a record variable, its field `f` is accessed as `x.f`, and a partial record update is as `[x EXCEPT!.f = v]` (changing `f` to `v` and leaving the other fields of `x` unchanged). When the same field `f` keeps a value `v`, it is noted by `f |-> v`. Also, if a state variable `x` is not modified by an action, it is written in TLA+ as `UNCHANGED «x»`. To define an instance of module `M` inside another module, we use the construct `INSTANCE M WITH`, with the constants and the state variables of `M` being replaced by expressions of the instantiating module after the keyword `WITH`. Then, we can access operators `op` and transitions `Act` of the module `M`, with the operator `"!"`. This operator gives access to these elements with `M.op` and

M.Act.

The Listing 2.5 illustrates an example of a simplified TLA+ specification to model an in-order processor design. We define the state variables representing the current cycle, the program, and the fetch pipeline stage. The specification Spec is defined through two operators: Init operator describes the initial state, while Next operator defines the next-state relation. We initialize the state variables to "0" and "empty" values in the initial state, where empty defines an empty content. Next is composed of several actions, in disjunction, where we update the current cycle (currCycle) and the first pipeline stage (_IF) in order to capture the execution of a program through the pipeline. TLA+ features expressions such as condition statements implemented with IF-THEN-ELSE and LET-IN to write the local definition.

Listing 2.5 – Simplified TLA+ specification.

```
VARIABLES currCycle, _IF, prog,......

Init == /\ currCycle = 0
        /\ _IF = [ PC |-> empty ]
        ........

FillIF == LET nxt == next_instr (prog) IN
          /\ _IF' = [ PC |-> IF condFillIF THEN nxt ELSE _IF ]
          /\ currCycle' = currCycle + 1
        ........

Next == FillIF \/ ......
Spec == Init /\ [][Next]_<< currCycle, _IF, prog,... >>
Prop == currCycle < 50
```

Formal verification is used to check the correctness of the properties. We specify the property to verify in Prop and then, the TLC model checker explores Spec and stops when the property is not verified, for instance, when the cycle reaches 50 (a counter-example is returned) or when the property is verified.

2.4 . Conclusion

We introduce the main elements we need in our workflow in Figure 4.1. We started by presenting a general background on hardware construction languages and highlighting their advantages in enabling reusability and customization. We specified our chosen hardware construction language and its compilation framework Chisel/FIRRTL in Section 2.1. Then, in Section 2.2, we introduced the main HDL design of processors related to the pipeline analysis

phase. Finally, in Section 2.3, we detailed the formal modeling and verification methods and tools we will employ for timing properties proof in our thesis work. From this preliminary presentation, we detail in the next chapter the use of these elements for the purpose of generating pipeline models to verify timing properties.

3 - State of the Art

Verifying the timing properties of safety-critical systems requires not only models of the critical applications but also models of the hardware that executes these applications. The hardware models are in general considered at the micro-architecture level. These micro-architecture models are generally developed from processor manuals. Recent trends in hardware design and ISA distribution led to more processor code being made available and supported by the emergence of new design languages with a high-level abstraction, which makes the automatic generation of micro-architecture models possible.

In this section, we introduce related works on constructing micro-architecture models from code of hardware designs. Thus, several aspects should be considered. First, we present various works addressing the core of our research problem, that of pipeline modeling, in Section 3.1, Second, we provide an overview of related work on hardware description languages (HDLs), in Section 3.2, the type of languages we analyze to construct the pipeline models. In this section, we also highlight the main advantages of employing Hardware Construction Languages (HCLs) over standard design languages when describing the semantics of hardware designs. Third, since our approach is based on code-level analysis, we use several techniques for code exploration to develop our analysis. Section 3.3 details related work on program analysis-based techniques and algorithms to be adapted for our analysis. Finally, we highlight further in Section 3.4 some related work on formal verification of timing properties by model checking. Our pipeline models are formalized and used in the verification of timing properties using model checking. We capture in this section the use of model checking to verify timing properties. We have a focus on their existing pipeline models in order to generate, with our thesis work, suitable models for model checking verification.

3.1 . Building Pipeline Models from Hardware Designs

Building pipeline models from hardware designs was the subject of several research works, however different in their purpose, technique employed, and hardware description language of the design. The works [42–44] aim to generate formal pipeline models for **functional verification** purposes. Other kind of works [3, 4, 9–11, 14, 45–47] focus on constructing pipeline models for **WCET applications.** The pipeline model construction could be done manually from processor manuals [9–11, 14, 45–47] or built from HDL designs [3, 4]. Finally, the works [48–50] address a **synthesis problem**, with [50] that focus on high-level synthesis and [48, 49] as they synthesize automatically a pipelined

implementation, automating the design from a sequential implementation.

We start with works [42–44] on pipeline models for **functional verification** purposes. A first work in automatically generating a formal pipeline is presented in [42]. This paper introduces the HADES tool, where the pipeline-based processors are represented as graphs, with the user being required to identify the architectural resources such as registers and memory ports. Then, the HADES tool identifies the pipeline stages and generates the pipeline model as a graph which is then used to deal with data hazards. Indeed, after having generated the pipeline graph, the HADES tool identifies the pipeline stages using a data flow analysis. Then it checks the correctness of the instruction execution through the identified pipeline stages. Furthermore, the abstract pipeline models generated with the HADES tool include the forwarding mechanism, as HADES concentrates on data hazards. It addresses the absence of problems caused by data hazards on microprocessors with a single pipeline and in-order execution.

Furthermore, generating the formal model from hardware designs is a recurrent task in the hardware compilation field. However, most works focus on RTL-level designs, described using standard hardware description languages (VHDL, Verilog) [42, 43]. Their micro-architecture designs are modeled using these kinds of languages. The work in [43] investigates a formal verification system for hardware designs called Cadence SMV [51]. In fact, the Cadence-SMV uses Verilog to express the system model. It uses the Verilog design as input to generate the SMV design. It is constructed by integrating the micro-controller RTL and embedded software assembly code levels. Then, the model is validated, and the properties to verify are proved by conducting a model checking tool..

Moreover, the work in [44] addresses the formal property checking for the hardware designs. This work aims to provide an automatic verification technique for Verilog designs as it builds hardware models from the input designs developed in Verilog. The model is a transition system at the RTL level. Verilog design is translated first to ANSI-C programs based on a synthesis semantics interpretation of the Verilog code, then to a common representation (IR) for the goal of formal verification with various technologies. The aim is to increase the performance and scalability in the verification since the new representation allows the application of a range of modern analysis techniques for software.

We continue our survey on pipeline modeling with works addressing pipeline models in the context of the **WCET analysis** [3, 4, 9–11, 14, 45–47].

The works in [3, 4] address the goal of analyzing processor design code [3] and determining micro-architecture models for the WCET analysis [4]. Moreover, these works [3, 4] present a framework based on static analysis of hardware designs. The hardware design, which is developed in the VHDL language,

is transformed into an intermediate control-flow representation on which an abstract interpretation is built, for each basic block of the input program.

Figure 3.1 illustrates the VHDL static analysis framework, which translates a VHDL code into a sequential program with an arbitrary execution order of its processes. The processes corresponding to functions, procedures, assignment signals, loops, etc., are transformed into routine constructs specified in the intermediate representation (CRL2) of the aiT timing analyzer. Then, a static analysis is performed through the program analyzer generator (PAG) on the intermediate VHDL representation, for timing analysis.

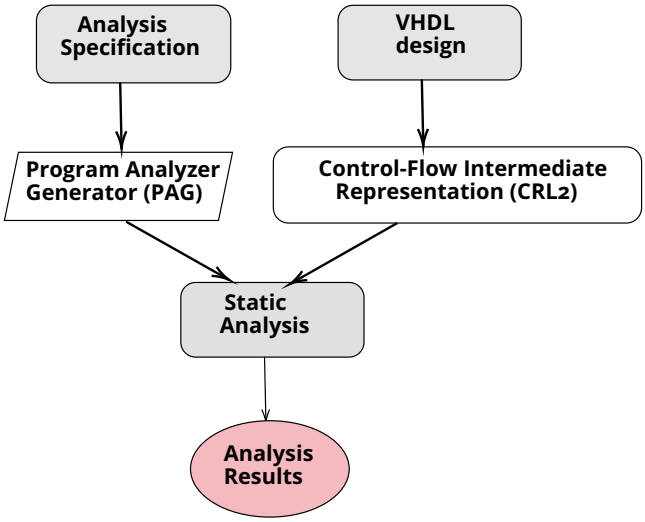

Figure 3.1 – Structure of a VHDL-based static analysis framework, from [3].

The results of [3] are applied in [4] to determine microarchitecture models using a semi-automatic procedure based on program slicing. More precisely, program slicing is used to eliminate parameterized features of the design (i.e., for conditional compilation), and certain signals (e.g., for events and interrupts) are manually initialized. Thus, these signals never change value and can be removed easily through dead-code elimination and control refinements. The approach in [4] performs value analysis to approximate register and memory contents, aiming to simplify the original VHDL design (i.e., changing pseudo-VHDL processes) to a datapath representation with accurate timing. The approach depends on correctly identifying the control path information, which is not fully automated.

This approach is based on the framework from [3] to perform dead-code elimination, control signal refinements, and datapath elimination. These applied transformations reduce the code size of the VHDL design. Then, abstractions are implemented concerning the memory, the types of variables, and the registers where the analysis focuses on the time of storing values and ignores the value stored. Figure 3.2 details the main abstractions applied to

the VHDL design in order to get timing models. The approach proves its efficiency for simple architecture, such as the DLX pipeline, and failed for more complex architecture, like LEON2, an industry processor. Indeed, some VHDL statements are not supported in their modeling as composite data structures and wait statements. Furthermore, the processor state abstractions are manual and dependent on the particular processor and its complexity.

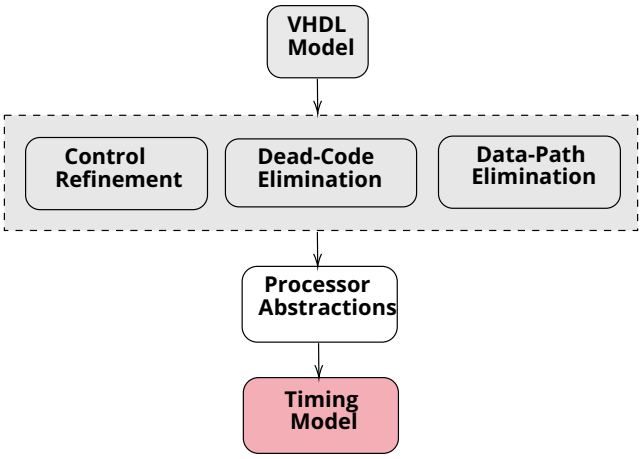

Figure 3.2 – VHDL timing model abstractions, from [4].

Pipeline models which are not extracted from HDL processor designs but relevant to the WCET analysis are addressed in [9–11, 14, 45–47].

State of the art static timing analyzers such as aiT [14, 45], Otawa [9], Heptane [10] or Chronos [11] consider architecture models (of caches or pipelines) and use static analysis to characterize their timing. These architecture models are usually handcrafted using mainly the processor documentation as input and sometimes validated against hardware simulators for conformance. Otawa [9] is a toolbox for the WCET analysis integrating analyses for both program and architecture. Listing 3.1 presents a snapshot of the micro-architecture configuration file in the XML format. First, the pipeline width and length are specified. Then, the pipeline model is presented with its stages. Each stage is characterized with its name specified in `id` field, lines 6, 11, 16, and 34 in Listing 3.1. Number of functional units and their latencies, lines 21-33, binding of instruction categories to the functional units. The pipeline model is not very rich. Its description focuses more on the pipeline stages level. In contrast, the pipeline mechanisms, such as forwarding and stalling, are described later in the OTAWA code. They are resolved at the program level, where the instruction dependencies are preliminarily studied.

Listing 3.1 – OTAWA XML file for hardware abstraction.

```
<processor class="otawa::hard::Processor">
  ...
  <stages>
     <stage id="FI">
        <name> FI</name>
        <width> 2</width>
        <type> FETCH</type>
     </stage>
     ...
     <stage id="EX">
        <name> EX</name>
        <type> EXEC</type>
        <width> 2</width>
        <ordered> true</ordered>
        <fus>
          <fu id="FALU">
          <name> FALU</name>
          <latency> 3</latency>
          <pipelined> true</pipelined>
        </fu>
       ...
     <stage id="CM">
        <name> CM</name>
        <type> COMMIT</type>
        <width> 2</width>
     </stage>
   </stages>
```

Heptane [10] and Chronos [11] analyzers are built on top of the freely available SimpleScalar simulator [52]. SimpleScalar is a cycle-accurate architectural simulator that allows the user to model a variety of processor platforms in software. Figure 3.3 illustrates the structure of Chronos tool to compute WCET bounds.

Heptane/Chronos compute WCET using static analysis at the binary code level. The input program is presented in basic block windows. These tools perform micro-architecture modeling, which yields time bounds for each basic block's execution. Each instruction in the basic block has a time interval at which it can start/finish the execution. Then, tools compute the dependence between instructions and the stalling delay to be added to the basic block time counts to refine the estimates. Thus, the forwarding and stalling mechanisms are not explicitly coded, and the WCET bounds are determined, including the program analysis and the pipeline modeling.

As such, the work in [46] considers basic blocks to be mapped into the pipeline stages in order to define an execution graph, which is then used to

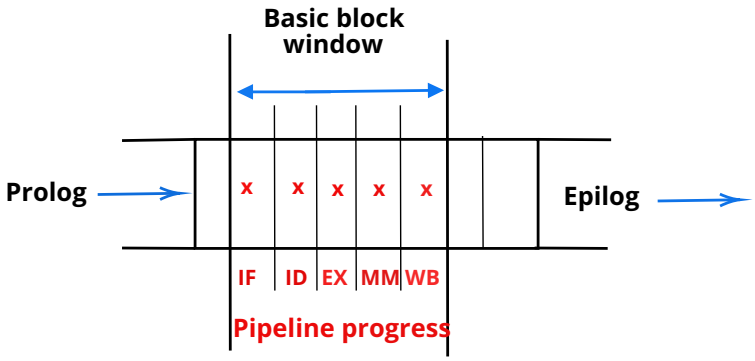

Figure 3.3 – Chronos pipeline modeling overview.

compute the WCET bound. WCET computation in [46] is enhanced by considering variable latencies in instruction sequences. The approach exploits the fact that several latency combinations produce the same WCET and employs Binary Decision Diagrams to improve the WCET analysis' scalability. The work in [47] also models an out-of-order superscalar processor for the WCET analysis. The approach is based on an execution graph that integrates both the micro-architecture models and the program-level analysis defined on basic blocks. The execution graph captures data dependencies, resource contentions and degree of superscalarity. Each node in the execution graph is associated with the latency interval of the corresponding pipeline stage. The estimation technique proceeds by a fixed-point analysis of the time intervals at which the instructions enter/leave a pipeline stage in order to avoid exhaustive enumeration of instruction schedules. Then, the basic block estimates are combined using Integer Linear Programming (ILP) to produce the program's WCET estimate.

Other related works on pipeline modeling address a **synthesis problem**, in [48–50]. The paper [50] proposes a SpecHLS, a source-to-source compiler framework that performs a high-level synthesis for speculative hardware structure. They focus on speculative loop pipelining for arbitrary control flow and memory speculation patterns. The framework takes as input C code and produces transformed C code that supports speculative loop pipelining. The C code is transformed into an intermediate representation (IR). Then, the analyses are explored on this representation. The approach is also applied for accelerator-based in-order pipelined RISC-V CPU and shows their performance improvement for speculative execution. Furthermore, [48, 49] create automatically the processor pipeline model from a sequential abstract model based on registers. More specifically, their goal is to synthesize pipeline designs from sequential representations, starting from simple designs [48] to more complex variants [49]. The approach in [48] has two phases. It first identifies the number of pipeline stages to create a sequential pipeline imple-

mentation based on the forward dependencies. Then it adds, in the second phase, the forwarding and interlock logic required for a pipelined execution.

Furthermore, the work [49] models the datapath as a transactional specification to synthesize automatically an in-order pipelined implementation. The transactional specification considers the datapath as executing one transaction at a time. It captures an abstract datapath, where its execution semantics is interpreted as a sequence of "transactions". Transactions consist of state elements that include the combinatorial and sequential logics, and next-state compute operations and updates through the logic blocks. The analysis is based on an assignment algorithm where each module (or interface in the case of a state element) is assigned to the corresponding pipeline stage.

Summary. Several works have been presented in this section to describe the techniques of building pipeline models from HDL designs. Hardware designs, in general, and pipeline designs, in particular, can be expressed with various kinds of hardware languages. We present next works on HDL languages with a highlight of their semantics and their advantages/shortcomings for the purpose of constructing abstract hardware models which could be used to verify timing properties.

3.2 . Semantics of High/Low-Level Hardware Languages

Hardware description languages (HDLs) allow the specification of hardware design's functional and temporal behavior. We distinguish between mainstream, traditional HDLs with low abstraction levels as VHDL and Verilog, and more expressive languages, capable of high-level of abstraction, such as Chisel [19], BlueSpec [24], SAFL [53], Lava [54], etc. These languages are known as Hardware Construction Languages (HCLs). HCLs can be seen as domain-specific languages that leverage the power of high-level languages like Python, Scala, etc., to build efficient hardware generators that produce, in the end, synthesizable VHDL or Verilog code.

Traditional HDL languages are declarative and provide very little reflexivity over the circuit being described and could require an appropriate abstraction level in order to generate a formal model. More precisely, these work [55–57] address this aspect by presenting **the semantic challenges of the Verilog language**. Recently, the hardware designers started to move towards **hardware construction languages** which are convenient to facilitate formal verification and synthesis [58] such as SAFL [53], Kami [59], and Bluespec [24], and domain-specific languages (DSLs) [60] as Chisel [19], Lava [54] and CλaSH [61]. Finally, there are high-level HDL languages designed for specific types of hardware like **accelerators**, e.g. Spacial [62] and Calyx [63]. Then, we give an

overview of the **intermediate representations** compiled from the hardware construction languages through the hardware compilation frameworks [23, 64]. We then describe work on **comparative studies** [65–69] regarding the general semantics of hardware construction languages and their main advantages over mainstream HDLs.

We start with **semantics challenges of Verilog**. The following works [55–57] consider Verilog as the driving example of a hardware description language widely employed in industrial designs. The work in [55] describes the Verilog language and addresses the variety of interesting semantic challenges that Verilog presents. The first challenge is that the studied Verilog semantics in [55] does not include value sizing and is not executable, making specifying what output a given program should produce more difficult. Also, a Verilog semantic challenge concerns the development of the language's smaller and traceable semantics and the synthesis formalization to prove the equivalence between the hardware structures generated and the behavioural source. Another semantic challenge concerns the simulation cycle semantics and how to integrate it into the formalization of Verilog.

Then, the works in [56, 57] take into consideration these challenges and propose formal semantics of Verilog towards providing a concise reference of the official language standard and aiding the development of Verilog-based verification tools. One of these semantics, proposed in [56] is based on the functional programming style and covers the Verilog functionalities (e.g., its non-determinism) in an executable framework, creating the possibility to test Verilog code. However, the semantics [56] does not deal with events and simulation traces. These features are considered in [57], where the focus is on the formal simulation semantics of Verilog. More precisely, this work [57] presents a formal semantics of traces (the sequence of states in a simulation) and events in HDLs and, in particular, in the Verilog language. In order to achieve this goal, their work proceeds in three main steps. First, it presents the syntax of Verilog linked to time constraints, for instance, events, clocks, etc. Second, it presents Verilog in the simple form of a pseudocode, in order to reduce this language constructions to a canonical form and to simplify the formalization and the verification. Finally, it presents the formal semantics of traces through a definition of traces for a combinational logic and for a sequential logic and then a combination of the two, taken at the circuit level.

In summary, Verilog [55] poses a variety of interesting semantic challenges that are considered in [56, 57], however with this language (or VHDL) the circuit details are fixed early in the design and the flexibility is lost when further modifications are required. Thus, hardware designers seek high-level descriptions to synthesize hardware.

In the following, we overview the semantics of more expressive hardware

languages, called **hardware construction languages**. As such, we enumerate SAFL [53], Kami [59] and Bluespec [24] as languages designed to facilitate the formal verification and Chisel [19], Lava [54] and CλaSH [61] as DSLs.

The work in [53] describes SAFL, a high-level hardware description language used in circuit design. In particular, SAFL is expressive to describe processors and their functional properties and uses transformations to change the design. For example, SAFL provides transformations to adjust the source program's area-time consumption and optimize the code without altering its complex structure. Thus, the main advantage of SAFL is that it is resource-aware.

The work in [59] introduces Kami, a platform for high-level parametric hardware specifications. Kami is based on Coq [70] to specify and verify hardware designs, then uses Coq's code extraction to generate code for FPGAs. A hardware design in the Kami language is presented as a set of modules as it addresses complete circuits through small components that are combined. For example, Kami is used to specify, implement and verify realistic multiprocessor systems with design functionality covering the pipeline and the simultaneous memory/cache access. The main advantage of the Kami framework is that it can generate HDL code for FPGAs from the formalization.

The work in [24] presents Bluespec, a high-level hardware description language that is essentially a variant of SystemVerilog. It is a language that simplifies and promotes formal proof while preserving the compiled circuit's performance. A hardware design in Bluespec is specified in atomic rules and guaranteed that rules appear to execute atomically one at a time, controlled by a scheduler. This scheduler expresses the synthesizable behavior as it is automatically generated to avoid race conditions between the atomic rules. The main advantage of Bluespec is its semantics based on rules which lead to naturally generating HDL code for FPGAs.

All these languages presented in [24, 53, 59] are considered recent hardware construction languages designed for synthesis and functional programming. SAFL [53] is best described as a synthesis rather than a construction language, while Bluespec [24] and Kami [59] are generally employed for formal proof. Furthermore, Bluespec has adapted imperative features and a syntax that resembles Verilog. In addition, Bluespec re-implements the functional language itself with its own syntax and parser. However, embedded languages in a DSL avoid having to re-implement a parser which greatly increases productivity. Furthermore, they are severely limited in practical applicability [58].

There are hardware construction languages that are embedded in a powerful host language, the so-called domain-specific languages (DSLs) for hardware. The work in [60] surveys such DSLs providing a vision that combines a meta-programmed host language with a core embedded hardware description language used as the basis for many domain-specific languages. Further-

more, this combination provides an abstraction that enables code reuse and improves programs' correctness. Several DSLs on hardware designs [19,54,61] exist in the literature.

Lava [54] and CλaSH [61] are two hardware construction languages embedded in Haskell a functional programming language that focuses on the hardware's structural representation. These languages support high-level programming features such as polymorphic types and higher-order functions, besides support for simulation and circuit synthesis. Lava descriptions can be interfaced with formal verification tools. CλaSH differs from Lava as the description contains a rich control structure. The control structure contains the case-expressions and pattern matching as specified with Haskell's choice elements. These elements are synthesized into the choice elements in the eventual circuit. Thus, they can be best specified in CλaSH than possible within Lava, and hence are less error-prone.

Chisel [19] is a DSL language in Scala. Chisel benefits from this integration as Scala is expressive, combining object-oriented and functional abstraction styles, and allows the encoding of a flexible hardware library. Furthermore, this embedding brings several other advantages like avoiding re-implementing a parser and having free access to all Scala libraries, leading to great productivity. Various aspects of the syntax and the semantics of Chisel are discussed at the end of this section, where we present several comparative studies as well as in the next chapters.

We continue our survey with another kind of hardware description languages which are not generic but specialized for **hardware accelerators** in [62, 63]. Spacial [62] is a language and compiler framework for higher-level descriptions of application accelerators. It is developed to support performance-oriented hardware designs implemented on reconfigurable spatial architectures, including FPGAs. Indeed, it simplifies the accelerator design process, allowing the development, testing and optimizing hardware accelerators. Calyx [63] is an intermediate language to compile high-level programs into hardware design. Calyx combines software to represent the control flow of a design with hardware as a structural language for hardware modules. The Calyx compiler optimizes the programs and emits synthesizable RTL.

The compilation chains for high-level HDLs also propose **intermediate languages** to handle the semantics gap with respect to low-level Verilog and VHDL. Moreover, developing automated techniques for analyzing such HDL designs becomes tantamount to working with both the high-level HDL languages as well as with language transformation (i.e., compilation) issues. For example, the work in [64] proposes an intermediate-language representation named LLHD, which is inspired by LLVM IR, thus, in the SSA form. LLHD plays similar roles in a hardware compilation chain as FIRRTL [23]; for example, it

helps to integrate compilation passes.

We continue with papers that focus on the **comparative studies** between hardware construction languages and traditional hardware description languages. Various research studies [65–69] have been carried out on Chisel to demonstrate its efficiency over other existing HDL languages. While low-level hardware description languages VHDL and Verilog are still used in industry, their age and original intention put them behind current high-level languages in terms of productivity and flexibility [65, 66]. The work in [66] reports the importance of static type checks of hardware generators in high-level languages compared to existing HDLs. HCL languages allow static type checking of a generator to discover potential design issues early in the design cycle. Furthermore, they need to support sophisticated hardware generators provided in high-level languages. The paper [65] details these properties by enhancing the main features of Chisel languages and compares its advantages with VHDL language in terms of productivity and code size. We present next these properties highlighted in [65].

First, Chisel features generate predefined blocks [65] in order to simplify hardware design by providing parameterization for common hardware. Second, its datatype structure makes interfacing modules easier, allowing code reuse. These datatypes will be explained in detail in Section 4.2. Finally, Chisel benefits from the high-level programming features of Scala language, such as abstract functions, polymorphism, etc. These concepts are not presented in VHDL language, which could lead to duplicated code for similar functionality. However, the work in [65] highlighted that the Chisel language is still in early development, which makes it not yet suitable for use in industry. At that moment, it was still under development and suggested a new stable syntax and documentation.

We continue with works in [67, 68] related to benchmark studies between low-level and high-level HDLs, especially between Verilog and Chisel language. The comparison metrics include the source code density, the synthesizer circuit area, the RTL simulation runtime [67], design flow runtime and speed of coding through a range of FPGA design components [68]. High-level programming features in Scala languages allow Chisel designers to reduce the source code lines. They can reduce codes by using compound types instead of typing every signal in Verilog and employ functions to describe hardware circuits as, for example, pipelines. Chisel improves over Verilog in code maintainability and scalability. It can also efficiently detect architecture issues and generate direct warning and error messages in complex designs [68].

Summary. This section has detailed works related to hardware languages, including the traditional HDLs and HCLs languages. We have focused on their semantics and their significant differences for the goal of building pipeline

models. Building pipeline models relies primarily on code analysis. We present the following works on code analysis techniques, focusing on those that can be exploited in our approach.

3.3 . Code Analysis Approaches

Analyzing open-source hardware designs means, on the one hand, integrating into the analysis the semantics of the HDL used in the design and, on the other hand, understanding what the design is about, for example, that is a processor design. Design complexity and library reuse make working on the whole design difficult, therefore, the need for structured code-level design and analysis techniques is necessary. Several code analysis techniques have been used in literature to address various problems in design, simulation, testing and formal verification. We group these works into **general-purpose** analysis techniques and code-level **HDL-specific** techniques. For instance, in the general-purpose category, we mention the works on the abstract syntax tree (AST) [71, 72], while other works use dependence graphs [5, 73, 74]. Furthermore, program slicing [75, 76] is among these techniques that intend to reduce the program and work on the relevant code portions. Finally, static analysis [77,78] is also used to analyze code for several purposes. For the HDL-specific methods, we intend to present the application of these techniques on hardware designs. As such, the works in [75,79] address program slicing applications, while the works in [80–83] propose applications of the static analysis.

We start first with an overview of **general-purpose** analysis techniques. Analyzing the source program code based on analyzing the corresponding AST has been developed in different research works [71, 72]. These works present an application example of analyzing C programs, more precisely, to monitor code evolution by comparing the source code of different versions of the code under analysis. This approach is based on analyzing the partial AST matching in order to identify the significant changes (additions, deletions of global variables, types, and functions) between code versions and report various statistics on dynamic software updates. The approach traverses, in parallel the two AST produced from the programs parsing and collects the construction elements as type and name to build a mapping. The approach in [72] also begins by finding function names that are common between program versions and do not change very often as the beginning assumptions. Then it goes through the AST (function body) to identify changes in function semantics.

We continue with works [5, 73, 74] that use dependence graphs to analyze code. The work in [5] presents an intermediate representation for programs based on the Program Dependence Graphs (PDGs) [73] which is used to ana-

lyze the data/control flows in programs. PDGs provide a unifying framework that makes data and control dependence explicit for each program operation and thus it permits an efficient and powerful program transformation. The intermediate representation in [5] presents various benefits to the program analysis and a basis for program optimization. Figure 3.4 illustrates a control dependence graph based on the control flow graph and dominators. A control flow graph is a directed graph G augmented with a unique entry node `Start` and a unique exit node `Stop` such that each node has at most two successors. The two successors of nodes have attributes ‘‘`T`’’ (true) and ‘‘`F`’’ (false) associated with the outgoing edges in the usual way. For each node `N`, we assume that there exists a path from `Start` to `N` and a path from `N` to `Stop`. We construct the control-dependent graph through the following definition. Let `X` and `Y` be nodes in `G`. `Y` is control dependent on `X` if there exist a directed edge `P` from `X` to `Y` with any `Z` post-dominated by `Y` and `X` is not post-dominated by `Y`. We can notice that a node `V` is post-dominated by a node `W` in `G`, if every directed path from `V` to `STOP` (not including `V`) contains `W`. Region nodes R1 through R6 and Entry have been inserted to more accurately represent control flow dependencies, especially in the presence of loops and conditional structures. The control dependence graph is built using a control flow graph and dominators in the approach [84, 85].

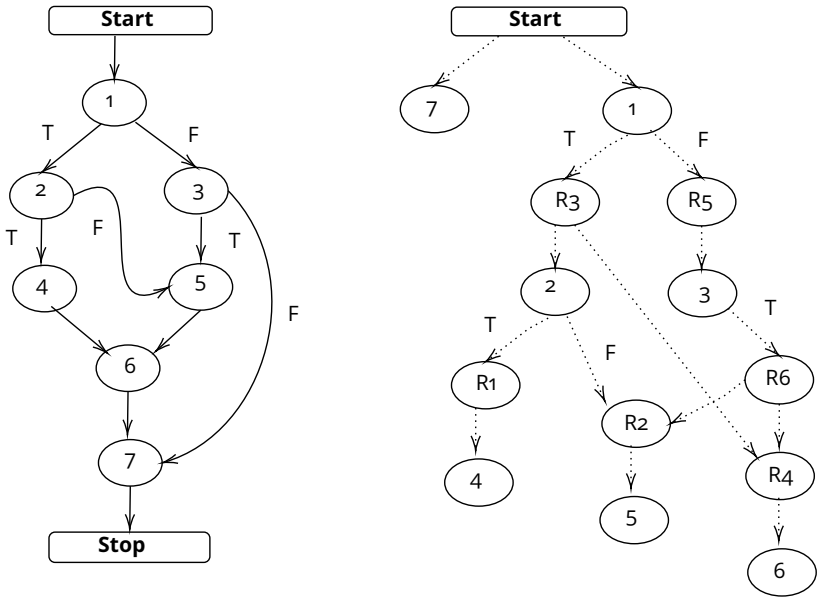

Figure 3.4 – A control flow graph and its control dependence subgraph, from [5].

Another work [74] related to the dependency graph implementation for the goal of detecting polymorphism computer malware. Polymorphism malware is typically based on malware viruses with different signatures. They

are challenging to detect since their signatures still need to be analyzed when they first appear. Thus, the approach is based on comparing the dependency graph of the target file with all the database virus dependency graphs to determine whether this target file is a polymorphic variant of existing detected malware.

Static analysis [77,78,86] is one of the most popular approaches employed when checking properties at code level. [86] describes the use of the static analysis fixpoint approach to solve the path problems in graphs. The paper [77] presents the application of semantic-static analysis to check errors in software design based on abstract interpretation. Semantic-static analysis can be used to prove properties at various levels of the software designs and always covers all the program executions as static analysis works on the control flow graph. The properties of interest in [77] are the run-time errors in the code. A detailed presentation of static analysis and its applications can be found in [78]. Program slicing [75, 76] is a static analysis technique that automatically extracts portions of programs according to an objective (also known as slicing criterion). Slicing provides a program reduction that simplifies the original code and facilitates its formal verification. The work in [76] presents an overview of algorithms for program slicing covering both static and dynamic approaches as well as language features like procedures, data types, pointers, etc., as data and control flow graphs.

We continue with works in [75, 79–83] related to **HDL-specific** code-level analysis. More precisely, the works in [75, 79] present implementations of program slicing for VHDL designs. The work in [75] presents a slicing tool that automatically decomposes programs based on data and control flow analysis, using PDGs, in order to construct sliced programs. This work aims to deal with the concurrent constructs in the VHDL language. The approach maps VHDL constructs into constructs of other traditional procedural languages such as C and Ada languages. The approach relies on the VHDL semantics in a way that ensures that all the VHDL program traces, which are parallel, are valid for the corresponding sequential program. The work in [79] extends the static program slicing to an improved variation called conditioned slicing. Conditioned slicing is an abstraction technique used for hardware verification that adds conditions to the slice in order to produce a smaller and more significant abstraction than static slicing. It specifies some initial states of interest in the slicing criterion for hardware verification. The states rely on the property to verify to apply conditioned slicing to Verilog code. The resultant conditional slices are more precise and relevant for the verification and also reduce the size of the verification state space.

The works in [80–83] focus on using static analysis on HDL designs. The work in [80] presents an application of static analysis on scheduling hardware designs to deal with resource-sharing between hardware components. The

aim is to ensure scheduling to access a shared resource throughout the hardware compilation from high-level to structural low-level languages. To achieve this, a dynamic scheduling logic is generated automatically in a manner that avoids deadlock, then the static analysis is used to remove the redundant scheduling logic which can significantly improve the efficiency of generated circuits. Another use of static analysis on HDLs is proposed in [81]. This work computes code coverage of an assertion in a given RTL source code. We note that assertions are properties that a hardware design should satisfy, and the coverage achieved during the verification is an important criterion to consider when assessing the quality of the verification results. The code coverage of the assertions is based on the coverage of the statements and their dependencies within the assertions. In order to compute the dependencies of the statements, static analysis is used to analyze the control data flow graph of the corresponding code. Finally, static analysis is also employed in the security evaluation of HDL code, as in [82, 83]. Sapper [82] and Caisson [83] are two hardware description languages for designing security-critical hardware components. These languages express security by using static analysis at compile time. They are based on the control of the information flow to put restrictions on the information flow through the system

Summary Various techniques and algorithms for program analysis have been presented in this section. We have also described their application over HDL designs for different purposes. In the following section, we present works related to the formal verification of timing properties using the model checking tool.

3.4 . Model Checking for Timing Properties

Formal verification is based on verifying properties of formal specifications, from where the necessity to generate formal models by adopting an appropriate level of abstraction. In Chapter 2, we have detailed model checking as our formal proof technique employed in our thesis as the way to verify timing properties, such as those related to the **computation of WCET bounds** [87–94]. All these works employ model checking tools to explore the execution of the input program under pipeline models. Furthermore, several works have also used model checking to prove **timing predictability-related properties** [95] such as the detection of timing anomalies [6, 96–100], we present hereafter their applications. Various formal modeling and verification languages are used for describing models and verifying timing predictability-related properties, in particular, UPPAAL in [87, 88, 91], UCLID5 in [96, 97] and TLA+ in [6, 95, 98].

We start with works [87–94] related to model checking for **computing WCET bounds**. The works in [87,88,91] compute WCET bounds by using model checking for both the program and the architecture. Model checking for complete WCET estimation is first proposed in [89]. This approach is standard, both the program and the architecture (pipeline and caches) are modeled as transition systems which are then encoded in a model checker. The main point of this work is that model checking is more precise than the combination of abstract interpretation and integer linear programming (ILP), developed in the WCET analyzers. They have experimented with the behavior of instruction cache access. Thus, the model checking has proved that it provides a precise cache behavior prediction that avoids the abstraction methods' over-approximations. Since ILP is known to suffer from numerical instabilities that can produce non-valid solutions, unlike model checking, which produces correct runtime predictions. The work in [90] supports an opposite point of view, that it is preferable to estimate the WCET using this combination of abstract interpretation and ILP than model checking. The main arguments are, on one hand, the fact that model checking suffers from a state explosion problem, and on another hand, model checking cannot compute properties (e.g., invariants) as abstract interpretation. However, they are experimenting with quite complex processors. Also, according to [90], model checking may produce more precise results since it exhaustively checks all possibilities, while abstract interpretation may lose information due to abstraction.

The work in [87] presents an approach to compute the WCET bounds based on a combination of model checking and static analysis. This work combines hardware and software components, where the hardware model consists of caching and pipelining and the software model is the control-flow graph of the application binary. Both models are represented as timed automata and model checked using the UPPAAL toolbox. Timed automata is a state machine with locations and edges. A state includes the system's variables, and the edges describe the possible transitions from the states. The main characteristics of UPPAAL is the existence of real-valued clocks in addition to finite-state transition systems. In order to evaluate the approach and to show its modularity, several implementations have been performed to support various architectures ARM7, ARM9, and ATMEL AVR. The experiments demonstrate that much tighter WCET estimates are found when taking instruction caching into account: up to 96% tighter estimates, and 72.2% on average. Also considering the data cache increases the average to 82.3%. Furthermore, the average analysis time is just under four minutes when taking both caches. Another work, in [91] also uses the UPPAAL toolbox to compute WCET bounds. The target architecture is a shared memory multicore system that executes parallel applications. The paper models and analyses the impact on WCET from a memory hierarchy comprising shared L2 caches and

core-specific L1 instruction and data caches.

This work [88] considers an architecture with a 5-stage pipeline, an instruction cache, a dynamic branch prediction mechanism, a branch target buffer (BTB) and a prefetch instruction buffer. The goal is to measure the impact of dynamic branch prediction and BTB on the estimation of the WCET. The approach models both the program and the architecture using UPPAAL. The program modeling relies on slicing to narrow the set of instructions and memory locations which are useful in calculating the WCET and to limit, in this way, the state space explosion problem. The work considers a micro-architecture inspired by the e200z4 Power 32-bit architecture, a 2-issue static scheduling superscalar core with five pipeline stages. The evaluation is conducted by a set of experiments that used the Malardalen WCET benchmarks [101] to generate the programs. The paper proves the positive impact of dynamic branch prediction modeling on the WCET estimations. Indeed, A more effective prediction strategy will result in fewer control risks and, thus, fewer configurations for the pipeline's lower stages, reducing the state space. Therefore, it shows the capacity of model checking to analyze the complex interactions between various micro-architecture components and to perform a complete analysis of the whole system.

The WCET analysis is dependent both on the properties of the program which is executed as well as the underlying hardware, thus, an accurate program and architecture analysis leads to precise WCET estimations. The architecture analysis aims to characterize the cache behavior [92], and the program analysis requires trustworthy information about the binary code to detect unreachable code paths [94] and loop bounds calculation [93]. The work in [92] proposes an approach to remove the uncertainty in the memory access, which can lead to the over-approximation of the WCET. The approach is based on model checking to reason about unknown accesses classified with an abstract interpretation analysis.

In addition, the paper [94] proposes Sequoll as a framework that employs the model checking on binary code to automatically compute both simple and more complex loop bounds and verify infeasible path information. Sequoll framework derives a control flow graph from the binary. Then, this graph can be exploited to identify reducible loops defined with a single entry point from outside the loop body. Sequoll framework analyses only reducible loops via a depth-first search, locating the innermost loop for each instruction and recreating the program's loop nests. The CFG is converted into a Single State Assignment (SSA) transformation to simplify the analysis. Therefore, a program slicing is also applied to CFG with slice criteria based on the relevant variables or control flow nodes The reduced program is then exploited with model checking, which counts the number of times the loop entry node executes. Furthermore, the work in [93] highlights also the loop bounds issues in WCET analysis. The approach includes model checking to find loop bounds for

applications written in ANSI-C. The analysis framework generates some additional code to create a loop counter and to specify an assertion to be checked then with model checking. The approach was tested on several benchmarks.

We continue with works on using model checking for verifying **timing predictability-related properties** [95], in particular timing anomalies [6, 96–100]. Timing anomalies pose various issues in timing analysis thus this phenomenon must be defined and included in the timing analysis workflow. The work in [95] describes an application of model checking to co-validate hardware-software designs for verifying timing properties through TLA+ modeling and verification tool. The work derives the timing models from hardware and software features and verifies their consistency.

Model checking is also used to evaluate the presence/absence of timing anomalies in predictable pipelines [96] and study their occurrences in an industrial processor [97]. The automated detection of timing anomalies is implemented using bounded model checking in [96]. The focus of this work is on amplification timing anomalies, defined as local timing variations leading to even larger global timing variations. These timing anomalies are in general caused by memory accesses.

The formal and executable proposed pipeline models are also developed by hand, in [96], these models are based on a particular abstraction called the canonical pipeline model, where a pipeline stage is modeled as a single variable, and the instructions are abstracted to instructions classes. This work experiments on several predictable pipelines modeled using the UCLID5 formal framework. The amplification timing anomalies have also been verified in [97], which considers a processor used in the automotive field - the TriCore architecture. The paper extends the canonical pipeline model proposed in [96] to cover the dual-pipelined TriCore and considers data dependencies and structural hazards. The encoding and the verification are also performed with the UCLID5 formal verification framework.

Other works related to model checking for timing anomalies are presented in [6, 98]. These works focus on the code-specific problem to identify timing anomalies based on the shared resources, where multiple instructions compete for functional units. In particular, the work in [6] considers the case of superscalar in-order pipelines, and the work in [98] is extended to treat out-of-order pipelines. Both works use TLA+ specification language to formally model the pipelines and verify the presence/absence of counter-intuitive timing anomalies using the TLC model checker tool.

Several formal definitions of timing anomalies exist in the literature [102–104], which are based on the LTL logic defined in Chapter 2. The work in [99] proposes a framework to test the existing definitions [102, 103]. The evaluation is performed on a formal model of an out-of-order pipeline which encodes most of the definitions of timing anomalies into an executable proce-

dure based on model checking. The work in [100] revises the formalization of counter-intuitive timing anomalies by adding the causality concept, which establishes relations between local and global timing effects. The proposed framework applies to a concrete architecture independent of the WCET analysis method. Furthermore, the main advantage of this formalization is that it includes multiple variations and resource utilization of instructions to reason about timing anomalies executions. This evaluation framework in [99] and the formalization in [100] are based on the TLA+ formal modeling and verification framework.

Summary. This section has presented works related to the formal verification of timing properties using model checking as the formal technique adopted in our thesis. We have described the works related to the WCET computation bounds, which employ model checking. Furthermore, we have highlighted those that focus on timing anomalies-related properties. All these works are based on building formal models of the architecture. Our work revolves around this purpose.

3.5 . Synthesis & Conclusion

This chapter reported related works relevant for our thesis work, which aims to construct micro-architecture models from hardware designs. We present next the limitations in the state of the art and the main differences that motivate the work described in the following chapters.

Many studies have been done for building pipeline models from hardware designs. Various works described come close to our approach proposed in the following chapters. We aim to generate in this thesis a dedicated pipeline model to verify timing properties. However, the works in [42–44] focus on the functionality of the hardware designs given at RTL level. The construction of the pipeline models in [42] requires preliminary information as the pipeline depth, i.e., the number of stages as in [48] and the architectural resources. Our objective remains mainly improved since we aim to automatically generate the datapath pipeline models by working directly on the original processor designs. Furthermore, our analysis seeks to provide a bound to the pipeline depth value.

Other works also address code-level analysis (of hardware designs) to extract pipeline models for timing analysis [3, 4]. Their approach is based on abstract interpretation to create a priori correct solutions. We propose an approach that engineers a solution that could be validated a posteriori. Timing analysis on hardware designs focuses on the abstractions highlighting the timing hardware constructions, thus, our thesis work will be based on the register analysis in the pipeline models. Our approach will focus on the registers considered in the datapath part of the hardware designs. The work in [3]

proposes a semi-automatic analysis that separates the control and datapath parts in its approach based on the simplification of the original design. In comparison, the work in [4] combines program slicing with hard-coded initialization. Regarding the control path, our approach considers all the control and datapath registers as long as they are in a given set of pipeline design modules.

There are many other works [9–11, 14, 45–47] that aim to provide the required pipeline models in order to compute safe and precise WCET bounds. However, these works handcraft these models from processor manuals. They simplify the pipeline models into a single variable for a stage, which can not reflect the pipeline design (e.g., forwarding). We aim to generate datapath pipeline models with a proper abstraction level to obtain correct and precise timing analysis. We consider the pipeline mechanisms that can impact the timing behavior of the design and forwarding is among these mechanisms. The models in [9–11, 14, 45–47] are designed to prioritize the instruction progression and not its actual computation. Furthermore, certain pipeline optimizations, such as the forwarding mechanism, are not explicitly encoded in the pipeline models but are handled at the code level through arbitrary timing values. However, the work [48], which creates a fully functional pipeline model, also includes the forwarding mechanism in their pipeline synthesis when assigning states to stages. Finally, in [48, 49], the correctness of the automated pipeline design is ensured during its construction while we will rely on techniques specific to software engineering to determine a solution.

The abstractions described in [3,4] are also based on analyzing the pipeline registers. However, these approaches analyze primarily the low-level languages Verilog and VHDL code. Whereas our thesis work will be positioned on high-level languages, we consider open-source processor designs developed with more expressive HDL languages such as Chisel/FIRRTL. Our proposed approach will work on the AST of FIRRTL, targeting the Chisel/FIRRTL compilation framework. We have presented in this chapter, through several comparatives studies [65–69] the main advantages of high-level hardware description languages called hardware construction languages, in particular Chisel language, over the low-level languages.

As introduced in this section, we have presented several techniques and algorithms for program analysis. The abstract syntax tree (AST) traversal is among these techniques described in [71, 72]. This AST traversal proposed in these works is similar to what we need, as we focus on registers and will also consider in the analysis the semantics of the AST nodes of the pipeline module's body. The approach in [72] poses the initial assumptions as common function names between program versions to traverse and then go through the AST to determine changes. Since we are interested in a register analysis, we can fix the initial assumptions as the pipeline registers, and then we go through the AST to construct the register analysis. Other works use depen-

dence graphs [5, 73, 74] to illustrate their code analysis. Our approach will construct an intermediate representation of the datapath pipeline designs. This intermediate representation performs a registers dependency graph to build the pipeline model. Furthermore, static analysis and program slicing are also presented in various works [75, 77–79]. Our proposed approach for constructing pipeline models from high-level HDL designs needs also to use static analysis and program slicing techniques. Our framework provides the possibility to employ similar techniques as the program slicing. We favor the timing aspect over other functional aspects of the design. Furthermore, we also have the slicing criteria for the Chisel design as we focus on the datapath pipeline modules. We will traverse the AST only for a given number of modules: the datapath pipeline modules. In addition, we will iterate over the AST in order to determine dependency relations between registers as static analysis (fixed-point computation), which determines relations between variables. Moreover, we can verify the registers dependencies coverage as in [81], which illustrates the implementation of static analysis to cover assertions.

We have detailed in this chapter the works related to model checking used to compute WCET bounds [87–94], also to prove timing predictability- related properties [95] such as the detection of timing anomalies [6, 96–100]. The work in [88] includes the branch prediction modeling in its pipeline models construction. Furthermore, [93, 94] computes the WCET bounds on an input program that analyzes loop bounds. In our thesis work, we aim to execute a sequence of instruction traces where the branches and bounds are already resolved for the input programs. We intend to employ model checking for the architecture as in [89] without cache behavior characterization [92]. Instead, we will integrate the cache latencies into the input program representation to compute the precise pipeline cycles. Furthermore, we plan to use TLA+ formal modeling and verification tools based on predicate logic and suitable for the computer architecture instead of UPPAAL as in [87, 91].

We intend to adopt the definition of timing anomalies from [104] and their integration into the existing detection procedure for timing anomalies from [6]. The definition concerns counter-intuitive timing anomalies unlike [96, 97] where they use model checking to verify the amplification timing anomalies with the UCLID5 formal framework. We plan to construct the pipeline models that include the pipeline mechanisms. Thus, our models differ from that of these previous works [6, 95, 98] in the architecture modeling.

3.6 . Problem Statements

The worst-case timing analysis is performed under (micro)-architecture considerations. Our primary goal is to develop micro-architecture models, mainly pipeline models, and thus derive, in this way, the required timing models. The micro-architecture models are usually hand-crafted, frequently based

on documentation [6, 98] and validated through testing. There are several static timing analyzers such as the industrial-strength tool aiT [14] and the academic ones Otawa [9], Heptane [10] and Chronos [11], for which micro-architecture models are developed in such a way. Since this process is error-prone and time-consuming, we aim to address this issue through the automatic generation of these models, specifically the pipeline models. This generation should be based on open-source hardware designs and should integrate timing behavior as they are used to prove timing properties.

Recent trends in hardware design led to more processor code being made available [1]. This progress is also supported by the emergence of new, high-level design languages (e.g., Chisel [19] or SpinalHDL [20]), sitting at the top of specialized hardware compilation frameworks (e.g., FIRRTL [105]). These hardware compilation chains also have the possibility to define configurable optimizations through compilation passes. In this context, the problem of deriving convenient pipeline models is backed by a comprehensive hardware compilation infrastructure. As a first purpose, we have to define the high-level language semantics and deal with the compiler framework to develop analysis.

In order to prove timing properties, we rely on formal verification as the backbone of our work. Formal verification ensures that correct results are produced under all possible execution conditions. We focus on timing predictability-related properties, more precisely on the detection of timing anomalies, thus, the need to generate formal models and then to integrate them into a procedure for the detection of timing anomalies [6]. This application is used to demonstrate the impact of automatically generated models.

We intend to address these key elements in the following chapters.

1. https://github.com/riscvarchive/riscv-cores-list

4 - Workflow of Timing Models Derivation

In this chapter, we present our general workflow to address the problem of designing abstract architecture models in order to formally prove timing properties. We detail the proposed workflow in Section 4.1, followed by a description of the selected RISC-V HDL designs to illustrate our approach, in Section 4.2. In Section 4.3, we give an overview of the HDL language and its compilation framework, and then we highlight the complexity of the language semantics through several RISC-V processor designs.

4.1 . Proposed Workflow

In order to ensure the respect of timing deadlines in safety-critical real-time systems and to prove their correct behavior, we present in Figure 4.1 our general workflow for this purpose. Figure 4.1 summarizes our proposed workflow for the formal verification of timing properties related to predictability [6, 22], starting from hardware designs developed in a hardware compilation framework. Thus, our workflow includes four major components. We propose a workflow that analyzes the hardware designs under a compilation framework (i.e., *Hardware Compiler Framework component*) to extract the pipeline models for timing analysis (i.e., *Pipeline Analysis* and *Program components*). Then, these models are integrated into a model-checking toolchain for property validation (i.e., *Formal Verification component*). We detail next each of these components and the workflow structure from the right-hand side components to the left-hand side ones.

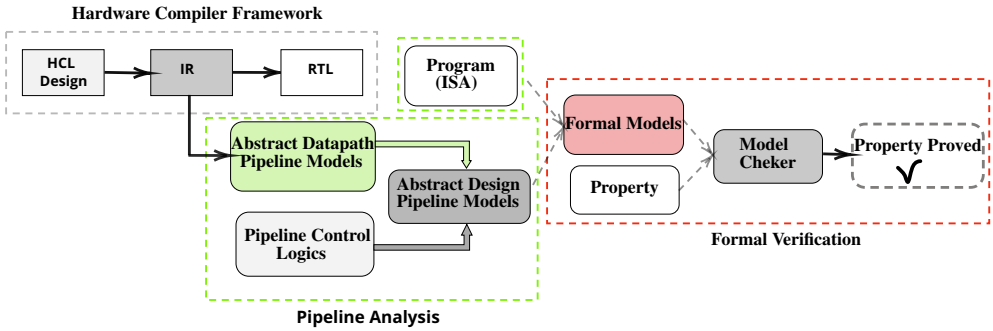

Figure 4.1 – Formal verification of timing properties from hardware designs workflow.

Various tools can be used to prove the timing behavior of the system, with different degrees of confidence, from testing to formal verification. Our workflow is based on a formal verification approach. Formal verification needs to

specify the system and then verify its properties. In our case, those related to timing predictability [6, 22]. [6, 22] employ a model checking on constructed formal models for this purpose.

As such, our required formal models target mainly the non-functional behavior of hardware designs, in particular the processors, as our goal is to verify their timing predictability-related properties. In other words, this means we should be reasoning about executions of an input program on an underlying architecture. In this setting, both aspects, that of the program (software) and the architecture features (hardware), become equally important to characterize the timing behavior of the system. The program provides a sequence of instructions to be executed while the architecture adds timing to this sequence, e.g., counting clock cycles while instructions pass through the pipeline stages. Since we do not need to consider all the functional aspects in our pipeline models, the proposed formal models are based on an abstraction that considers the execution progress of the instructions from the program through the pipeline with cycle-accurate granularity. This combines the *Program component* and *Pipeline Analysis component*. Therefore, the abstraction focuses on the sequential logic of the pipeline, more specific details are addressed in Chapter 5.

A pipeline design includes a datapath and a control path. Automatic construction of these architectural models is possible because hardware designs are made available, i.e., open-source, and are supported by the emergence of high-level Hardware Description Languages (HDLs) and hardware compilation frameworks, for example [18,20,21]. Different ways to abstract the pipeline can be derived while analyzing its code, both data and control paths. Chapter 3 gives more details on the abstract models constructed. In our workflow from Figure 4.1, the pipeline datapath model is automatically constructed. We note that the control path captures the instruction flow through the pipeline stages and currently, the control flow is manually developed and integrated with the datapath in our model. Then, we also need to map the instructions, i.e., the input program, on the abstract pipeline models such that we could define an execution model, in terms of both software and hardware aspects.

Generating formal models from hardware designs is based on the accurate representation of specific details of the processor micro-architecture. In our workflow, in particular the *Hardware Compiler Framework component*, they are derived from HDL processor designs, presented in Chapter 2. We are interested in high-level design languages that pose different challenges when generating abstract processor models specific to the targeted properties. We specify the main challenges in Section 4.3. Note that our contribution related to the automatic generation of abstract processor models is agnostic to the choice of a hardware compilation toolchain but it is presented for Chisel/-FIRRTL workflow. More precisely, we propose an analysis of HCL processor designs based on the high-level language Chisel [19], sitting at the top of spe-

cialized hardware compilation frameworks (e.g., FIRRTL [23]). These hardware compilation chains also have the possibility to define configurable optimizations through compilation passes. So, how can we take advantage of hardware compilation frameworks to ease the generation of abstract pipeline-level models? In this context, the problem of deriving convenient pipeline models is backed by a comprehensive hardware compilation infrastructure.

The HCL designs and their intermediate representations (IR) generated with the compilation chains correspond to the input of our workflow. The Chisel/FIRRTL compilation chain is described in detail in the next section 4.2. Our proposed approach to generate automatically the abstract pipeline models is described in Chapters 5 and 6. Then, we highlight the formal model generation from the abstract pipeline models, this constitutes the output of our workflow. Finally, we present the use of such outputs for the verification of timing predictability-related properties. Both, workflow output generation and its application are detailed in Chapter 7.

4.2 . Chisel/FIRRTL High-Level HDL Designs

We select the Chisel/FIRRTL hardware-compilation toolchain [18, 23] to illustrate our approach. Chisel (Constructing Hardware In a Scala Embedded Language) [18] is an open-source hardware-construction Domain-Specific Language (DSL) embedded in the Scala programming language. Chapter 2 introduced the Chisel language and its corresponding Chisel/FIRRTL hardware compilation framework. Chisel constitutes the frontend part of the toolchain, FIRRTL is the middle-end from which all Scala-related hardware generators have been executed, and Verilog is the backend. We focus next on describing the Chisel language and the FIRRTL representation, highlighting the main features needed to develop our analysis.

4.2.1 . Chisel Hardware Description Language

Chisel offers a set of Scala libraries to write complex, parameterizable circuit generators and new hardware types that can be manipulated to produce synthesizable Verilog. Each constructor in Chisel language combines the hardware design elements and software programming features provided in Scala language.

We introduce in this section some features of Chisel (*Hardware Compiler Framework component*) with applications in RISC-V processor designs. These examples include both Chisel (`Module`, `Reg`, `Wire`, `Bundle`, `Vec`, etc.) and Scala constructions (`val`, `class`, `function`, etc.). Our pipeline analysis is grounded on these constructs, hence on these languages and the compilation toolchain. Chapters 5, and 6 address this pipeline analysis.

Modules. A hardware design in Chisel would typically consist of a set of modules. Chisel modules are defined using Scala classes, i.e., the keyword `class`. A Chisel module contains at least one interface wrapped in an object `IO()`. For example, a Chisel `DatPath` module of the RISC-V Sodor processor [1] is defined as follows, in Listing 4.1.

Listing 4.1 – Module in Chisel language.

```
1  class DatPath(width: Int) extends Module {
2     val io = IO (new DpathIo ())
3     val if_pc_next  = Wire (())
4     val if_reg_pc = RegInit (io.reset_vector)
5     val mem_reg_inst = Reg(UInt())
6     val wb_reg_inst = Reg(UInt())
7     .....
8     when ( C1) {
9       if_reg_pc := if_pc_next
10    }
11    val wb_reg_inst = RegNext(mem_reg_inst)
12    if_pc_next := if_reg_pc + 4
13 }
14 class Core(width: Int) extends Module
15 {
16    val io = IO(new CoreIo())
17    val d = Module(new DatPath())
18    d.io.reset_vector := io.reset_vector
19 }
```

The variable `if_pc_next` is a wire declared inside the module `DatPath` and assigned later with the value `if_reg_pc + 4`, where `if_reg_pc` is a register initialized to `io.reset_vector`, line 12, while `d` is an instance of `DatPath` defined in Scala language with `new` in the module `Core`, in lines 14-19, which is further used to access and update this particular value. The remaining constructions are described later.

Ports. Each module defines an interface IO to hardware components called a port. A port is simply any data object that has directions assigned to its members. The Chisel module `DatPath` of RISC-V Sodor processor defines a new type `DpathIo`, in Listing 4.2, lines 1-4, that can be used as an IO interface with ports defined by `Input/Output` Chisel language constructs. The Chisel language also includes an IO compound named `Flipped` type. It changes the direction of all fields of its argument. The class `DpathIo` defines a `Flipped` type, line 4, in order to include the IO control interface through the class `CtlToDatIo`, lines 6-8 in Listing 4.2.

Listing 4.2 – Port in Chisel language.

```
1 class DpathIo(width : Int) extends Bundle {
2    val reset_vector = Input(UInt())
3    val dec_inst   = Output(UInt())
4    val ctl = Flipped(new CtlToDatIo())
5 }
6 class CtlToDatIo extends Bundle() {
7    val stall   = Output(Bool())
8    val if_kill = Output(Bool())
9 }
```

Data Types. The collection of bits is represented by `Bits` type and the Boolean values by the `Bool` type. Chisel also provides `Bundle`, `Vec`, which are a compound type to define collections of values with named fields. The `Bundle` type contains Data types indexed by name, while `Vec` is an indexable vector of data types. The code below, Listing 4.3 is a snapshot of the Rocket processor. The `Bundle` and `Vec` presented, lines 2-3 are nested in order to build complex data structures for the registers `ex_ctrl` and `ex_reg_rs_bypass`. Each field of the class `IntCtrlSigs` is a register type, lines 4-8. For example `ex_ctrl_fp` is a register type in `Rocket` module.

Listing 4.3 – Data Types in Chisel language.

```
1 class Rocket(width : Int) extends Module {
2   val ex_reg_rs_bypass = Reg (Vec(width, Bool()))
3   val ex_ctrl = Reg (new IntCtrlSigs)
4   class IntCtrlSigs extends Bundle {
5     val fp = Bool()
6     val branch = Bool()
7     val alu_fn = Bits(width)
8     val mem_cmd = Bits(width)
9   }
10 }
```

Registers. In Chisel, there are several ways to declare and specify the connection to the register input. As shown in RISC-V Sodor processor in Listing 4.1, the variable `if_reg_pc` is a register declared in the module `DatPath` using the primitive `RegInit`, initialized with a value `io.reset_vector`, line 4, and updated then in `when` construction that defines a conditional block, `C1`, line 8. Therefore, we pass the next value to the register `wb_reg_inst` using the primitive `RegNext`, it will clock the new value every cycle unconditionally, line 11. Note that `UInt` design unsigned Int type for the registers `mem_reg_inst`, `wb_reg_inst`, lines 5-6.

Other Constructs. A Chisel module features combinational and/or sequential circuitry design elements using design primitives. The operator := executes a mono-directional connection element-wise from right to left, while the operator <> is a bidirectional (i.e., bulk) connection element-wise.

Listing 4.4 – Other Constructs in Chisel language.

```
1 class Rocket(width : Int) extends Module {
2    val ibuf = Module(new IBuf())
3    ibuf.io.imem <> io.imem.resp
4    when (!ctrl_killd) {
5      ex_reg_pc := ibuf.io.pc
6    }
7 }
```

The operator <> is used to connect two modules, `Rocket` and `IBuf`, in Listing 4.4, line 3. Furthermore, the `when` block is used to perform a conditional register update, whenever the condition evaluates to true.

4.2.2 . FIRRTL - The Intermediate Representation

FIRRTL language also has first-class support for high-level constructs such as vector types, bundle types, conditional statements, partial connects, and modules. These high-level constructs are then gradually removed by a sequence of lowering transformations. Eventually, the circuit is simplified to resemble a structured netlist, which can then be translated into any output language (e.g., Verilog). All FIRRTL constructs interoperate with one another; a simplified circuit is expressed using a subset of the FIRRTL specification.

The FIRRTL representation is internally represented with an Abstract Syntax Tree (AST) structure as described in Figure 4.2, where passes recursively visit nested elements to manipulate the AST. Chapter 2 presents an example of this pass. It consists of IR nodes represented by objects, each of which is a subclass of the following IR abstract classes: circuit, module, port, statement, expression, or type. The circuit is the root node of any FIRRTL data infrastructure, it consists of a set of modules. The statement node describes the module body and includes the combinatorial and sequential logics as registers, connections, conditional blocks, etc. Each register update is represented by a connect node that is also part of the statement class. Expressions represent references to declared components or logical and arithmetic operations.

Like the software compilers, the FIRRTL compiler framework transforms the designs to the RTL level. It enables this automatic design transformation through optimizations and other generic passes. Thus, the main features of FIRRTL-based hardware compilation are its reusability and its extensibility of customization. As such, we can develop and integrate custom passes in this

toolchain at different levels. Then, the compiler supports a robust mechanism for communicating information throughout the compilation process to the lower level. In this context, a question would be to find what are the main differences between the FIRRTL levels, and which level is the most appropriate to integrate our custom analyses via compilation passes. More details are addressed in the next section.

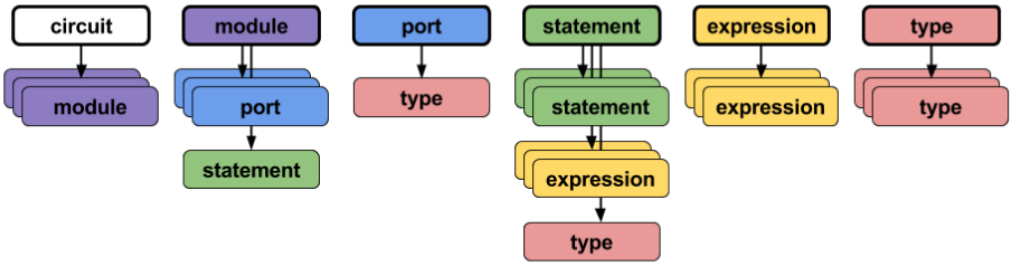

Figure 4.2 – FIRRTL Abstract Syntax Tree (AST).

4.3 . Pipeline Construction in the Chisel/FIRRTL Framework

We describe in this section a description of the compilation framework issues and features in Sections 4.3.1, and 4.3.2, followed by the presentation of Chisel-based RISC-V processor designs for the goal of reporting the semantics language complexity in Section 4.3.3.

4.3.1 . Choice of FIRRTL Forms

FIRRTL proposes different Intermediate Representations (IRs), named forms (i.e., high, mid, low). Each form uses a smaller, stricter and simpler subset of the Chisel language features and defines different transformations to generate the next (lower) form. Compiling FIRRTL to Verilog is developed as a set of passes that implement optimizations, such as constant folding or dead-code elimination. A so-called *high form* supports the Chisel high-level constructs such as vector types, bundle types, and conditional statements that must be lowered for synthesis. These constructs are replaced by a set of low-level features, resembling a structured netlist that simplifies its translation to Verilog, in the form named *low-form*. *mid-form* and *low-form*, are increasingly restrictive subsets of the FIRRTL language that omit many of the higher level constructs.

We now describe the differences between the high and low forms, through several cases results from FIRRTL compilation. We also highlight some compilation issues from Chisel to FIRRTL representation.

High/Low forms: Conditional Statements. The high form supports high-level constructs such as conditional statements as `when` conditions, while in the low form, these kind of statements go through multiple optimizations producing instead a simplified and flattened set of multiplexers. The aim is to be closer to hardware representation, as translated into Verilog.

We illustrate now this case, through a very simple example presented by Listing 4.5. This example defines two registers, `reg1` and `reg2`, at lines 1 and 2. Both registers are initialized on reset using the `RegInit` construct. Note that the reset and clock signals are implicit in Chisel (but can be explicit in the FIRRTL forms). Finally, both registers are updated at lines 3 and 4, but for register `reg2`, this update depends on the value of the `cond` variable (actual value not defined to simplify) and is thus performed within a Chisel `when` construct (line 4).

Listing 4.5 – A simple Chisel code.

```
1 val reg1 = RegInit(0.U(4.W))
2 val reg2 = RegInit(0.U(4.W))
3 reg1 := value
4 when (cond) { reg2 := reg1 }
```

Listings 4.6 and 4.7 describe, respectively, the FIRRTL high form and low form obtained from the Chisel code presented in Listing 4.5.

Listing 4.6 – FIRRTL high form from Listing 4.5.

```
1 reg reg1 : UInt<4> , clock with :
2     reset => (reset, UInt<4> )
3 reg reg2 : UInt<4> , clock with :
4     reset => (reset, UInt<4> )
5 reg1 <= value
6 when cond :
7     reg2 <= reg1
```

Listing 4.7 – FIRRTL low form from Listing 4.5.

```
1 reg reg1 : UInt<4> , clock with :
2     reset => (UInt<1> , reg1)
3 reg reg2 : UInt<4> , clock with :
4     reset => (UInt<1> , reg2)
5 node _GEN_0 = mux(cond, reg1, reg2)
6 reg1 <= mux(reset, UInt<4> , value)
7 reg2 <= mux(reset, UInt<4> , _GEN_0)
```

It can be noticed that the `when` statement remains in the high form (line 6, Listing 4.6), while it is translated into a multiplexer in the low form (line 5,

Listing 4.7). Note that in Chisel, a condition can be translated into a set of multiplexers to implement a multi-variable condition.

Such a statement can be safely translated into a formal statement when targeting a pipeline-stage abstract model. Thus, the higher-level form facilitates the integration of our design-level properties, which demonstrates the need to develop custom FIRRTL passes in high-level form.

4.3.2 . Chisel to FIRRTL Compilation Issues

Chisel is compiled in FIRRTL representation through multiple passes. Our analysis is positioned in FIRRTL high-level form. We present next the main issues and differences through the compilation from Chisel constructions to FIRRTL high form.

RegNext Construct Compilation. The register update can be expressed in Chisel in different ways, as explained in Section 4.2.1. The Chisel construction `RegNext` is employed in order to delay one cycle the associated register update. In order to ensure this delay through compilation, an additional register is implemented for each delay at the FIRRTL high-level form.

We illustrate now this implementation through an example in the KyogenRV processor described in Listing 4.8. The registers `imem_read_sig`, `imem_req` are two Boolean registers, where the second one is updated under the conditional statement `when` with `RegNext` construction with the value of the first register respectively (line 4). We notice that at the FIRRTL high-level form in Listing 4.9, an additional register `REG_1` appears in conditional block `when`, lines 6-8. It ensures the one-cycle delay for the register `imem_req` update. Generally, these additional registers are identified by a general structure `REG` in our analysis. This issue can produce an additional register dependence, which leads to an inaccurate analysis. However, we can handle this issue by relying on these registers FIRRTL names.

Listing 4.8 – Chisel code of Kyogen processor.

```
1 val imem_read_sig: Bool = RegInit(false.B)
2 val imem_req: Bool = RegInit(false.B)
3 when (cond) {
4  imem_req := RegNext(imem_read_sig)
5 }
```

Bulk Connection Compilation. The bidirectional connection is performed using the `bulk` connection constructor provided by the Chisel language. It is expressed by the operator <> and allows components with aggregate types

Listing 4.9 – FIRRTL high form from Listing 4.8.

```
reg imem_read_sig : UInt<1> , clock with :
      reset => (reset, UInt<1> ("h0"))
reg imem_req : UInt<1> , clock with :
      reset => (reset, UInt<1> ("h0"))
when cond:
  reg REG_1 : UInt<1> , clock with :
    reset => (UInt<1> ("h0"), REG_1)
  REG_1 <= imem_read_sig
  imem_req <= REG_1
```

to be connected in a type-safe manner with a single statement. However, the FIRRTL compiler removes this `bulk` connection through transformations. These passes rewrite the `bulk` connection into a series of individual connections. To illustrate this feature, we present a snapshot of the Chisel code in Rocket processor in Listing 4.10, and its corresponding FIRRTL code is described in Listing 4.11.

Listing 4.10 – Chisel code of Rocket processor

```
class FrontendResp extends Bundle {
  val pc = UInt(32)
  val data = UInt(40) }
class IBuf extends Module {
  val io = IO(new Bundle {
    val imem = Flipped(Decoupled(new FrontendResp))
    .....
  })}
class FrontendIO extends Bundle {
  val resp = Flipped(Decoupled(new FrontendResp))}
class Rocket extends Module {
  val io = IO(new Bundle {
      val imem = new FrontendIO
      .....})
  val ex_reg_pc = Reg (size)
  val ibuf = Module (new IBuf)
  when (cond1) {
    ex_reg_pc := ibuf.io.pc }
  ibuf.io.imem <> io.imem.resp
}
```

Listing 4.11 – FIRRTL high form from Listing 4.10.

```
1  module IBuf:
2   output io:
3     flip imem: {flip ready: UInt<1>, valid: UInt<1>,
4                 bits: {pc: UInt<40>, data: UInt<32>}
5  module Rocket:
6   output io:
7     imem: flip resp: {flip ready: UInt<1>, valid: UInt<1>,
8                       bits: {pc : UInt<40>, data: UInt<32>}}
9   inst ibuf of IBuf
10  when cond1:
11    ex_reg_pc <=ibuf.io.pc
12  ibuf.io.imem.bits.pc <=io.imem.resp.bits.pc
13  ibuf.io.imem.bits.data <=io.imem.resp.bits.data
```

The `Rocket` module includes the module `IBuf`, in Listing 4.11, line 9, using the instance element `inst`. Each module has its IO interface grouped in `Bundle` type, in Listing 4.10, lines 5, and 12. Each field is a simple input/output or an instance of another defined IO class. For `Rocket` module, the value `imem` is an IO instantiated from the class `FrontendIO` and then `FrontendResp`, line 6. This IO implementation is performed through the compound type of the IO interface named `DecoupledIO`. `DecoupledIO` is an interface for data transfer. It has three signals or elements: `valid and bits` are output signals controlled by the source, the first is activated if there is data to transfer and the second corresponds to the data. `ready` is an input signal controlled by the receiver and indicates that it is ready to receive the data. For the modules `Rocket` and `IBuf`, the class `FrondendResp`, lines 1-3, is an IO interface used as `DecoupledIO`, lines 6, 13, and 10.

In order to connect the IO interface of two modules, the bulk-connection <> is implemented in the module `Rocket`. This connection is shown with red arrows in Listing 4.10, line 19. The connections of this operator are then compiled into individual connections of each construction field, in lines 12-13 in Listing 4.11. Each individual field `pc` and `data` in `Rocket` and `IBuf` modules are connected then in high form FIRRTL level. The interfaces between modules are implemented using this kind of connection components. Thus, their compilation into FIRRTL high form level leads to connecting the constructions of the modules interfaced. However, it can add a cost in terms of analysis runtime.

4.3.3 . Description of Chisel-Based Processor Pipelines

The pipeline architecture in Chisel-based processors also includes a control path and datapath. Using the modularity of Chisel/Scala, Chisel hardware designers can develop the control path and datapath in separate modules that interact with each other. As such, it facilitates the management of the

abstract pipeline models expected to generate, with a clear overview of the pipeline design. Furthermore, we can analyze both control and datapath with the aim of generating timing models without focusing on the functional aspect. We introduce next several hardware design features using code snapshots of RISC-V processors designed through the Chisel/FIRRTL compilation framework. We intend to describe how the pipeline architecture is built in Chisel-based processors. We detail the pipeline structure in Chisel, the connection between pipeline stages through the connection between pipeline registers, and how the micro-architecture mechanisms such as forwarding are implemented in Chisel.

We present first the pipeline design structure of different Chisel-based processors. We exemplify them with the mono-module pipeline of KyogenRV processor [29], then the multi-module and hierarchical pipeline design of the Rocket processor [2] and the flat and parametric pipeline construction of the Fuxi processor [30]. The multi-module pipeline refers to the pipeline design specified in several FIRRTL modules. We present how Scala constructions added in Chisel implement the connection between these modules. Finally, we present two different implementations of the forwarding mechanism, as developed in RISC-V Sodor [1] and Fuxi [30] processors.

KyogenRV Pipeline. The pipeline design in the Kyogen processor is presented as a 5-stage (IF, ID, EX, MEM, WB) with a mono-module architecture. Its pipeline architecture is presented in a single Chisel/FIRRTL module as shown in Figure 6.7.

Listing 4.12 describes a simplified Chisel pipeline code of the KyogenRV processor. The pipeline datapath is specified in a single Chisel module named `KyogenRVCpu`. The pipeline registers are all defined inside this module, lines 2-5. The sequential logic represented by `registers` is updated through the combinatorial logic using `when` conditional block, lines 6-8. Thereafter, the connection between pipeline stages is established.

Listing 4.12 – Simplified Chisel code of Kyogen datapath pipeline.

```
1  class KyogenRVCpu:
2     val if_pc: UInt = RegInit()
3     val id_inst: UInt = RegInit()
4     val id_pc: UInt = RegInit()
5     val ex_pc: UInt = RegInit()
6     when(cond1) {
7          id_pc := if_pc
8          id_inst := io.r_imem_dat.data
9     }
10    ex_pc := id_pc
```

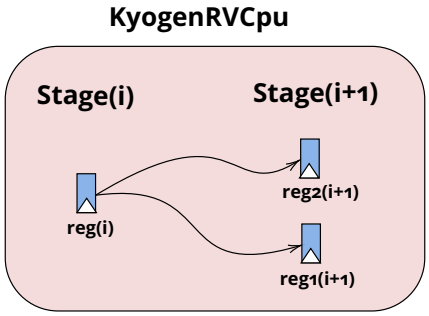

Figure 4.3 – Datapath pipeline module of Kyogen processor.

Rocket Pipeline. The Rocket core is a 5-stage pipeline (IF, ID, EX, MEM, WB). However, it is sometimes presented as a 6-stage pipeline with a separate stage implementing the branch prediction algorithm. The Chisel design of Rocket consists of several modules grouped into two parts – `Frontend` and `Rocket`, corresponding to the pipeline frontend and the pipeline backend respectively, as shown in Figure 4.4. More specifically, the first two stages, PC-generator and IF, are implemented in the `Frontend` module, while the remaining four pipeline stages (i.e., from ID to WB) are in the `Rocket` module. Figure 4.4 also presents the connections between the pipeline design elements (e.g., registers or input/output interfaces); the corresponding code snippets are in Listing 4.13 in Chisel and in Listing 4.14, in FIRRTL high form. From the code organization point of view, the pipeline is developed in five modules: `Rocket`, `IBuf`, `Frontend`, `ShiftQueue` and `RocketTile`.

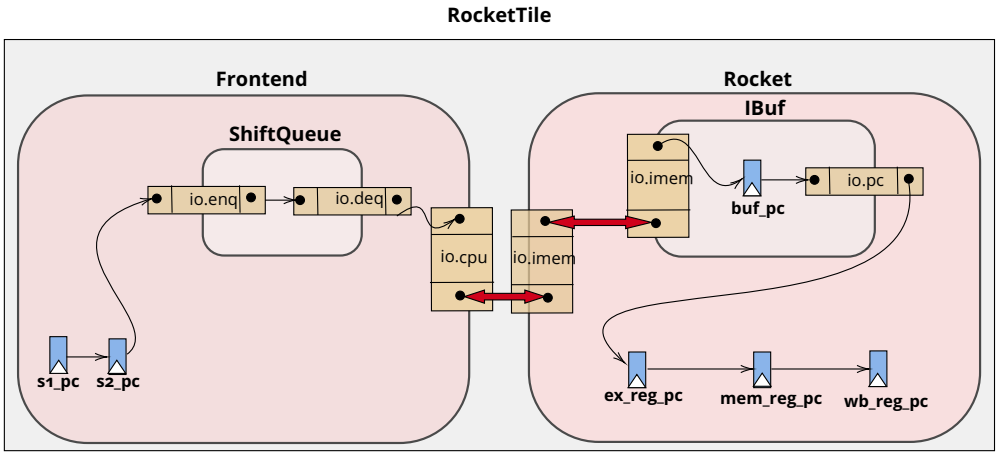

Figure 4.4 – Datapath pipeline modules of Rocket core.

Listing 4.13 – Simplified Chisel code of Rocket datapath pipeline.

```
class RocketTile:
  val core = Module(new Rocket)
  frontend.module.io.cpu <> core.io.imem
------------------
class Rocket:
  val ex_reg_pc = Reg (size)
  /* decls: mem_reg_pc and wb_reg_pc */
  val ibuf = Module (new IBuf)
  when (cond1) {
    ex_reg_pc := ibuf.io.pc
    /* updates: mem_reg_pc and wb_reg_pc */ }
  ibuf.io.imem <> io.imem.resp
------------------
class Frontend:
  val io = IO(... /* cpu field */)
  /* decls: s1_pc and s2_pc */
  s2_pc := s1_pc
  val fq = Module (new ShiftQueue)
  fq.io.enq.bits.pc := s2_pc
  io.cpu.resp <> fq.io.deq
------------------
class IBuf:
  val io = new Bundle {/* imem and pc fields */}
  val buf = Reg (io.imem.bits)
  buf.pc := io.imem.bits.pc
  io.pc := Mux (cond2, buf.pc, io.imem.bits.pc)
------------------
class ShiftQueue :
  io.deq.bits := io.enq.bits
```

Listing 4.14 – High-form FIRRTL for Listing 4.13.

```
module RocketTile:
 inst frontend of Frontend
 inst core of Rocket
 core.io.imem.resp.bits.pc <=
        frontend.io.cpu.resp.bits.pc
---------------
module Rocket:
 inst ibuf of IBuf
 when cond1 :
   ex_reg_pc <=ibuf.io.pc
```

```
11    /* updates: mem_reg_pc and wb_reg_pc */
12   ibuf.io.imem.bits.pc <=io.imem.resp.bits.pc
13   ibuf.io.imem.bits.data <=io.imem.resp.bits.data
14  ---------------
15  module Frontend:
16   output io: {.../* cpu field */}
17   s2_pc <=s1_pc
18   inst fq of ShiftQueue
19   fq.io.enq.bits.pc <=s2_pc
20   /* explicit update, field by field, of <>*/
21  ---------------
22  module IBuf:
23   output io : {/* imem and pc fields */}
24   reg buf.pc : UInt<40>
25   buf.pc <=io.imem.bits.pc
26   node _io_pc_T_1 =
27       mux(cond2, buf.pc, io.imem.bits.pc)
28   io.pc <=_io_pc_T_1
29  ---------------
30  module ShiftQueue:
31   io.deq.bits.pc <=io.enq.bits.pc
```

Listing 4.13 describes how the pipeline modules are connected. Thus, we illustrate the connection between the registers PC (i.e., those with `_pc` suffix) of all the pipeline modules in different stages. The `RocketTile` module is designed to connect the modules `Frontend` (not shown) and `Rocket`, in line 2 using their respective IO interface (`frontend.module.io.cpu <> core.io.imem`), in line 3. This connection is shown with red arrows in Figure 4.4 and implemented using the bulk operator <> in Chisel. The PC registers `ex_reg_pc`, `mem_reg_pc`, `wb_reg_pc` are described in the same module, i.e., `Rocket`, while the PC register of the decode stage is presented in the `IBuf` module. `IBuf` is instantiated in `Rocket` in order to connect the decode and execute stages through the output `io.pc`, which receives the output of the register `buf.pc` in the `IBuf` module, in lines 28-30 in Listing 4.7.

The registers of fetch and decode stages are connected through the interfaces between the corresponding modules. The register `buf.pc` of the decode stage, in the `IBuf` module, is interfaced with the input port `io.imem.bits.pc`, in line 27 in Listing 4.7. The PC registers `s1_pc` and `s2_pc` of the fetch stage are in the `Frontend` module, and `s2_pc` is connected to the output port `fq.io.enq.bits.pc` which is an element of the `ShiftQueue` module, in line 21 in Listing 4.7. Here, the `ShiftQueue` module is used as an I/O interface as it connects the input `io.enq` to `io.deq` which is connected to the output `io.cpu.resp`. Thus, the fetch stage register `s2_pc` is connected to the output `io.cpu.resp`, while the input `io.imem.bits.pc` is connected to the decode

stage register `buf.pc`. Finally, the connections between registers in pipeline stages are obtained through the IO interface of the module `RocketTile`.

Fuxi Pipeline. Fuxi is a 32-bit pipelined RISC-V processor, designed to run simple operating systems and bare-metal software. In contrast to the hierarchical pipeline design of Rocket, in Fig. 4.4, the pipeline design of Fuxi is flattened, in Fig. 4.5. More precisely, each pair of pipeline stages (e.g., `Fetch` and `Decode`) is connected through an interface I/O named `StageIO`, described in a module `MidStage`.

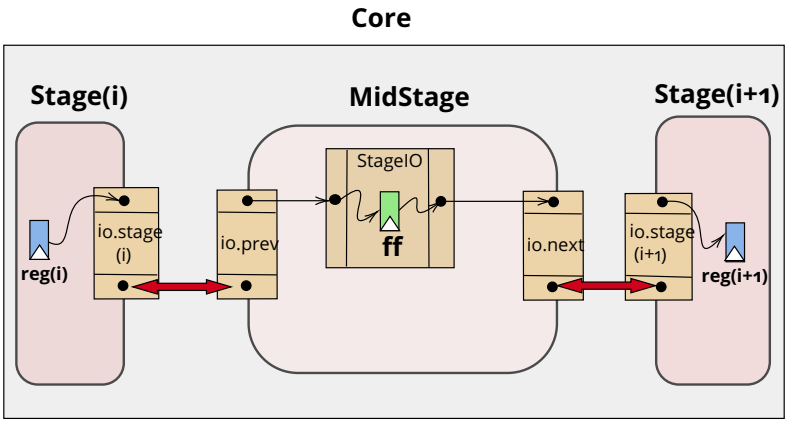

Figure 4.5 – Datapath pipeline modules of Fuxi.

Listing 4.15 shows a simplified Chisel implementation of how the pipeline modules of Fuxi are connected. More precisely, the module `Midstage` is instantiated several times, using the `StageIO` interface, creating the pipeline connectors between stages, in lines 2-3 of Listing 4.15. The connection of two pipeline stages is through the I/O interface of `Midstage`, which features two ports, `prev` and `next`, lines 14-15. Moreover, `Midstage` contains a bundle register named `ff`, where each field is a simple register.

Listing 4.15 – Simplified Chisel code of Fuxi processor.

```
1  class Core:
2    val ifid   = Module(new MidStage(new FetchIO))
3    val idex   = Module(new MidStage(new DecIO))
4    /* Idem for AluIO and MemIO */
5  ------------------
6  class StageIO extends Bundle
7  class FetchIO extends StageIO:
8      ...  val pc = UInt(Width)
9    /* Idem for DecoderIO, AluIO and MemIO */
10 ------------------
```

```
11 class MidStage (StageIO):
12   val io = IO(new Bundle {
13     /*IO of previous/next stage*/
14     val prev     = Input(StageIO)
15     val next     = Output(StageIO)
16   })
17   val ff = Reg(StageIO)
18   when (cond) {
19    ff := io.prev
20   }
21   io.next := ff
```

Forwarding of Sodor and Fuxi. Sodor is a family of processors of different pipeline depths, from 1- to 5-stage. We present, in Listing 4.16, a snippet of Sodor 5-stage processor design, i.e., a classical IF to WB pipeline. More specifically, we focus on its forwarding mechanism from the later pipeline stages to the EX stage. The input to register `exe_rs2_data` of EX is updated, in line 9, with the result of the forwarding, through the wire `dec_rs2_data`, in lines 4-7. This forwarding is coded with a Chisel construct `MuxCase` – a syntactic sugar for a cascade of multiplexers with conditions `C1-C3`, lines 5-7, where each selection dependency is represented as a tuple in a Scala array. At FIRRTL level, the `MuxCase` construct is compiled into nested `Mux`, for each selection.

Listing 4.16 – Forwarding implementation in the pipeline of Sodor.

```
1 val exe_rs2_data = Reg (size)
2 val dec_rs2_data = Wire (size)
3 /* C1-C3: enable and selection signals */
4 dec_rs2_data := MuxCase (rf_rs2_data,
5    Array (C1 → exe_alu_out,
6           C2 → mem_wbdata,
7           C3 → wb_wbdata))
8 when (C4) /* C4: no stalling condition */ { ...
9   exe_rs2_data := dec_rs2_data }
```

The forwarding of Fuxi is implemented using a Scala function `forwardReg` with IO interfaces as parameters, i.e., the register file `rf` and the operand `read`, as shown in Listing 4.17, line 1. Fuxi proposes an alternative implementation of the forwarding condition checks `C1-C3`, namely the `MuxCase` implementation of Sodor is replaced by `when-.elsewhen` statements of Chisel, in lines 3-10. Each forwarding path is then created using calls to `forwardReg`, in lines 13-14. This function is called for two operands of the instruction in the Decode stage: `io.regRead2/io.regRead1`. Thus, the data `read.data` is updated under the when condition if it matches execute, memory, or write back addresses.

At the FIRRTL level, the hardware compilation framework flattens all the

abstract functions and takes into account the function calls with the call parameters instead. The Listing 4.18 describes the corresponding FIRRTL high form of the function `forwardReg(io.regRead1, io.rf1)` called in Listing 4.17, line 13.

Listing 4.17 – Forwarding implementation in the pipeline of Fuxi.

```
1  def forwardReg(read: RegReadIO, rf: RegReadIO) {
2      /* C1-C3 : condition signals */
3      when (C1) {
4          read.data := io.aluReg.data
5        } .elsewhen (C2) {
6          read.data := io.memReg.data
7        } .elsewhen (C3) {
8          read.data := io.wbReg.data
9        } .otherwise {
10         read.data := rf.data
11       }
12 }
13 forwardReg(io.regRead1, io.rf1)
14 forwardReg(io.regRead2, io.rf2)
```

Listing 4.18 – FIRRTL high form of forwarding implementation in the pipeline of Fuxi.

```
1  node _T = neq(io.regRead1.addr, UInt<1> ("h00"))
2      node _T_1 = and(io.regRead1.en, _T)
3      when _T_1 :
4        node _T_2 = eq(io.regRead1.addr, io.aluReg.addr)
5        node _T_3 = and(io.aluReg.en, _T_2)
```

The code snippets in Listing 4.13 to Listing 4.18 present processor design choices addressed using Scala/Chisel programming language-specific constructs. Since Chisel is built over Scala, it is often the case that processor designs mix constructs of the two languages. For example, the forwarding of Sodor combines a Chisel `MuxCase` with a Scala `Array`, while the forwarding of Fuxi embeds a Chisel `when-.elsewhen` in a Scala function. Furthermore, all the connection between pipeline modules is performed using the interface IO. They are implemented in Chisel with the bulk connection construct.

Different descriptions of Chisel-based processors are presented in this section, such as the pipeline structure and the various ways to implement the forwarding mechanism at Chisel and its compilation at the FIRRTL level. We have shown that relying on the FIRRTL level allows us to capture the Chisel diversity in our analysis. Furthermore, we highlight the complexity of the hardware compilation framework to compile the Chisel constructions.

4.4 . Conclusion

In this chapter, we have detailed our proposed workflow, which is to be used to formally prove timing properties. We have proceeded first to present the selected high-level HDL language and its compilation framework Chisel/-FIRRTL, as the expected input of our workflow. We have introduced then the Chisel constructions that we rely on to develop our approach in Section 4.2. The main goal of Chisel HDL is to allow complex hardware designs from simple components using an advanced module system (in a combination of Scala's features and Chisel's specialized operators and language constructs). In essence, Chisel proposes syntactic sugars to represent a hardware design as a series of primary component updates (wires, registers) under various static and dynamic evaluation contexts (classes, objects, functions, when-blocks etc.). Chisel is compiled in FIRRTL representation which has three levels (high, mid, low), and designs written in Scala are typically transformed into the high form of FIRRTL, which goes through multiple optimizations passes to a low form. Since we aim to formally verify the timing properties, we need to construct the micro-architecture models with an appropriate level of abstraction. Thus, we implement our analysis at the high form of FIRRTL, where the designs are not flattened and preserves the Chisel semantics as `when` conditional blocks (that FIRRTL low form flattens their structure and uses a `Mux` instead, making the condition extraction more difficult). These Chisel statements describe the different execution paths under conditions that go with timing properties analysis.

The complexity of the hardware compilation framework and the diversity of Chisel design implementations presented in section 4.3 confirm the need for studies of the hardware compilation features/issues. Furthermore, the code analysis of Chisel HDL processor designs (e.g., pipeline structure - stages, register forwarding, etc.) is the first brick in our approach. Therefore, the abstract pipeline model needs to include the pipeline structure in terms of single/multi-module pipeline, also the hierarchy of the pipeline in order to ensure the efficiency and strength coverage of our approach.

We intend to address our approach that generates the abstract pipeline models in the next Chapters 5, and 6.

5 - Register Analysis of Pipeline Designs

The construction of the complete abstract pipeline models requires both datapath and control path models as described in the workflow in Figure 4.1. Generally, these pipeline models are developed manually and their development is based on the manuals provided by the manufacturers of the hardware platforms. However, in our work, we aim to automatize the construction of abstract datapath pipeline models combined with manual control logic to define the execution models on these architectures.

Each datapath pipeline design comprises several components (ALU, CSR, Memory, Registers, etc.) that perform all the required operations. Since our approach aims to verify the timing properties, we construct the abstract datapath models with a focus on the timing aspect of the pipeline; namely, we focus on the pipeline stages and, more precisely, the pipeline registers.

In this chapter, we describe how we can construct datapath pipeline models automatically, starting with a register analysis. We present the registers analysis for different pipeline structures in Section 5.1. We apply this analysis on RISC-V Chisel-based processors and detail the results in Section 5.2. Then, we summarize the experimental results on these RISC-V processors in Section 5.3.

5.1 . Register Analysis Formalization

In order to construct a pipeline datapath model for the timing analysis, we need to determine the pipeline depth (i.e., the number of pipeline stages) as well as the connections between these stages. A pipeline stage is to be identified by a set of registers, which are updated whenever the pipelined execution passes from a stage to another.

Intuitively, the construction of a pipeline datapath proceeds as follows: we compile the Chisel code of hardware designs into the corresponding FIRRTL representation (AST). Then, we proceed with the development of our custom pass, which allows us to process the FIRRTL AST and build the desired datapath pipeline abstraction. Writing a FIRRTL pass usually requires writing functions that traverse the FIRRTL data structure (AST) to either collect information or replace IR (i.e., Intermediate Representation) nodes with new IR nodes. The circuit is the root node of any FIRRTL AST, and there is only one Circuit. The circuit contains a list of module definitions and the name of the top-level module. The pipeline modules are given as input, and for each register in these modules, we investigate each module in order to perform a register analysis to determine precedence relations between registers. We also perform an output port analysis to determine how registers from different modules

will be connected. Finally, we provide an assignment procedure to map each register to a pipeline stage according to the results of the previously mentioned analyses. This phase is more detailed in the next Chapter 6. Next, we introduce some notations and definitions, following a standard set-theoretic approach.

Notations. We consider $\mathcal{P}r = \{M_1, M_2, ..., M_n\}$, a processor design, defined by a set of n modules M_i and $\mathcal{P}$ the pipeline of $\mathcal{P}r$, with $\mathcal{P} \subseteq \mathcal{P}r$. We assume that $\mathcal{P}$ is given and the (FIRRTL) AST of $\mathcal{P}$ is denoted by $AST_{\mathcal{P}}$, so as AST_M represents the AST of each $M \in \mathcal{P}$.

Furthermore, for each module $M \in \mathcal{P}$ we define:

- $Insts$, the set of instanced modules,
- I/Os, the set of input/output ports,
- $Regs$, the set of registers (or sequential logic),
- $Combs$, the set of combinatorial elements (e.g., wires, multiplexers, etc.),
- $Ctxs$, the set of contexts (scopes),
- $Exts$, the set of external entities.

Each of these sets, when subscripted by M represents the corresponding set of a module M and, when subscripted by $\mathcal{P}$, represents the union of all the corresponding sets of the pipeline modules in $\mathcal{P}$.

An element $ins \in Insts_M$ identifies the name and the type of a module instantiated in M. The module of $\mathcal{P}$ which contains directly or indirectly (through transitivity) the instances of all the other pipeline modules in $\mathcal{P}$ is called 'top', denoted by $\top$.

Furthermore, an element $x \in I/Os$, $Regs$ or $Combs$ is the identifier of the respective design element, i.e., x is a name of an input/output port or a register or a wire.

An element $ctx \in Ctxs_M$ is defined as a mapping between the context condition and the respective register updates in M. A context, which is expressed in Chisel with a `when` statement, allows guarded register updates (or any other combinational element) in a compact manner. Moreover, `when` is often used to provide alternative updates of the same register, updates which are guarded by different conditions.

Our approach considers a given set of pipeline modules $\mathcal{P}$, which is a subset of $\mathcal{P}r$, the complete processor design.

We denote by $Exts$, the set of all design elements (i.e., ports, registers, wires, etc.) which are defined in $\mathcal{P}r \setminus \mathcal{P}$. For a module $M \in \mathcal{P}$, we define by $Exts_M$, the set of external design elements which are used in M. For example, $Regs_M$ consists of only the registers defined in module M, whereas the "external" registers of M (e.g., defined in other modules but used in M) are to be included in $Exts_M$. A similar argumentation stands for I/Os_M and $Combs_M$ with ports and respectively, wires being also included in $Exts_M$.

5.1.1 . Register Analysis Algorithm

We present next the algorithm to construct pipeline datapath models from Chisel-based processor designs. As such, we introduce an intermediate representation of the processor pipeline $\mathcal{P}$, based on the set of pipeline registers $Regs$, a set of dependency relations between these registers, and a set of contexts $Ctxs$. The dependencies between registers are used to produce chains of registers, and the contexts are used to determine additional relations between these registers, encoding how the processor is coded. Intuitively, this intermediate representation is a non-strongly connected graph with registers as nodes and their dependencies as edges. Also, different contexts determine different connected components in this graph.

The sets $Regs$, $Combs$, I/Os and $Ctxs$ of $\mathcal{P}$ are obtained, for each module $M \in \mathcal{P}$, from the $AST_{\mathcal{P}}$, using the standard visitor from [105]. We name this operator

$$\texttt{process\_mod} : AST_M \rightarrow I/Os \times Regs \times Ctxs.$$

The initial partition of the processor design $\mathcal{P}r$ into the pipeline design P is also sufficient to determine the set $Exts$.

The following Algorithm 1 illustrates our AST visitor used to construct the sets $Regs$, $Combs$, I/Os and $Ctxs$ of $\mathcal{P}$. Algorithm 1 is a recursive algorithm that visits all FIRRT nodes in the AST. First, we start with the root node circuit of our FIRRTL AST, line 2. The circuit is composed of a set of modules. We visit each module with the function `walkModule`, lines 5-7. For each visited module, we visit the module body which includes the set of statements and ports through `walkStatement`, lines 9-15 and `walkPort` functions, lines 17-20. The statement node contains the registers defined as `DefRegister` node, conditions through `Conditionally` node, updates and assignment of variables as `Connect` node, etc., lines 12-14. Moreover, the ports can be an input or an output module defined as `BundleType` node with a specification of the direction, line 19. We store the set of registers, contexts, and input/output ports from the nodes browsed. For each visited statement, we visit each of its children statement and expression nodes expressed in `walkExpression` function, lines 22-25. Note that all the FIRRTL transformations use these recursive walks of the FIRRTL AST to modify or simplify the circuit.

To express a recursive walk, every IR node has implemented a custom `map` function. A node's map applies a user-specified function to the subset of children whose node type matches the function's input and return node type. For example, in `walkModule` function, map function will be applied to the module's children which are the statement nodes, and then return the module as the node type, lines 6-7. We recursively walk all FIRRTL modules, statements and expressions by calling map on modules, statements and expressions. This procedure is called the pre-order traversal, as we traverse the FIRRTL AST using `map` to write recursive functions that visit every child of every

node.

Algorithm 1: Process Module Function

Input : $\mathcal{P}$ and $state = AST_{\mathcal{P}}$ - The FIRRTL AST
Output: $Cstrs : Regs, I/O, Ctxs$ - Data structure sets of $\mathcal{P}$

Function `process_mod`($state$)**:**
    $circuit$ = $state$.circuit `/* root node of AST FIRRTL */`
    $Cstrs \leftarrow circuit$ map walkModule `/* browse circuit's modules and store data structure */`
return $Cstrs$;
Function `walkModule`($module$)**:**
    $module$.map(s => walkStatement(s)) `/* browse the module */`
    $module$.map(p => walkPort(p)) `/* go through the ports */`
return $module$;
Function `walkStatement`($stmt$)**:**
    $stmt$.map(s => walkStatement(s)) `/* for children statement */`
    $stmt$ match
        case reg: DefRegister => $Cstrs\_Regs = Cstrs\_Regs \cup \{reg\}$
        case cd: Conditionally => $Cstrs\_Ctxs = Cstrs\_Ctxs \cup \{cd\}$
        case other => skip
    $stmt$.map(s => walkExpression(s)) `/* visit each expression */`
return $stmt$;
Function `walkPort`(p)**:**
    p match
        case bd: BundleType => $Cstrs\_I/Os = Cstrs\_I/Os \cup \{bd\}$
        case other => skip
return p;
Function `walkExpression`(exp)**:**
    exp match
        case mx : Mux => walkExpression(mx)
        case other => skip
return exp;

A depth-first search traversal of the $AST_{\mathcal{P}}$ establishes an order between the considered pipeline modules, determined by the instancing relation, i.e., the set $Insts$. However, for a module that is instantiated several times, e.g., `MidStage` of Fuxi, in Listing 4.15, our approach computes a module summary and uses it for all instances of this particular module. We denote by

$$\texttt{order\_mods} : \mathcal{P} \times AST_{\mathcal{P}} \rightarrow \mathcal{P}$$

an operator which establishes an order between the pipeline modules up to top $\top$ as the last element.

Informally, an analysis for a module has (1) an intra-module component that focuses on the module's registers and (2) an inter-module component

that overcomes the module and focuses on the module's interface (i.e., input-output ports), in order to connect between several pipeline modules.

We address the intra-module part by computing dependencies between registers through a visitor combinator, characterizing the connectivity of each register w.r.t. the other design elements. Operationally, for a given register r in module M, a visitor combinator iteratively collects the nodes of the AST_M which affect the inputs of r. This iterative process corresponds to a standard dataflow analysis which terminates at a register frontier, defined next.

Definition 12 *For a register $r \in Regs_M$, we denote by In_r the* **input frontier** *of r, defined by $\{c_1, c_2, ..c_i.., c_n\}$, where $c_i \in Regs_M$ or $c_i \in I/Os_M$ or $c_i \in Exts_M$.*

Such an input frontier contains three kinds of design elements: registers (i.e., that precede r in the pipeline datapath), ports or other design elements of the considered pipeline $\mathcal{P}$. These are stop conditions of the visitor combinator as it aims to collect, for all registers in each module $M \in \mathcal{P}$, the respective register contexts.

Definition 13 *For a register $r \in Regs_M$, we denote by $\mathcal{C}_r$ the* **register context** *of r, defined by the pair $\langle In_r, out \rangle$ with In_r and out being the input frontier and the output connection of r respectively.*

The register context C_r is computed by an operator

$$\texttt{regs\_ctx} : AST_M \times Regs \times I/Os \rightarrow In \times Regs$$

where In is the set from *Definition 12*.

The goal of the inter-module part of our analysis is to determine, for each pipeline module an input-output interface in order to connect the registers of different modules. Hence, we also define an output frontier of a module.

Definition 14 *For an output port $p \in I/Os_M$, we denote by Out_p the* **output frontier** *of p, defined by $\{c_1, c_2, ..c_i.., c_n\}$, where $c_i \in Regs_M$ or $c_i \in I/Os_M$.*

An output frontier is established for each output port p of a module M which registers or input ports are connected to it. The connections between the module's registers and the output port are particularly important when connecting registers from different modules.

Definition 15 *For an output port $p \in I/Os$, we denote by $\mathcal{C}_{io}$ the* **output context** *of p, defined by (Out_p, p) with Out_p and p being the output frontier and the output respectively.*

The output context C_{io} is computed by an operator

$$\texttt{ios\_ctx} : AST_M \times Regs \times I/Os \rightarrow Out \times I/Os$$

where Out is the set from *Definition 14*.

For a pipeline module M, for all registers $r \in Regs_M$ and output ports $p \in I/Os_M$, the operators `regs_ctx` and `ios_ctx` determine a summary of M. Next, our approach determines the dependency relations between registers. We establish first a precedence relation between two registers then we use these relations to define our working structure.

Definition 16 *For two register contexts $\mathcal{C}_{r1}$ and $\mathcal{C}_{r2}$ of $r1$ and $r2$ respectively, a predicate $\mathbf{prec(r1, r2)}$ is true if $r1 \in In_{r2}$, i.e., in the input frontier of $r2$, and false otherwise. We denote by $Pred$ the set of $(r1, r2)$ with $r1, r2 \in Regs$, for which $\texttt{prec}(r1, r2)$ evaluates to true.*

Definition 17 *The* **intermediate representation** *of a pipeline design $\mathcal{P}$, denoted by $IR_{\mathcal{P}}$ is a non-strongly connected graph $G = (V, E)$ with the set of nodes $V = Regs$ and the set of edges $E = Pred$.*

This graph is generated by an operator

$$\texttt{preced\_regs} : Cr \times Cio \rightarrow I$$

where Cr and Cio are the sets in *Definition 13* and *Definition 15* respectively whereas I is the graph from *Definition 16*.

Definition 18 *The operator* $\texttt{ctx\_reg} : Regs \rightarrow 2^{Regs}$ *is defined as* $\texttt{ctx\_reg}(\mathbf{r}) = R$ *where for each $r_i \in R$, $\exists ctx \in Ctxs$ with $ctx = cond \mapsto Upds$ and $r, r_i \in Upds$.* `ctx_reg` *places registers from different connected components of $IR_{\mathcal{P}}$.*

Algorithm 2: Pipeline datapath construction: Register Analysis

Input : $\mathcal{P}$ and $AST_{\mathcal{P}} = \{\text{AST}_1, ..., AST_n\}$
Output: $Regs \mapsto Stages$ - pipeline datapath of $\mathcal{P}$

1 $o \leftarrow \texttt{order\_mods}(\mathcal{P},\ AST_p)$ `/* order modules based on instancing, ⊤ is last. */`
2 **foreach** $m \in o$ **do**
3 $\quad (I/Os_m,\ Regs_m,\ Ctxs_m) \leftarrow \texttt{process\_mod}(AST_m)$
4 $\quad C_r \leftarrow C_r \cup \texttt{regs\_ctx}(AST_m,\ Regs_m,\ I/Os_m)$
5 $\quad C_{io} \leftarrow C_{io} \cup \texttt{ios\_ctx}(AST_m,\ Regs_m,\ I/Os_m)$
6 $IR_{\mathcal{P}} \leftarrow \texttt{preced\_regs}(C_r, C_{io})$ `/* build intermediate representation of` $\mathcal{P}$ `*/`

Algorithm 2 constructs a pipeline datapath out of a given set of pipeline modules $\mathcal{P}$ while working on the corresponding $AST_{\mathcal{P}}$ representation. Once

the processing order of the pipeline modules is determined, in line 1, the algorithm calculates a module summary, in lines 4-5, for each module of $\mathcal{P}$. The intra-module phase constructs the register context for the relevant module, in line 4. The inter-module phase performs with the output context constructed, in line 5. The last iteration, which is over the top $\top$ module, is followed by the construction of the intermediate representation of $\mathcal{P}$, in line 6, used to generate the pipeline datapath.

5.2 . Application to RISC-V Processors

We present the register analysis application for various RISC-V Chisel-based processors. Section 5.2.1 details the mono-module datapath pipeline application. Then, we continue with the multi-module designs illustration in Section 5.2.2.

5.2.1 . Mono-module Datapath Pipeline Design

The mono-module datapath pipeline design refers to the datapath pipeline specified in a single FIRRTL AST module. Figure 5.1 illustrates the mono-module design with a focus on the main Chisel/FIRRTL construction of register analysis. All the analyses for a mono-module pipeline are performed in the relevant module and delimited by the module interface I/O. For each register in the pipeline module, we construct the register context with a recursive walk through the combinatorial logic until the input frontier as a stop condition. These combinatorial logics present the operations such as those with a single operand `op1` as `NOT`, and with two operands as operations `Mux`, `AND`, `XOR`, `OR` etc.

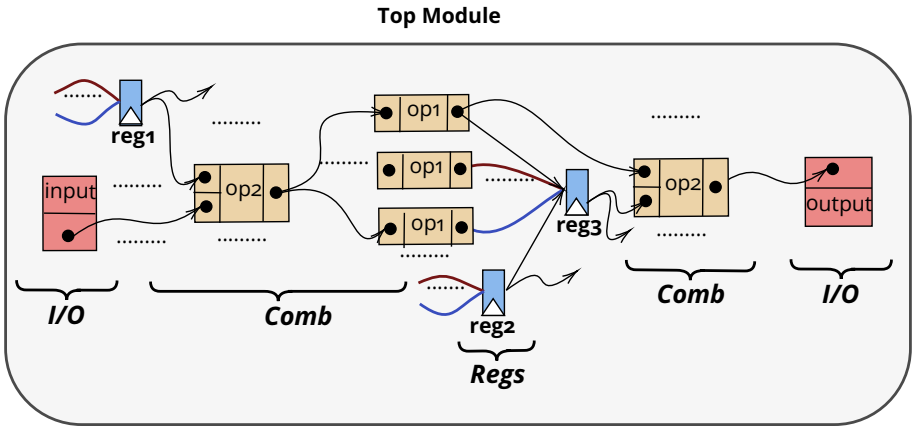

Figure 5.1 – Register analysis for mono-module datapath pipeline design.

We exemplify next the notations and definitions described above with the RISC-V Mini Chisel-based processor. Then, we illustrate the `Process Module Function` (Algorithm 1) through RISC-V Mini FIRRTL AST visitor.

Listing 5.1 – Snapshot of Chisel code of RISC-V Mini processor.

```
1  class Datapath(width) extends Module {
2    val io  = IO(new DatapathIO) /*ctrl, icache fields*/
3    val alu = Module(width)
4     /***** Fetch / Execute Registers *****/
5    val pc   = RegInit(size)
6    val fe_inst = RegInit(size)
7    val fe_pc  = Reg(size)
8    /***** Execute / Write Back Registers *****/
9    val ew_inst = RegInit(size)
10   val ew_pc  = Reg(size)
11   val ew_alu = Reg(size)
12   val inst = Mux(io.ctrl.inst_kill, NOP ,io.icache.resp.bits.data)
13   when (cond1) {
14     fe_pc  := pc
15     fe_inst := inst
16   }
17   when(cond2) {
18     ew_pc    := fe_pc
19     ew_inst  := fe_inst
20     ew_alu   := alu.io.out
21   }
```

We illustrate next the examples of Regs, I/Os, Combs, Ctxs, and Exts.

Example 1 *The registers* `pc`, `fe_inst`, `fe_pc`, `ew_inst`, `ew_pc`, `ew_alu` *are in* $Regs_{Datapath}$, *as in Listing 5.1, lines 5-11. The* $Combs_{Datapath}$ *is represented with the* `Mux` *constructor to assign the wire* `inst`, *lines 12. Figure 5.2 summarizes the RISCV-Mini registers specified in Listing 5.1.*

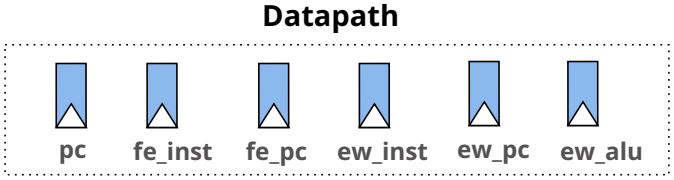

Figure 5.2 – Datapath pipeline registers in RISC-V Mini.

Example 2 *The input/output port is described in the class* `DatapathIO` *in Listing 5.1, line 2. It is a* `Bundle` *type that presents the inputs and outputs of module* `Datapath`. *The module* `alu` *is in* $Exts_{Datapath}$, *in Listing 5.1, line 3.*

Example 3 *The context* `cond1` $\mapsto$ *(*`fe_pc`, `fe_inst`*) is in* $Ctxs_{Datapath}$, *as in Listing 5.1, lines 13-15. Similarly,* `cond2` $\mapsto$ *(*`ew_pc`, `ew_inst`, `ew_alu`*) is in* $Ctxs_{Datapath}$,

in Listing 5.1 which includes three register updates in the same context, lines 17-20. Figure 5.3 represents these two contexts and their registers inside.

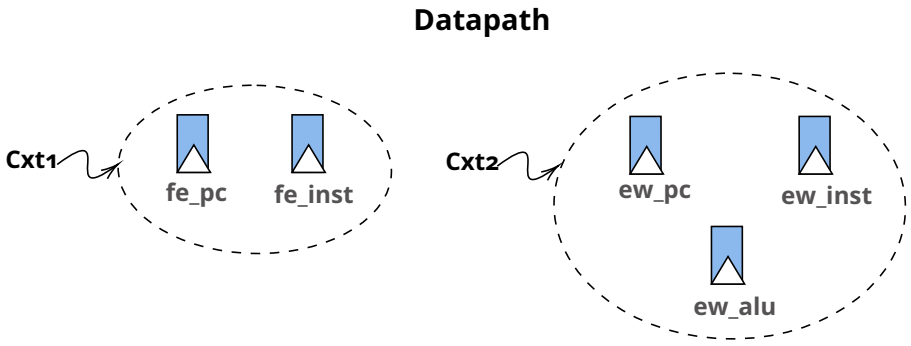

Figure 5.3 – RISC-V Mini contexts.

We illustrate following the examples of input frontier and register context.

Example 4 *The register context of* `fe_inst` *in* $Datapath$ *processor, in Listing 5.1 line 15, includes the constructors* `inst` *which is a* `Mux` *primitive. With a recursive browse of all nodes starting from this register to all inputs frontier, the register context of this register are the pairs* `<io.ctrl.inst_kill, fe_inst>` `<io.icache.resp.bits.data, fe_inst>`*, with module inputs* I/Os *as input frontiers.*

Example 5 *The register context of* `fe_pc, ew_pc, ew_inst` *in* $Datapath$ *processor, in Listing 5.1, lines 14, 18, 19 are the pairs* `<pc, fe_pc>, <fe_pc, ew_pc>, <fe_inst, ew_inst>` *respectively. All their input frontiers are delimited by processor registers.*

Example 6 *The register context of* `ew_alu` *in* $Datapath$ *processor, in Listing 5.1, lines 20, is the pair* `<alu.io.out, ew_alu>` *with* $alu.io.out \in alu \in Exts_M$*, where the input frontier is an element of the external module* `ALU`*.*

Figure 5.4 represents the context of all the registers described in these examples above.

Now, we illustrate the Algorithm 1 application on the RISC-V Mini processor in Figure 5.5. RISC-V Mini processor is mono-module datapath pipeline design; its datapath pipeline is specified on a single module named `Datapath`. Through `process_mod` function, we process this module to get the ports that are stored in `DatapathIO` IO module and the module body which is a set of statements. In order to construct the context registers, for each register, we walk through the statements until input frontiers, which are expression nodes in FIRRTL AST. For example, to construct the context of the register

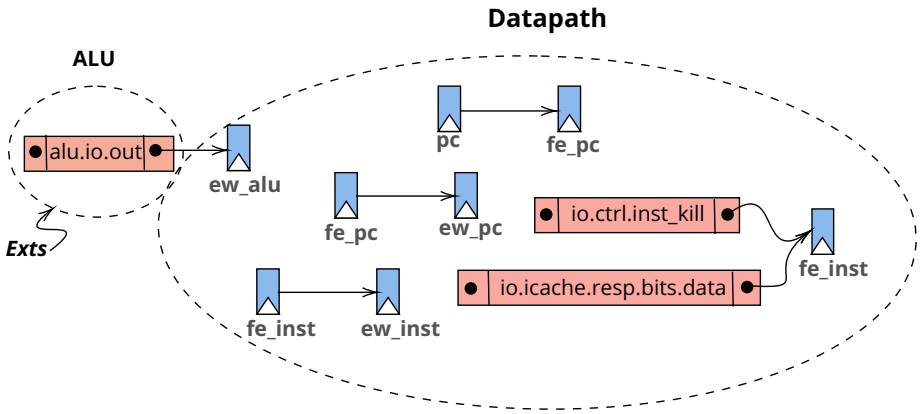

Figure 5.4 – RISC-V Mini registers context.

`fe_inst`, we go through the AST with `walkStatement` first to get the wire `inst`, which is a multiplexer `Mux` and corresponds to an expression node in FIRRTL. Then, we stop at `io.ctrl.inst_kill` as the input frontier through multiple `walkExpression` calls.

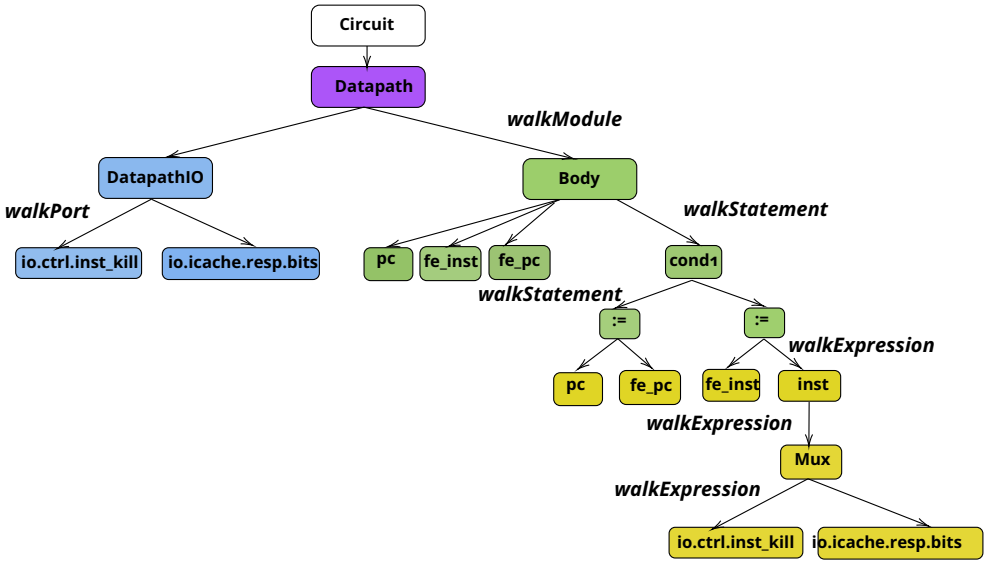

Figure 5.5 – `Process Module Function` for RISC-V Mini processor.

5.2.2 . Multi-module Datapath Pipeline Design

The architecture of the pipeline design is becoming increasingly complex. This complexity can be characterized in pipeline structure for example. As described above, the datapath pipeline design can be a mono-module or multi-module architecture. The multi-module datapath pipeline designates the datapath pipeline developed in several FIRRTL modules. Each module describes single or multiple pipeline stages. Then, the connection between modules is

obtained through the interface IO. Adding to the intra-module register analysis on each module, we extend the approach to the inter-module analysis.

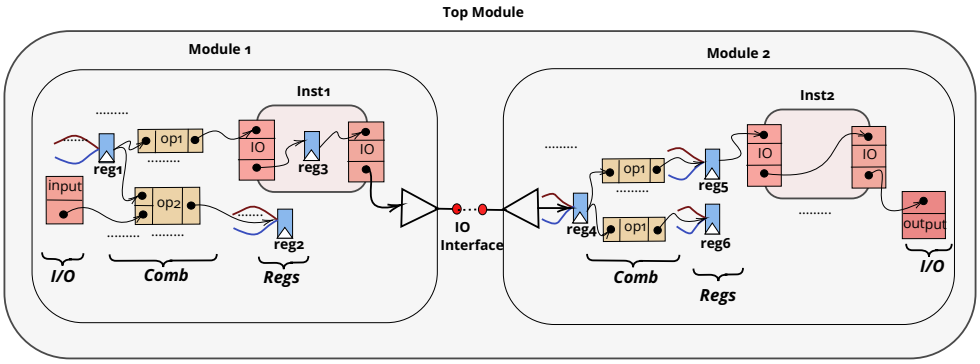

Figure 5.6 – Register analysis for multi-module datapath pipeline design

Figure 5.6 describes the main Chisel/FIRRTL constructions to illustrate multi-module register analysis. We define the top $\top$ module that contains all the instances of pipeline modules that can include pipeline registers as `Inst1` or be used as an I/O interface as in `Inst2`. The approach connects the registers of different modules through the input/output interface. It traverses recursively the combinatorial logics of the output context until the output frontier as a stop condition. Then, we can obtain the registers context of all the pipeline modules.

The RISC-V Chisel-based processors as `Rocket`, `Fuxi`, and RISC-V Sodor 3-stage are multi-module datapath pipeline designs. We illustrate next the notations refer to the multi-module pipeline through the Chisel code of RISC-V Sodor 3-stage processor.

Listing 5.2 – Snapshot of Chisel code of Sodor 3-stage processor.

```
1  class Core(width) extends Module {
2    val io = IO(new CoreIo())
3    val frontend = Module(new FrontEnd())
4    val dpath = Module(new DatPath())
5    frontend.io.cpu <> dpath.io.imem
6  }
7  class DpathIo(width) extends Bundle() {
8     val imem = Flipped(new FrontEndCpuIO())
9     val dmem = new MemPortIo(conf.xprlen)
10    val ctl  = Input(new CtrlSignals())
11    val dat  = new DatToCtlIo()
12 }
13 class DatPath(width) extends Module {
14    val io = IO(new DpathIo())
15    val alu = Module(new ALU())
```

```
16  /* Pipeline State Registers */
17    val wb_reg_ctrl    = Reg(new CtrlSignals)
18    val wb_reg_pc      = Reg(size)
19    val wb_reg_alu     = Reg(size)
20    val wb_reg_wbaddr = Reg(size)
21  /* Wires Declaration */
22     val exe_pc   = io.imem.resp.bits.pc
23     val exe_alu_out = alu.io.out
24     val exe_inst = io.imem.resp.bits.inst
25     val exe_wbaddr = exe_inst(RD_MSB, RD_LSB)
26  /* Registers Update */
27     wb_reg_pc := exe_pc
28     wb_reg_ctrl := io.ctl
29     wb_reg_alu := exe_alu_out
30     wb_reg_wbaddr := exe_wbaddr
31  }
32  class FrontEndIO(width) extends Bundle {
33    val cpu  = new FrontEndCpuIO
34    val imem = new MemPortIo(width)
35  }
36  class FrontEnd(width) extends Module {
37    val io = IO(new FrontEndIO)
38    /* Pipeline State Registers */
39    val if_reg_pc    = RegInit(size)
40    val exe_reg_pc   = Reg(size)
41    val exe_reg_inst = Reg(size)
42  /* Registers Update*/
43    when (io.cpu.resp.ready) {
44       exe_reg_pc  := if_reg_pc
45       exe_reg_inst := io.imem.resp.bits.data
46    }
47    io.cpu.resp.bits.inst := exe_reg_inst
48    io.cpu.resp.bits.pc  := exe_reg_pc
49  }
```

Listing 5.2 describes the multi-module datapath pipeline design of RISC-V Sodor 3-stage. The datapath pipeline includes three modules: `Core`, `FrontEnd`, and `DatPath`, lines 2,36,13. We focus next on the connection between modules in the multi-module datapath pipeline, all ensuring that the mono-module features are the same. The first two stages fetch, decode are implemented in `FrontEnd` module, while the execute stage is in `DatPath` and finally, `Core` is top $\top$ module of $\mathcal{P}_{Sodor3}$ designed to connect these two modules using their respective IO interface (frontend.io.cpu <> dpath.io.imem), in line 5. They are specified as `instance` in `core` module. Each module defines its ports as `Bundle` module, as `DpathIO` for `DatPath` module, `FrontEndIO` for `FrontEnd`

module, and `CoreIO` for `Core` module, lines 7,32,2 respectively.

Example 7 *The instance* `dpath` *of type module* `DatPath`*, and* `frontend` *for the module* `FrontEnd`*, lines 5-6, are denoted by* `(dpath, DatPath)` $\in Insts_{Core}$ *and* `(frontend, FrontEnd)` $\in Insts_{Core}$*, for the processor RISC-V Sodor-3stage.*

Example 8 *Whereas the modules* `FrontEnd`*,* `Core`*, and* `DatPath`*, form the pipeline* $\mathcal{P}$ *of RISC-V Sodor-3stages in Listing 5.2, other modules like* `ALU` *presented in* `DatPath` *module, line 16, are in* $Exts_{Core}$*.*

The connection between registers in a multi-module datapath pipeline design requires constructing the output context delimited by the output frontier. This context enables then the possibility to connect registers of different modules.

Example 9 *The output context of* `io.cpu.resp.bits.inst` *and* `io.cpu.resp.bits.pc` *in* `FrontEnd` *module, as the pairs* `<exe_reg_pc, io.cpu.resp.bits.pc>` *and* `<exe_reg_inst, io.cpu.resp.bits.inst>`*, in Listing 5.2, lines 47-48. They include the registers* `exe_reg_inst`*,* `exe_reg_pc` *as their output frontiers. Figure 5.7 illustrates the context of* `FrontEnd` *module outputs.*

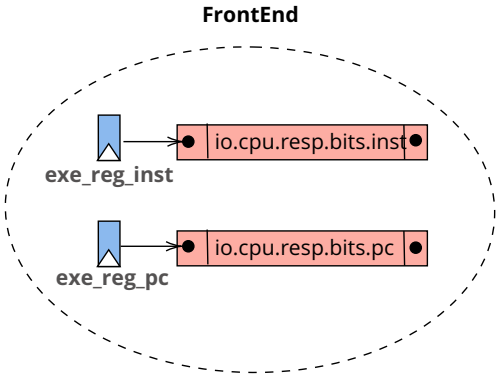

Figure 5.7 – Outputs context in RISC-V Sodor-3stages.

Through the output context `<exe_reg_pc, io.cpu.resp.bits.pc>`, the connection between the registers `exe_reg_pc` and `wb_reg_pc` declared in two different modules `FrontEnd` and `DatPath` respectively, lines 48, 27, 22 in Listing 5.2. The connections are established through the interface IO specified in top $\top$ module `Core`: `frontend.io.cpu <> dpath.io.imem`, line 5 in Listing 5.2. Figure 5.8 illustrates how to obtain the connection between registers of these two modules through the `Core` module IO interface.

Now, we illustrate how we construct the intermediate representation of RISC-V Sodor-3 stages pipeline design through the application of the Algorithm 2. For the three pipeline modules of RISC-V Sodor-3stages, `order_mods`

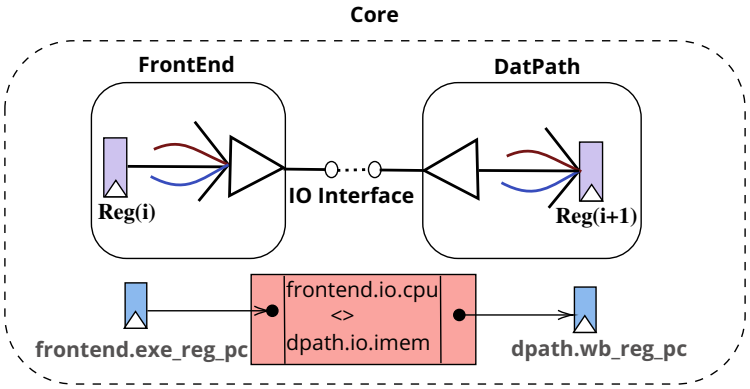

Figure 5.8 – IO interface for register precedence relation

produces the ordered set `Frontend`, `DatPath`, and then `Core`, where `Core` is the top $\top$ of $\mathcal{P}_{Sodor3}$. For each module, with the previous order, the $\mathcal{AST}_m$ is visited through $process\_mod$ function developed in Algorithm 1. The algorithm's illustration on a single module is detailed in the previous subsection 5.2.1. Then we construct the register context for the module `DatPath`, it includes the pairs <`io.imem.resp.bits.pc, wb_reg_pc`>, <`io.ctl, wb_reg_ctrl`>, <`io.imem.resp.bits.inst, wb_reg_wbaddr`> where their input frontiers are the module inputs. Then, the pair <`alu.io.out, wb_reg_alu`> with `alu.io.out` $\in Exts_M$, where the input frontier is an element of the external module `ALU`. The process is also applied for the module `FrontEnd` to construct the register context and the output context, which includes the pairs <`exe_reg_pc, io.cpu.resp.bits.pc`>, <`exe_reg_inst, io.cpu.resp.bits.inst`>. Finally, the process ends with the module `Core`, which includes the interface IO (`frontend.io.cpu <> dpath.io.imem`), line 5, in Listing 5.2. This interface implemented in this module allows the registers context delimited by the module inputs to extend to the other module registers, and then obtain the intermediate representation specified in `preced_regs` function. For example, we can obtain the precedence relation between two registers (`dpath.exe_reg_pc, frontend.wb_reg_pc`) through the register context <`frontend.io.imem.resp.bits.pc, frontend.wb_reg_pc`> and the output context <`dpath.exe_reg_pc, dpath.io.cpu.resp.bits.pc`> which can be reduced to the precedence relation using the interface (`frontend.io.cpu <> dpath.io.imem`). Then, the intermediate graph (as in Figure 5.9) is derived for all the datapath pipeline design.

5.3 . Experimental Results

We implemented our approach in Scala as a pass in the Chisel/FIRRTL hardware compiler framework. We evaluated its results on several open-source Chisel-based RISC-V processors described in detail in Section 2.2.2. Each processor design is considered *as is*, without manual modifications or

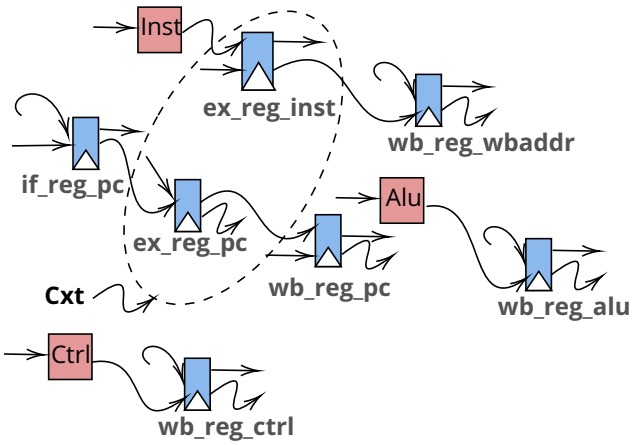

Figure 5.9 – Intermediate representation graph for RISC-V Sodor 3-stage

simplifications. Our evaluation is on a quad-core `Intel Core i7` at 1.90GHZ and 16GB RAM memory. We consider the following six RISC-V processor designs: RISC-V Mini [28], Sodor [1] with 3-stage and 5-stage, KyogenRV [29], Fuxi [30], and Rocket Chip [2]. These processors are developed in different Chisel versions, from '3.2' for Fuxi to '3.5' for Rocket. Our analysis is applied over single and multi-module datapath pipelines and these processors include both structures. We report the results of our register analysis in Table 5.1. We summarize the processor statistics related to the code and the pass results.

Table 5.1 – Register analysis experimental results on RISC-V processor designs.

	LOC	$\#Mp$	$\#Regs$	$\#Cxts$	$\#Frt$	$\#IRp$	*Runtime* (s)
Mini	241	1	11	3	8	18	0.12
Sodor-5	646	1	37	14	88	70	0.68
KyogenRV	4568	1	108	59	53	128	16.38
Sodor-3	575	3	26	8	17	24	0.52
Fuxi	2438	10	54	9	33	78	1752.51
Rocket	4159	5	240	68	224	133	49.63

The first three lines in Table 5.1 report the processors made of a single module implementation of the pipeline datapath, while the three last lines concern the multi-module pipelines.

The first column, *LOC* presents the code size of each pipeline $\mathcal{P}$, while the second column, $\#M_P$ indicates the number of pipeline modules single or multi-module, i.e., the cardinal number of $\mathcal{P}$. The following three columns report statistics related to the numbers of registers $\#Regs$, of contexts $\#Cxts$

as well as the largest input frontier $\#Frt$. This parameter measures the maximal connectivity between a register and other design elements, either sequential or combinational. Then, the next column $\#IR_P$ summarizes the intermediate representation results. It reports the resultant graph depth, i.e., the cardinal number of $Pred$. Finally, the last column reports the run time of the register analysis algorithm 2.

We can notice that the run-time of register analysis refers mainly on FIRRTL AST recursive traversal ($\#IR_P$ construction is not significant). This runtime value increases with the number of modules and depends on the datapath pipeline structure. For example, it is longest for the Fuxi processor, where the datapath pipeline is developed in 10 modules, and the pipeline structure is a linear traverse across the width of the pipeline. This leads to frequent module changes moving between pipeline stage modules and the MidStage module, as illustrated in Figure 4.5 in Section 4.3.3. Furthermore, the size of the intermediate representation graph is proportional to the number of registers for all the processor designs. The graph for Rocket processor is the largest, a 5-stage multi-module datapath pipeline design composed of 240 registers, and this graph is smallest for RISC-V Mini, a single 3-stage pipeline design.

5.4 . Conclusion

We have presented our approach that automatically generates the abstract datapath pipeline models, starting with the first phase: the register analysis. We have described the main steps to develop this register analysis by developing different algorithms working directly on the hardware designs at the FIRRTL level.

Our goal is to generate an intermediate representation of the datapath pipeline designs. In our case, it is represented as a non-strongly connected graph. The nodes are datapath pipeline registers, and the edges correspond to the dependency relation between registers and registers with the same context. We have applied this analysis to various RISC-V Chisel-based processors with different pipeline depths, and we summarized the processor illustrations and their experimental results.

The resultant intermediate representation will be the input feature for the following phase in our approach, where we assign pipeline registers to their respective pipeline stage. We present this phase next, in Chapter 6.

6 - Automatic Generation of Abstract Datapath Pipeline Models

The automatic generation of abstract pipeline models requires information about how the pipeline is designed, for example, if it is coded as a single or a multi-module pipeline (i.e., Chisel modules) and also if there is a hierarchy of modules. In Chapter 5, we have presented the first phase of our approach, which outputs an intermediate representation of pipeline datapath based on a register analysis. We intend to address in this chapter the second phase, which is built around two objectives: to determine the pipeline depth defined by pipeline stages and to identify how to connect these stages.

We detail in this chapter the automatic construction of the abstract datapath pipeline model, presenting the procedure to assign registers to pipeline stages in Section 6.1. Then, we exemplify our approach on several open-source RISC-V processors (in Section 4.3.3) of various degrees of complexity, i.e., from single to multi-modular pipeline designs, with the application being described in Section 6.2. In Section 6.3, we report on the experimental results.

6.1 . Construction of Abstract Datapath Pipeline Models

The automatic generation of abstract datapath pipeline models is based on two phases. The first phase is based on register analysis, which outputs an intermediate representation of the pipeline design described as a non-connected graph. The second phase relies on a register-to-pipeline stage assignment function to determine the datapath model. This phase provides an assignment procedure to map each register to its corresponding pipeline stage according to the results of the previously mentioned analyses. Next, we introduce some notations and algorithms of the assignment procedure.

Algorithm 3 summarizes the two phases of the construction algorithm of the datapath pipeline. The first phase corresponds to the register analysis presented in the previous Chapter and described in lines 1-6 in Algorithm 3. The second phase of the registers' assignment is described in the `assign_regs` function.

The abstract pipeline datapath of pipeline denoted by $\mathcal{P}$ is constructed by unfolding its intermediate representation $IR_{\mathcal{P}}$ and assigning nodes (i.e., registers) to pipeline stages. We denote by

$$\texttt{to\_stage} : Regs \rightarrow \mathbb{N}^{+}$$

an operator to assign to the registers their stages.

Algorithm 3: Pipeline datapath construction

Input : $\mathcal{P}$ and $AST_{\mathcal{P}} = \{AST_1, ..., AST_n\}$
Output: $Regs \mapsto Stages$ - pipeline datapath of $\mathcal{P}$

1 $o \leftarrow$ `order_mods`$(\mathcal{P},\ AST_p)$ `/* order modules based on instancing, ⊤ is last. */`
2 **foreach** $m \in o$ **do**
3 $\quad (I/Os_m,\ Regs_m,\ Ctxs_m) \leftarrow$ `process_mod`(AST_m)
4 $\quad C_r \leftarrow C_r \cup$ `regs_ctx`$(AST_m,\ Regs_m,\ I/Os_m)$
5 $\quad C_{io} \leftarrow C_{io} \cup$ `ios_ctx`$(AST_m,\ Regs_m,\ I/Os_m)$
6 $IR_{\mathcal{P}} \leftarrow$ `preced_regs`(C_r, C_{io}) `/* build intermediate representation of` $\mathcal{P}$ `*/`
7 `assign_regs`$(1,\ PC,\ IR_{\mathcal{P}}, Regs)$ `/* assign registers to stages, PC is in stage 1 */`

Algorithm 4 presents the assignment of registers to pipeline stages. The intermediate representation $IR_{\mathcal{P}}$ is a non-strongly connected graph that can present a cycle and nodes that correspond to the registers that can not all be connected. Thus, this algorithm starts by assuming that the program counter $PC \in Regs$, assigned in the first pipeline stage, i.e., `to_stage`$(\texttt{PC}) = 1$, and performs two assignment strategies.

We distinguish two cases: case 1 driven by register dependencies and case 2 driven by a heuristic defined using the context dependencies, $Ctxs$. We further define three possibilities for case 1: the `linear_case` 1, the `min_case` 1 and the `max_case` 1. When the selected register r cannot be assigned by the above conditions, the operator `select_reg` selects another assigned register r' to stage i' from which we proceed, in line 16.

Linear_case 1 (lines 2-4 of Algorithm 4) is applied when the register has only one source. More specifically, a register r is assigned to the pipeline stage $i + 1$ if it is the only destination register of reg, which is already assigned to stage i. Formally, it is applied when the condition $C1\_linear$ is true, where

$$C1\_linear = \exists!(reg, r) \in Pred$$

Min_case 1 (lines 5-7 of Algorithm 4) considers the case when several source registers are already assigned with no precedence relation between the source register of the minimal stage and the other source registers. The stage assigned to r strictly follows this minimal stage of its source registers, which are already assigned to later pipeline stages. Thus, they are connected to r through backward edges (i.e., the forwarding mechanism). This case is applied to some configurations of the forwarding mechanism.

Formally, the `min_case` 1 is applied to assign r, preceded by $regs$, when the condition $C1\_min$ is true, where

$$C1\_min \;=\; \exists (reg, r) \in Pred \;\wedge\; \texttt{to\_stage(reg)} = i \;\wedge$$
$$\forall (r_2, r) \in Pred,\; r_2 \neq reg \;\wedge$$
$$\texttt{to\_stage}(\texttt{r}_2) >= \texttt{to\_stage(reg)} \;\wedge\; \nexists (reg, r_2) \in Pred$$

Figure 6.1 illustrates this case in the processor RISC-V Sodor 5-stage. The source registers `if_pc`, `ex_reg1` and `ex_reg2` of the register `dec_reg` are already assigned, and there is no relation between the source register of minimal stage `if_pc` with the other sources. So, we assign to `dec_reg` the minimal stage of its sources plus 1.

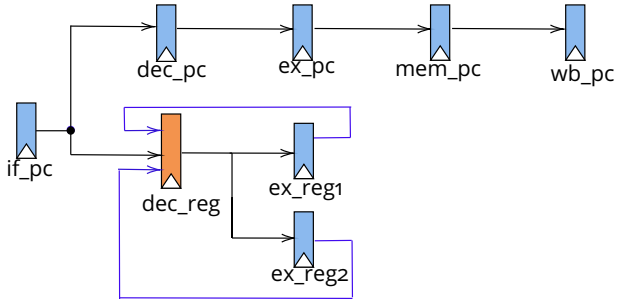

Figure 6.1 – Simple application of min case 1 ($C1\_min$).

Algorithm 4: Register to pipeline stage assignment

```
1  Function assign_regs(i, reg, IR_P, Regs):
       /* C1_* and C2 identify r in IR_P */
2      if C1_linear then
3          to_stage(r) = i + 1 ;                    /* linear_case 1 */
4          assign_regs(i + 1, r, IR_P, Regs)
5      else if C1_min then
6          to_stage(r) = i + 1 ;                       /* min_case 1 */
7          assign_regs(i + 1, r, IR_P, Regs)
8      else if C1_max then
9          to_stage(r) = i + 2 ;                       /* max_case 1 */
10         assign_regs(i + 2, r, IR_P, Regs)
11     else if C2 then
12         to_stage(r) = i ;                              /* case 2 */
13         assign_regs(i, r, IR_P, Regs)
14     else
15         (i', r') ← select_reg(IR_P, Regs)
16         assign_regs(i', r', IR_P, Regs)
17     return to_stage;
```

Max_case 1 (lines 8-10 of Algorithm 4) also addresses the case of several source registers that are already assigned. It checks if the source register of the minimal stage is directly connected to all the other sources. In that case, the stage assigned to r follows the maximal stage of its source registers.

Formally, the `max_case` 1 assigns r (preceded by reg) when $C1\_max$ is true, where

$$\begin{aligned} C1\_max = \; & \exists (r_2, r) \in Pred \; \wedge \; \texttt{to\_stage}(\texttt{r}_2) = i + 1 \; \wedge \\ & \forall (reg, r) \in Pred \; \wedge \; r_2 \neq reg \; \wedge \\ & \texttt{to\_stage}(\texttt{r}_2) >= \texttt{to\_stage}(\texttt{reg}) \; \wedge \exists (reg, r_2) \in Pred \end{aligned}$$

Figure 6.2 illustrates this case, which is found in the Rocket processor to consider compressed instructions in the pipeline design. We explain this implementation later. The two source registers `buf.pc` and `s2_pc` of register `ex_reg_pc` are already assigned, and moreover, `s2_pc` precedes `buf.pc`. So, we assign to register `ex_reg_pc` the maximal stage (i.e., that of `buf_pc` plus 1).

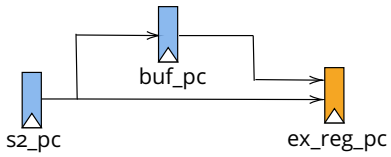

Figure 6.2 – Simple application of max case 1 ($C1\_max$).

Rocket is able to handle compressed instructions. Thus, the pipeline design has an additional stage to accommodate them, becoming a 6-stage pipeline instead of a 5-stage one. Listing 6.1 presents a code snippet to illustrate this situation. If the `useCompressed` condition is true, in module `Rocket`, the value of `nBufValid`, in module `IBuf`, is greater than 0, and the register `buf.pc` is connected to the output `io.pc`. As such, an additional pipeline stage is added, with the register `buf.pc` preceding `ex_reg_pc`. Otherwise, the register `ex_reg_pc` has at its input frontier `io.imem.bits.pc`, which is further connected to the register `s2_pc`.

Listing 6.1 – Simplified code for compressed instructions, in Rocket

```
1 class Rocket :
2   val ibuf = Module (new IBuf)
3   useCompressed: Boolean = true
4   val fetchWidth: Int =
5     if (useCompressed) then 2 else 1
6   ex_reg_pc := ibuf.io.pc
7 -----------------
8 class IBuf
9   val n = fetchWidth - 1
```

```
10    val nBufValid =
11      if (n == 0) then UInt(0) else Reg(fetchWidth)
12    io.pc :=
13      Mux(nBufValid > 0, buf.pc, io.imem.bits.pc)
```

Our approach is developed at FIRRTL high form level. The high form compilation passes rely on the check and simplification transformations and do not modify the circuit as a propagation constant pass. Thus, the analysis is conservative as it considers both outcomes of the multiplexer condition, i.e., for compressed and uncompressed instructions. Therefore, it constructs a 6-stage pipeline datapath model for Rocket, applying the `max_case` 1. A complete construction of this model is presented in Section 6.2.

This last possibility of case 1 presents an inaccurate result if the register has two sources connected and one of them is assigned in the same stage as the particular register. According to this, it will be placed incorrectly with the `max_case` 1. This case refers to the backward edges presented in the forwarding mechanism when we can find connections between registers of the same and latter stages. In this case, the register source with the minimally assigned stage is not connected to all the sources of the register under assignment and the `min_case` 1 will be applied. We illustrate this case in Figure 6.3.

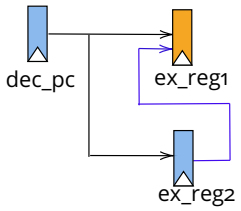

Figure 6.3 – Inaccurate application of `min/max_case` 1.

The source registers `dec_pc` and `ex_reg2` of `ex_reg1` are already assigned, and the source register `dec_pc` precedes this source `ex_reg2`. So, the `max_case` 1 is applied, assigning the register `ex_reg1` to the maximal stage plus 1. However, this is the result of an over-approximation. The register `ex_reg1` should be located in the same stage as `ex_reg2` since it is a part of the forwarding mechanism that must be completed with other sources in the latter stages, where the source with the minimal stage is not connected to all these other sources. Thus, the `min_case` 1 will be applied, and we assign the register `ex_reg1` to the minimal stage plus 1.

Case 2 (lines 11-13) assigns a register r in the same pipeline stage as an already assigned register reg from the same context. Formally, the `case` 2 is

applied when $C2$ is true, where

$$C2 = \texttt{ctx\_reg}(\texttt{r}) = R \wedge \exists reg \in R \wedge \texttt{to\_stage}(\texttt{reg}) = i.$$

Figure 6.4 illustrates in more detail an example of this case in the Rocket processor, namely the assignment of register `ex_reg_inst`, which is connected to an external module `inst` (i.e., that is not included in the considered datapath pipeline modules). `ex_reg_inst` is assigned according to this `case` 2 since it is collected in the same context as an assigned register `ex_reg_pc`, thus in the same stage as this register.

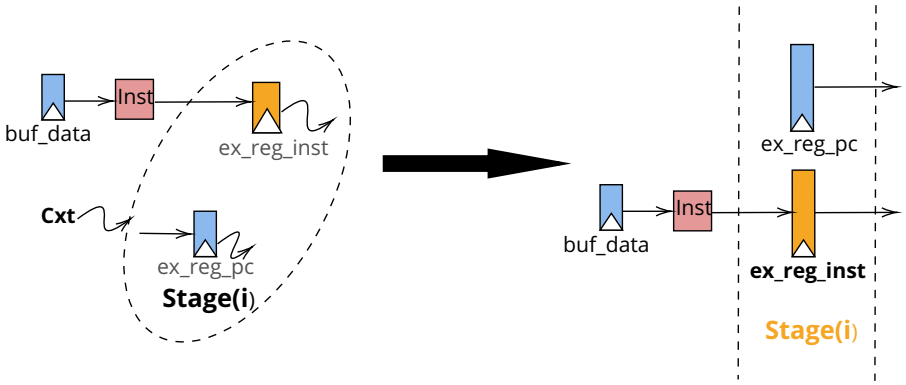

Figure 6.4 – Simple application of case 2.

The solution produced by our algorithm is the graph $IR_{\mathcal{P}}$ with registers as nodes, for which pipeline stages have been assigned. Our algorithm is based on dependency relations between registers as well as an heuristic that considers the way the design is made. However, some registers may not be assigned to a pipeline stage due to the heuristic nature of our assignment algorithm that combines the various assignment cases defined before.

Therefore, the correctness of the algorithm is reduced to prove a subsumption relation relating the original processor design, say $Q_{\mathcal{P}r}$ and the solution $P_{IR_{\mathcal{P}}}$ of our algorithm. Precisely, $Q_{\mathcal{P}r}$ is the set of `prec` predicates, derived and evaluated with respect to a set of processor design executions and $P_{IR_{\mathcal{P}}}$ is the logical encoding (i.e., also as a set of `prec` predicates) of the transitions in the subgraph solution of our algorithm.

6.2 . Application to RISC-V Processor Designs

In the last section, we have presented the register assignment procedure described in the `assign_regs` function of Algorithm 3. This function, as mentioned before, maps registers in their pipeline stages. In this section, we illustrate the application of this procedure on different RISC-V Chisel-based processors with mono-module and multi-module datapath pipeline designs.

Mono-module Pipeline Design - The RISC-V Sodor 5-stage. We illustrate the construction of the pipeline datapath model for the RISC-V Sodor 5-stage processor. Our analysis generates an $IR_{\mathcal{P}}$ graph as the one presented in Figure 6.5 and proceeds to register placement.

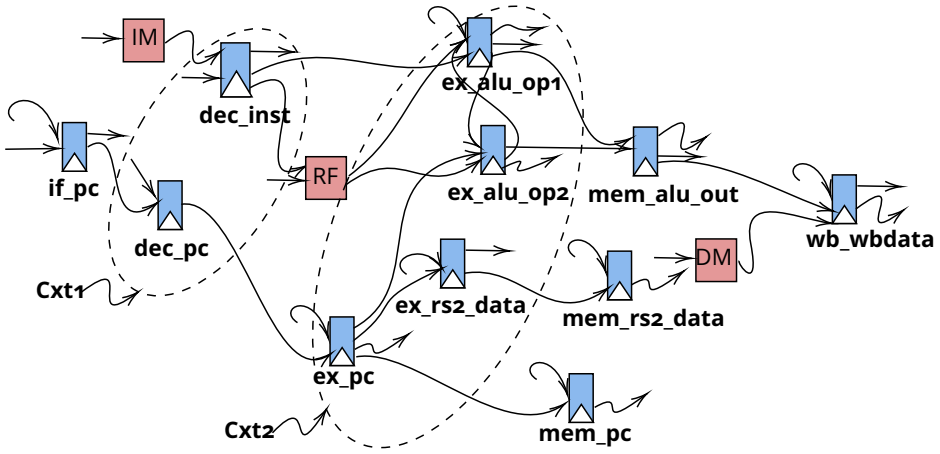

Figure 6.5 – Intermediate representation graph for RISC-V Sodor 5-stage

The processor is made of 37 registers and we present, in Figure 6.6, some of these registers that we consider relevant. We specify that the `if_pc` register (i.e., the PC register) is located in the first pipeline stage. Our algorithm places the registers `dec_pc`, `ex_pc` and `mem_pc` into respectively consecutive stages as they are part of the same connected component of $IR_{\mathcal{P}}$ with one register source (*linear_case* 1, shown in Figure 6.8). Then, our algorithm assigns the register `dec_inst` as it is updated in the same context as `dec_pc` (using *case* 2). Indeed the frontier of the register context of `dec_inst` contains a reference to an external design element, the module IM (standing for the instruction memory).

The next registers to be assigned are the forwarding registers `ex_alu_op1`, `ex_alu_op2` and `ex_rs2_data`. They should be assigned with the *min_case* 1 since all these registers are connected to several source registers where there are no precedence relations between the source register of minimal stage `dec_inst`, and all other source registers (like the `mem_alu_out`, and `wb_wbdata` registers). However, their respective input frontiers contain registers that are not currently assigned (i.e., `mem_alu_out` and `wb_wbdata`), since they are part of the backward edges in blue color in Figure 6.6, coming from the data forwarding semantics. However, the context of register `ex_pc` contains these not assigned registers enabling the use of the *case* 2 of our algorithm. Finally, our algorithm places the remaining registers, i.e., `mem_rs2_data` with the *linear_case* 1 based on the register dependencies, and `mem_alu_out` with

min_case 1. The source registers of `mem_alu_out`, which are `ex_alu_op1`, `ex_alu_op2`, and `ex_pc` are already assigned in the same stage, and there are no precedence relations between all of them. Thus, `mem_alu_out` is assigned with *min_case* 1. Therefore, the algorithm assigns the register `wb_wbdata` with the *linear_case* 1 based on the register dependencies. We summarize in Section 6.3, the experimental result of our algorithm application on the complete design of RISC-V Sodor 5-stage.

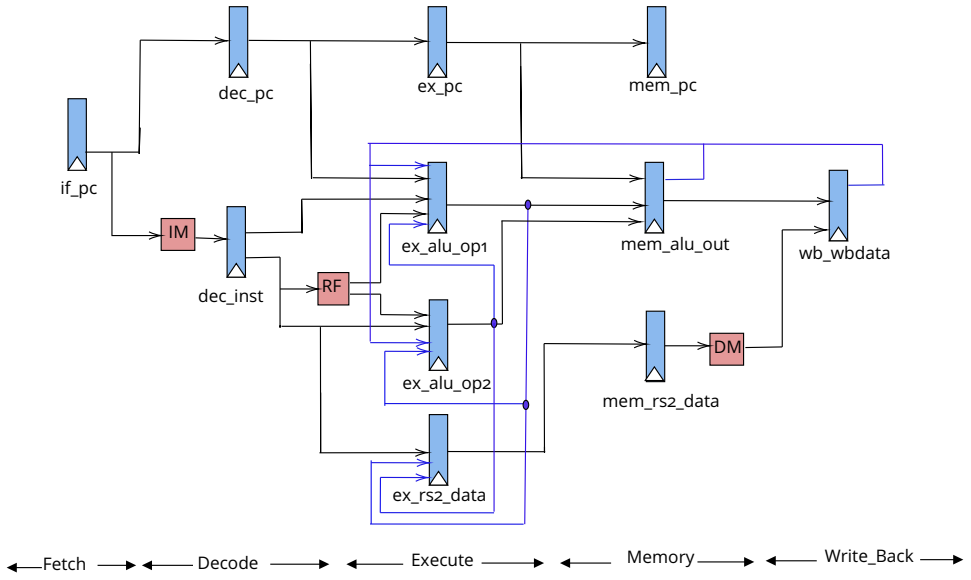

Figure 6.6 – RISC-V Sodor 5-stage pipeline datapath model.

Mono-module Pipeline Design - Kyogen RISC-V. We now illustrate how our pass analyzes a RISC-V processor, called KyogenRV [29] in order to construct its abstract pipeline model. KyogenRV is an open-source 5-stage pipeline processor. More details of this processor are presented in Section 2.2. By our classification, it is a mono-module pipeline design. Our pass focuses on the top module of the pipeline, namely `KyogenRVCpu`, which has 108 registers, out of which we present the application of our algorithm only on a subset of registers, i.e., those linked to the PC register in Table 6.1 and Figure 6.7.

Table 6.1 describes the register assignment phase on the KyogenRV processor. We assume that `if_pc` is the `PC` register and is placed in the first pipeline stage. Then our algorithm assigns the register `id_pc_temp` using the *linear_case* 1, since it follows directly the source register `if_pc`. Our algorithm proceeds to assign the register `if_npc` with the *case* 2 since it is updated in the same context with the `PC` register. This enables the assignment of the register `id_npc_temp` with the *linear_case* 1. Then, our algorithm assigns the register `id_inst` linked to an external module `instruction memory` using the *case* 2 as it is updated in the same context as the register `id_pc_temp`, fol-

lowed with the assignment of the registers `ex_inst` and `id_inst2_temp` with *linear_case* 1. Furthermore, the registers `id_pc` and `id_npc` are also assigned using *case* 2, since they are connected to registers that are not already assigned. Then, the registers `ex_pc, mem_pc, ex_npc, mem_npc, wb_pc` are assigned into their respective pipeline stages with *linear_case* 1, as they are part of the same connected component of the resulting intermediate graph. The next registers to be assigned are `ex_reg_raddr` and `ex_reg_waddr`. Their respective input frontier includes the external module `RegisterFile`, so they are assigned with the *case* 2 relying on the same context with `ex_pc`. Then, the following registers `mem_reg_waddr` and `wb_reg_waddr` are assigned with the *linear_case* 1.

The forwarding mechanism of the KyogenRV processor is implicitly implemented with wires that go through the external `ALU` module. So, the registers `ex_rs,mem_rs, mem_alu_out` are updated with the output of the module `ALU`, being assigned with *case* 2.

Listing 6.2 presents a subset of the data forwarding implemented from the WB and MEM towards the EX stage. The wire `ex_reg_rs1_bypass` is updated from the output of the `mem_alu_out` register (line 5), located in the MEM stage, or from the `wb_alu_out` register (line 6), located in the WB stage, through the Chisel `MuxCase` construction. In the FIRRTL low form, a `MuxCase` is translated into a cascade of multiplexers. This forwarding corresponds to the blues arrows, shown in Figure 6.7, from the registers `wb_alu_out` and `mem_alu_out` to the input of the ALU red box. The other blue arrow, between the register `wb_reg_waddr` and the input of the ALU, is part of the check to detect the need for forwarding a value, as in line 6.

Listing 6.2 – Forwarding in KyogenRV 5-stage.

```
1  val ex_reg_rs1_bypass = Wire(UInt(32.W))

3  /* Ci, i = 1..2 - conjuncts of write enable and selection
       ↪ signals */
4  ex_reg_rs1_bypass := MuxCase(ex_rs(0), Seq(
5    (ex_reg_raddr(0) === mem_reg_waddr && C1) → mem_alu_out
6    (ex_reg_raddr(0) === wb_reg_waddr && C2) → wb_alu_out))
7  ...
8  when (C4 /* no stalling condition */) {
9    mem_rs(0) := ex_reg_rs1_bypass
10 }
```

Finally, our algorithm assigns the register `pc_cntr` with the *min_case* 2, where its source registers `mem_pc, mem_alu_out` are assigned in the same pipeline stage.

Table 6.1 – Experimental results on KyogenRV processor.

Reg to assign	Module	Source register	#St
if_pc	KyogenRVCpu	-	1
id_pc_temp	KyogenRVCpu	if_pc	2
if_npc	KyogenRVCpu	-	1
id_npc_temp	KyogenRVCpu	if_npc	2
id_inst	KyogenRVCpu	-	2
ex_inst	KyogenRVCpu	id_inst	3
id_inst2_temp	KyogenRVCpu	id_inst	3
id_pc	KyogenRVCpu	-	2
ex_pc	KyogenRVCpu	id_pc	3
mem_pc	KyogenRVCpu	ex_pc	4
id_npc	KyogenRVCpu	-	2
ex_npc	KyogenRVCpu	id_npc	3
mem_npc	KyogenRVCpu	ex_npc	4
wb_npc	KyogenRVCpu	mem_npc	5
ex_reg_raddr[0]	KyogenRVCpu	-	3
ex_reg_raddr[1]	KyogenRVCpu	-	3
ex_reg_waddr	KyogenRVCpu	-	3
mem_reg_waddr	KyogenRVCpu	ex_reg_waddr	4
wb_reg_waddr	KyogenRVCpu	mem_reg_waddr	5
ex_rs[0]	KyogenRVCpu	-	3
ex_rs[1]	KyogenRVCpu	-	3
mem_rs[0]	KyogenRVCpu	-	4
mem_rs[1]	KyogenRVCpu	-	4
mem_alu_out	KyogenRVCpu	-	4
wb_alu_out	KyogenRVCpu	mem_alu_out	5
pc_cntr	KyogenRVCpu	mem.pc $\rightarrow$ 4, mem_alu_out $\rightarrow$ 4	5
w_addr	KyogenRVCpu	-	5
w_data	KyogenRVCpu	-	5

We present the abstract pipeline model of KyogenRV in Figure 6.7. It shows a subset of the identified registers and their dependencies (omitting the combinatorial circuitry). Our pass correctly identifies the 5 stages, with forwarding mechanisms from the WB and the MEM to the EX stage implemented with the blue edges (potentially merged). We details the statistics result in Section 6.3.

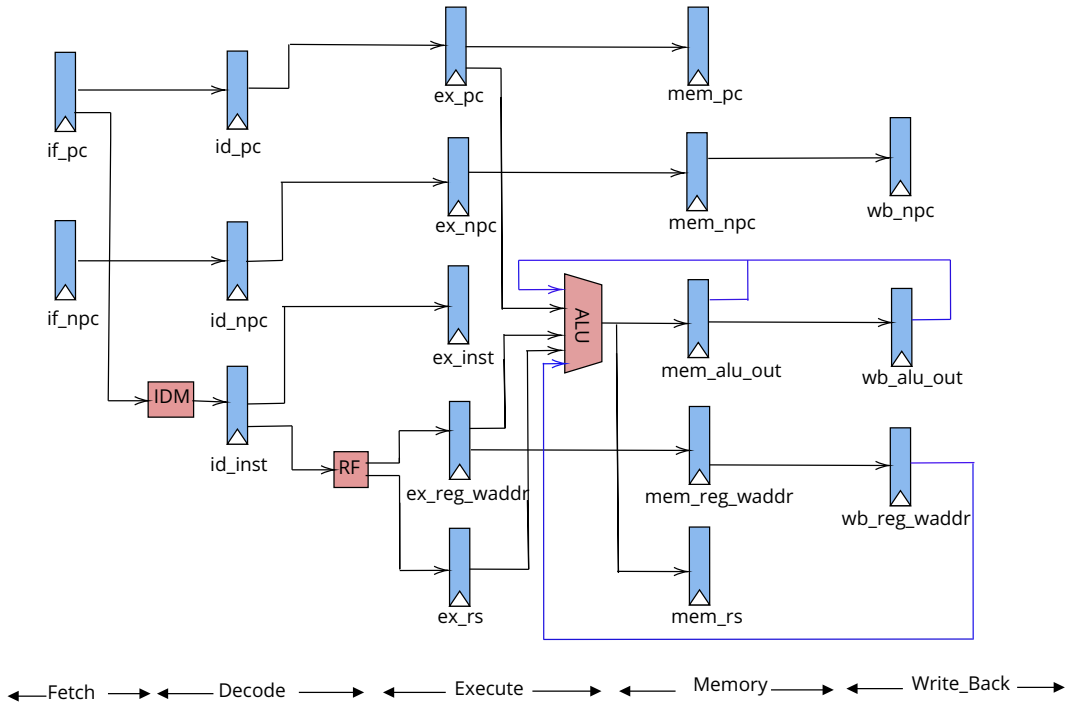

Figure 6.7 – Partial representation of the abstract pipeline model of the KyogenRV processor.

Multi-module Pipeline Design - Rocket Processor We detail now the abstract datapath pipeline construction of a multi-module pipeline design, that of the Rocket processor. This particular pipeline is specified in five Chisel modules `Rocket, IBuf, FrontEnd, RocketTile, ShiftQueue` when the first three modules contain the pipeline registers, and the last two are developed as interface modules.

Table 6.2 details the execution of the register assignment algorithm (i.e., Algorithm 4) on the Rocket processor. Since this design is made of 240 registers, we choose to present only a subset of these registers. We start, as usual, with `s1_pc`, the PC register, assumed to be assigned in the first pipeline stage. Then, our algorithm proceeds to place registers `s2_pc`, `buf.pc` and `buf.data` into their respective stages using the *linear_case* [1] as these registers are part of the same connected component of $IR_{\mathcal{P}}$. The most interesting case is that of register `ex_reg_pc`. Our algorithm proceeds to assign this register through the `max_case` [1] since its source registers `s2_pc` and `buf.pc` are already assigned to stages 2 and 3 respectively, and the source min `s2_pc` precedes the other source `buf.pc`. Then, the *linear case* [1] is applied in order to assign the registers `mem_reg_pc` and `wb_reg_pc` to their corresponding stages. The next registers to be assigned are `ex_reg_cause` and `ex_reg_inst` as their respective input frontier contains a reference to an external design element, the module `Inst`. So, these registers are assigned using case [2] as

they are updated in the same context as `ex_reg_pc`. Then, our algorithm applies the *linear_case* 1 to assign pipeline stages to registers `mem_reg_cause` and `mem_reg_inst`, and further to registers `wb_reg_cause` and `wb_reg_inst`.

Furthermore, case 2 is applied to register `mem_reg_wbdata`, which is connected to an external design element, the module `ALU` and which is updated in the same context as `mem_reg_pc`. A similar case is the register `wb_reg_wbdata`, updated in the same context as `wb_reg_pc`, thus, case 2 is applied. Finally, our algorithm assigns the remaining registers `ex_reg_rs0`, `ex_reg_rs1` to pipeline stages. More precisely, their respective input frontier contains a reference to an external design element, the module corresponding to the register file `RF`. So these registers are assigned using the case 2, as they are updated in the same context as the registers already assigned to the execute stage.

Figure 6.8 shows the resulting pipeline datapath of Rocket, omitting those registers that are not on the path from the initial PC register.

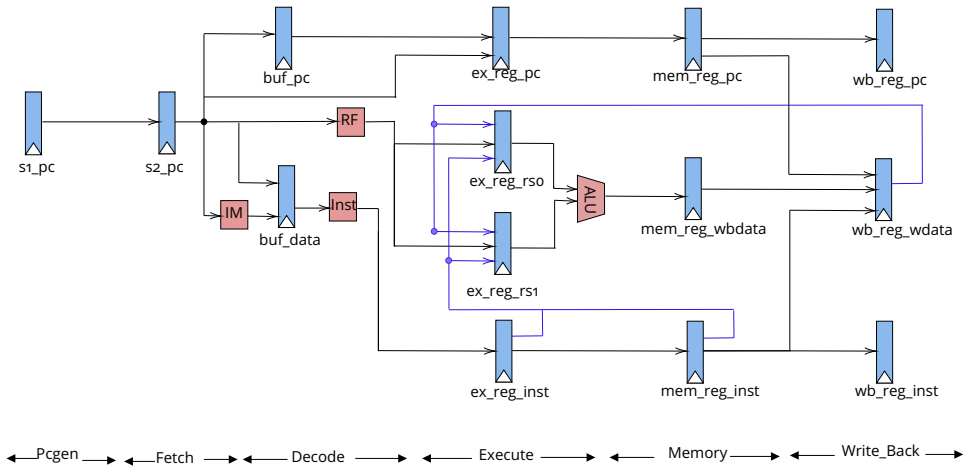

Figure 6.8 – Partial representation of the abstract pipeline model of the Rocket processor.

6.3 . Experimental Results and Synopsis

We evaluated our analysis on several Chisel-based RISC-V processors. We consider our use case processors presented in Chapter 2, the mono-module designs such as RISC-V Mini [28], Sodor 5-stage [1], and KyogenRV [29], and the multi-module structure as Sodor 3-stage [1], Fuxi [30], and Rocket Chip [2]. Our evaluation is on a quad-core `Intel Core i7` at 1.90GHZ and 16GB RAM memory. We report the results of our analysis in Table 6.3.

Table 6.2 – Experimental results on Rocket chip processor.

Register to assign	Module	Source register	#St
s1_pc	Frontend	-	1
s2_pc	Frontend	s1_pc	2
buf.pc	IBuf	s2_pc	3
buf.data	IBuf	s2_pc	3
ex_reg_pc	Rocket	buf.pc → 2, s2_pc → 1	4
mem_reg_pc	Rocket	ex_reg_pc	5
wb_reg_pc	Rocket	mem_reg_pc	6
ex_reg_cause	Rocket	-	4
mem_reg_cause	Rocket	ex_reg_cause	5
ex_reg_inst	Rocket	-	4
mem_reg_inst	Rocket	ex_reg_inst	5
wb_reg_inst	Rocket	mem_reg_inst	6
ex_reg_raw_inst	Rocket	-	4
mem_reg_raw_inst	Rocket	ex_reg_raw_inst	5
wb_reg_raw_inst	Rocket	mem_reg_raw_inst	6
ex_reg_wphit[0]	Rocket	-	4
mem_reg_mem_size	Rocket	-	5
wb_reg_mem_size	Rocket	mem_reg_mem_size	6
ex_reg_mem_size	Rocket	mem_reg_mem_size	4
mem_reg_wdata	Rocket	-	5
wb_reg_cause	Rocket	-	6
wb_reg_wdata	Rocket	-	6
wb_reg_wphit[0]	Rocket	-	6
mem_reg_wphit[0]	Rocket	wb_reg_wphit[0]	5
ex_reg_rs_bypass[0]	Rocket	-	4
ex_reg_rs_bypass[1]	Rocket	-	4
ex_reg_rs_lsb[0]	Rocket	-	4
ex_reg_rs_lsb[1]	Rocket	-	4
ex_reg_rs_msb[0]	Rocket	-	4
ex_reg_rs_msb[1]	Rocket	-	4

Table 6.3 – Experimental results on RISC-V processor designs.

	LOC	$\#Mp$	$\#Regs$	$\#Cxts$	$\#Frt$	linear_case	min_case	max_case	*Case 1*	*Case 2*	*runtime(s)* -intra-	*runtime(s)* -assignment-
RISC-V Mini	241	1	11	3	8	3	-	-	4	7	0.12	0.049
Sodor 5-stage	646	1	37	14	88	24	2	-	27	10	0.68	0.17
KyogenRV	4568	1	108	59	53	41	2	-	44	30	16.38	0.047
Sodor 3-stage	575	3	26	8	17	6	-	-	7	1	0.52	0.062
Fuxi	2438	10	54	9	33	29	4	-	34	19	1752.51	0.054
Rocket	4159	5	240	68	224	13	-	1	15	15	49.63	0.087

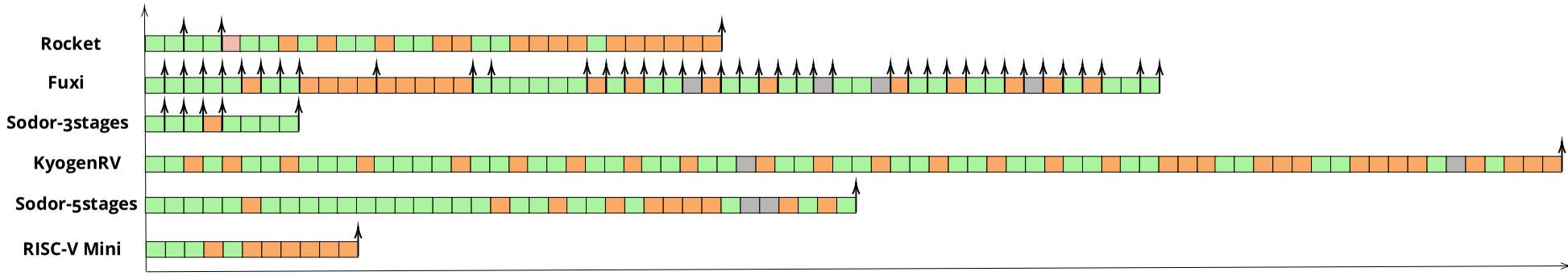

Figure 6.9 – Execution of the register assignment algorithm, i.e., Algorithm 4.

The first column, *LOC* presents the code size of each pipeline $\mathcal{P}$, while the second column, $\#M_P$ indicates the number of pipeline modules, i.e., the cardinal number of $\mathcal{P}$. Our analysis is applied over single and multi-module datapath pipelines. The first three lines in Table 6.3 report the processors made of a single module to implement its pipeline datapath, while the three last ones concern the multi-module pipelines. The next three columns report statistics related to the numbers of registers $\#Regs$, of contexts $\#Cxts$ as well as the largest input frontier $\#Frt$. This parameter is a measure of the maximal connectivity between a register and other design elements, either sequential or combinational. The next five columns detail the number of registers successfully assigned to pipeline stages. As such, columns *linear_case*, *min_case* and *max_case* report the number of placed registers by the respective case. Then, column *Case 1* reports the aggregated results of these three cases including the `PC` register, and column *Case 2* presents the number of registers placed by this particular case. Finally, the last two columns report the run time of our algorithm: the first corresponds to the intra-module phase (e.g., lines 2-5 of Algorithm 3), while the second corresponds to the register assignment phase (e.g., Algorithm 4).

The depth of each pipeline has been correctly computed, namely our approach constructs a pipeline datapath of the same depth as in the respective processor specification. Moreover, using the naming conventions with respect to register names and the pipeline stages (e.g., prefixes `dec_` and `ex_ or exe_` for decode and execute stage respectively, as shown in Listing 4.13 or Listing 4.16), we could also check that the registers are assigned in their expected pipeline stage.

We also encountered several issues. For example, our approach initially identified the KyogenRV design as a 7-stage pipeline instead of a 5-stage one. This over-approximation was due to the use of the Chisel `RegNext` construct. More specifically, the semantics of `RegNext` produces a one-cycle delayed version of the associated signal as described in Listing 4.8 in Section 4.3. At the FIRRTL level, it is translated into an additional register for each delay and thus an additional pipeline stage as there are two uses of `RegNext` in KyogenRV, as detailed in Listing 4.9 in Section 4.3. However, these registers can be identified based on their compiled FIRRTL names (i.e., a particular prefix is added), allowing us to discard them when building the $IR_{\mathcal{P}}$ and then the assignment of registers.

Furthermore, our approach assigns all the registers of the pipeline modules only for two designs, Sodor 5-stage, and RISC-V Mini. For the KyogenRV processor, the unassigned registers are, in fact, not related to the datapath but to the control path as our algorithm also collected a vector register named `rv32i_reg[0-31]`, implementing the register file. We present next an example of a control register called `ex_ctrl.alu_op1` in Listing 6.3. This register is an element of the register `ex_ctrl`, which is a `Bundle` type `IntCtrlSigs`, lines

11-13 in Listing 6.3. Thus, all the fields of this class `IntCtrlSigs`, lines 2-6 in Listing 6.3, such as `ex_ctrl.br_type`, `ex_ctrl.legal`, `ex_ctrl.alu_op1` and `ex_ctrl.alu_op2` are registers defined in the control path and used in the datapath module.

Listing 6.3 – Control registers in KyogenRV processor.

```
1    /* Defined in Conrol Module*/
2  class IntCtrlSigs extends Bundle {
3     val legal:    Bool = Bool()
4     val br_type:  UInt = Bits(size)
5     val alu_op1:  UInt = Bits(size)
6     val alu_op2:  UInt = Bits(size)
7     ......
8  }
9    /* Called in KyogenRVCpu Module */
10 class KyogenRVCpu extends Module {
11   val ex_ctrl: IntCtrlSigs = RegInit()
12   ex_op1 := MuxCase (0.U(32.W),
13        (ex_ctrl.alu_op1 == OP1_RS1) → ex_reg_rs1_bypass,
14        ......
15   )
16 }
```

For Rocket and Sodor 3-stage designs, the control registers are generally defined as a compound type such as `Bundle` or `Vector` type. Thus, each field is considered an individual register and represented as such when the register assignment is performed. However, these registers are from the control path, updated in the external modules $Exts$, for instance, the modules `CSR` or `BTB` of Rocket. Indeed, Rocket is more complex, and modules have emerged that we have not yet considered. With these modules, we need to be able to link registers and, therefore, assign many registers. Sodor 3-stage and 5-stage are two variants of RISC-V Sodor, of which our algorithm assigns all the pipeline registers only for the 5-stage. This difference between the two variants of the same processor is due to their pipeline datapath structure. The Sodor 5-stage is a mono-module pipeline design. However, the Sodor 3-stage pipeline datapath is built on several pipeline modules.

We can also notice that the runtime of the assignment phase is significantly lower than the one of the intra-module phase. The latter increases with the number of modules, and thus, is the highest for the Fuxi processor. This runtime value for the intra-module phase summarizes all the iterations of the processor pipeline modules.

Next, we summarize in Figure 6.9 the execution steps of the register assignment, i.e., Algorithm 4, for all the processor designs. We illustrate the evolution of this assignment function when successfully assigning registers.

The coding color of the algorithm traces is also displayed in Table 6.3: green represents the *linear_case* [1], gray and pink are *min_case* [1] and *max_case* [1] respectively, and orange represents *case* [2]. Furthermore, the multi-module aspect is also reported by specifying the transition between pipeline modules when assigning registers, using the symbol '↑'. We notice that the register assignment applies the *linear_case* [1] in the beginning as it is driven by the register dependencies. We also notice that the transition between modules exposes the way the implementation of the multi-module aspect is done in our analysis. For Fuxi, the flat module design results in frequent module changes, moving between pipeline stage modules and the `MidStage` module, as shown in Fig. 4.5. It is, however, not the case of Rocket and Sodor 3-stage. This is due to the hierarchical module design, shown for Rocket in Fig. 4.4, where the registers are restricted to three modules `IBuf`, `Frontend` and `Rocket`. More precisely, the registers of the stages PC-generator and IF, are implemented in the module `Frontend` while the registers of stages from ID to WB are in the module `Rocket`.

6.4 . Conclusion

The automatic generation of the abstract datapath pipeline models aims to identify the pipeline depth, and the pipeline stages from building associations of the register-to-pipeline stage. It is structured in two phases; the first phase concerns register analysis which outputs an intermediate representation of the datapath pipeline designs. The resulting graph is exploited in the second phase presented in this chapter.

In this phase, we focus on assigning registers to their pipeline stages. We specify the first stage as being the stage where the `PC` register is placed. Then we assign others according to two strategies; the first one is based on the dependencies between registers and the second on heuristics, based on `when` condition statements, which is used to determine additional relations between these registers. This heuristics is based on how the processor is coded. Furthermore, we have applied this analysis to various RISC-V Chisel-based processors with different pipeline depths ranging from 3 to 6 stages. We have illustrated the resulting abstract datapath pipeline models for several processor designs. We have also summarized the result statistics and discussed the experimental results for each processor design.

This thesis work aims to automatically build a framework from high-level designs to abstract models used then for different applications. Our purpose is to generate pipeline models for timing analysis of safety-critical systems. We present in the next Chapter 7 how to generate formal micro-architecture models from the generated abstract pipeline models with the goal of formally verifying timing-related properties such as the detection of timing anomalies.

7 - Automatic Construction of Formal Models From Abstract Models

The formal verification of timing properties over pipeline designs requires the construction of appropriate formal models. These models should consider the execution progress of the input program through the pipeline with cycle-accurate granularity thus, generating such infrastructure should be based on the abstract datapath model. As such, we aim to automatically generate formal models from the constructed abstract datapath pipeline models, seen in previous chapters, to which we then manually add the control path to capture the instruction flow through the pipeline stages.

We present in this chapter how to generate and then integrate formal pipeline models for the purpose of verifying timing properties of safety-critical systems such as those related to temporal predictability [6, 22]. We start with detailing in Section 7.1, the formal model generation from the abstract pipeline datapath models. Then, we illustrate this generation on various RISC-V processor designs in Section 7.2. Finally, we present as a use case the integration of such formal pipeline models into an existing model checking-based procedure for the detection of timing anomalies in Section 7.3. Finally, we evaluate the impact of these semi-automatically generated formal models in this integration through a series of synthetic benchmarks.

7.1 . Formal Modeling of Processor Pipeline Designs

The formal model required should integrate both program and architecture features, where the program provides a sequence of instructions to be executed while the architecture adds timing to this sequence, e.g., counting clock cycles while instructions pass through a pipeline. A formal pipeline design should be based on both a datapath and a control path. In the following, we combine automatically-constructed datapath pipeline models detailed in Chapters 5, 6 with control path logic, to define the notion of pipelined execution. We also need to map the instructions, i.e., the input program, onto these abstract pipeline models so as to finish integrating both software and hardware aspects. The two abstract models (i.e., datapath and control logic) and the program representation are then translated into formal specifications using the TLA+ language [36].

7.1.1 . TLA+ Specification from Datapath Pipeline Designs

We introduce next in, Listing 7.1, the skeleton and some notation elements of the TLA+ specification language to ease the understanding of the TLA+ code snapshots of the formal pipeline model that we introduce in this chapter.

Listing 7.1 – Template of a TLA+ formal model.

```
----------------------MODULE ModuleName--------------------
EXTENDS M1, M2, ... /* add declarations, definitions from the
    ↪ modules M1, M2 to the current module */
CONSTANTS .../* Constants declaration */
VARIABLES ... /* State variables */
-----------------------------------------------------------
Init == .... /* Initial state predicate */

Next == \/ Actions /* Next state relation */
        ...
Spec == Init /\ [] [Next]_<<Variables>> /* Specification */
Prop == ... /* Property to verify */
```

A TLA+ model is organized as a collection of modules, as presented in Listing 7.1, where each module contains a module body. The body consists of a sequence of statements, which can be a declaration, definition, assumption or theorem. In order to build large hierarchical specifications, we can build a new module on top of other modules. One way to do this is with the EXTENDS declaration at the beginning of the module, for example including *M1, M2* in module `ModuleName`. The state variables are declared after the keyword VARIABLES.

We present next in Listing 7.2 the header of the RISC-V Mini TLA+ formal model. The formal model is described in module `MODULE formal_model_mini`. The module includes other modules, such as the standard modules `Sequences`, `Integers`, `TLC` required to manipulate procedures, integers, and assertions. Furthermore, it includes the module `Instructions` defining a representation of the instruction set architecture (ISA). RISC-V Mini is a processor design with three pipeline stages. Thus, the state variables are built around these stages and `currCycle` variable to describe the cycle-accurate granularity.

Listing 7.2 – TLA+ snapshot code of RISC-V Mini

```
-------------MODULE formal_model_mini ----------------------
EXTENDS Sequences, Instructions, Integers, TLC
CONSTANTS Program
VARIABLES currCycle, stage_1, stage_2, stage_3
-----------------------------------------------------------
...
```

TLA+ allows us to specify the system through a single TLA+ formula containing, in particular, an initial-state predicate (`Init`) and a next-state relation (`Next`) built from actions, capturing the update of variables. More precisely, an action relates the values of variables in the current state `x` to their values in the next state `x'`. The specification could contain several other operators that can be parameterized and used later for the next-state relation. Furthermore, the TLA+ specification also contains the properties (i.e., temporal logic formulas) which are to be verified. We employ model-checking techniques, using the TLC tool [34], to explore the TLA+ specification while verifying the required timing properties. The initial state and the next-state relation are defined through the next algorithms.

Algorithm 5 describes the formalization of a datapath pipeline model from the constructed datapath abstract pipeline model. Thus, we consider as input the resulting graph of the datapath pipeline generated with the approach described in Chapters 5, and 6 and an operator $ElmtProg$ to which the program is then assigned. The graph corresponds to the resulting registers-to-pipeline stages and their dependencies, $IR_{\mathcal{P}}$. The program assigned to the operator $ElmtProg$ represents a sequence of the instructions that are described according to the representation of the RISC-V ISA. We detail in Section 7.1.2 the corresponding representation. The algorithm output conforms to the formal specification of the datapath pipeline, formalized using the TLA+ language.

Algorithm 5: Datapath TLA+ Model Generation

Input : $IR_{\mathcal{P}}$, $ElmtProg$ and $Pregs = Regs \mapsto Stages$ - pipeline datapath of $\mathcal{P}$

Output: $InitPipe, UpdatePipe$ - Datapath TLA+ specification of $\mathcal{P}$

1 $State\_var \leftarrow$ (`currCycle`, `Stages`) `/* State variables. */`
2 $InitPipe \leftarrow$ `formal_Init`($Pregs$, $State\_var$)
3 $SrcReg \leftarrow$ `MaxCaseRegs` ($IR_{\mathcal{P}}$)`/* Regs assigned with max case. */`
4 $PredRegSel \leftarrow$ `FRegsSelection` ($IR_{\mathcal{P}}$) `/* Regs used as a selection of combinatorial logic(mux) */`
5 $UpdatePipe \leftarrow$
`formal_Next`($Pregs$, $State\_var$, $IR_{\mathcal{P}}$, $SrcReg$, $PredRegSel$, $ElmtProg$)

Our TLA+ specification targets the execution of instructions through the pipeline stages with cycle-accurate granularity. Thus, we define first the state variables, in line 1, which includes the pipeline depth (i.e., pipeline stages, in variables `Stages`) and clock cycles defined in `currCycle` variable. Second, we distinguish two operators for our specification – initial and update. The initial operator $InitPipe$, line 2 in Algorithm 5 initializes an empty pipeline described in `formal_Init` function detailed in Algorithm 6.

Thus, we consider the resulting graph of the datapath pipeline model `Pregs`,

then we initialize each register in each pipeline stage with `empty`, lines 3-4 in Algorithm 6. We denote by `R`, line 4 in Algorithm 6 an operator to assign values to the registers and output TLA+ records $Stage_i\_assign$ with fields being the registers assigned to the corresponding pipeline stage. The operator `Stage_upd` assigns the registers updated with `empty` to each pipeline stage. As mentioned, these fields are initialized with `empty`, where `empty` defines an empty content of a register. Then, we output the `StageInit`, which contains the TLA+ records of all the pipeline stages, lines 4-5 in Algorithm 6.

The update state is produced with the `formal_Next` function, line 5 in Algorithm 5. This function is detailed in Algorithm 7. We model the pipeline update where each stage update is modeled by an action on its registers and their dependencies. We consider the corresponding dependencies graph $IR_{\mathcal{P}}$. We check the registers assigned with the `max_case` 1 as they are specified in `MaxCaseRegs` function, line 3 in Algorithm 5. Then, we only keep in the $IR_{\mathcal{P}}$ graph its register dependency, which corresponds to the maximal stage, line 3-4 in Algorithm 7. Since the $IR_{\mathcal{P}}$ graph includes all the dependencies between registers, for the register assigned with the `max_case` 1 which has two register sources, we only keep the dependency corresponding to the register with the maximal stage. Indeed, this choice is not fortuitous, it is consistent with the WCET analysis, where we consider the longest path. Afterward, we update each stage by updating its pipeline registers, line 5 in Algorithm 7.

Algorithm 6: `formal_Init` function

```
1 Function formal_Init(Pregs, State_var):
2     StageInit ← TRUE
3     foreach (Stage_i, Regs) ∈ Pregs do
4         Stage_i_assign = Stage_upd(Stage_i, R(Regs, empty))
          StageInit = StageInit ∧ Stage_i_assign
5     return StageInit;
```

We update the register `PC` of the first stage with the first instruction of the input program, lines 6-8 in Algorithm 7. Furthermore, we consider the registers for other pipeline stages and update each register with its dependencies, lines 9-26. We distinguish between three cases. The first case describes the register without dependencies, it is updated with `empty` content, lines 11-12. We denote by `Reg_updt` an operator to assign a value to the register. The second case targets the registers placed with the linear case, lines 13-15. The register is updated with the value of its source register using the function `MapStage`. This function takes two arguments: the pipeline stage of the source register and the source register, then it outputs the value of type string for updating the register, line 14. Finally, the last case updates registers with multiple dependencies, lines 16-26. Thus, we present next the dependencies related to the control conditions for each register, line 17. We address

first the conditions associated to the forwarding mechanism and defined in `bypass_cond` operator, lines 19-21, then, we specify the other conditions specified in `select_cond` operator, lines 23-24, which are all to be evaluated to true or false according to the execution pattern of the instructions.

We note that all the registers dependencies used as a selection relation in the combinatorial logic do not exist in the update state of the registers, line 4 in Algorithm 5 and line 19 in Algorithm 7. Since these registers are used in the combinatorial components (e.g., multiplexers) in selection conditions, our algorithm removes all these dependencies from the graph `IRp` to represent in the formal model. Since our approach that generates the datapath pipeline model outputs more register dependencies (over-approximation), the analysis does not include the combinatorial logic that distinguishes between the selection and data dependence relations. Afterwards, we output the TLA+ record in `StageUpdate` variable, which contains a record of each pipeline stage `R` defined in $Stage_i\_assign$ variable, lines 26-27 in Algorithm 7.

Our TLA+ specification `Spec` is defined with an initial state `Init` and a state transformer `Next` as follows:

```
Spec == Init /\ [][Next]_<< State_var >>
```

The initial state is a conjunction between the resulting empty pipeline stages defined in `InitPipe` and the remaining state variable `currCycle`:

```
Init == /\ InitPipe
        ....
        /\ currCycle = 0
```

The transition state is also defined as a conjunction between the update of pipeline stages and `currCycle` variable for an ideal pipeline as follows:

```
Next == ... /\ UpdatePipe
        ... /\ currCycle' = currCycle + 1
```

The TLA+ specification integrates both the datapath and control path of the pipeline. Thus, each action, `Init` and `Next` integrates the generated pipeline stages and the control path to define the execution and entail changes in the datapath process.

Algorithm 7: `formal_Next` function

```
1  Function formal_Next(Pregs, State_var, IR_P, SrcReg, PredRegSel, Prog):
2      StageUpdate ← TRUE
3      if !SrcReg.isEmpty then
           /* keep only the max path for max_case */
4          IR_P = IR_P \ SrcReg
5      foreach (Stage_i, Regs) ∈ Pregs do
6          if Stage_i == 1 then
7              Stage_i_assign = Stage_upd(Stage_i, R(Regs, Prog))
               StageUpdate = StageUpdate ∧ Stage_i_assign
8          else
9              foreach reg ∈ Regs do
10                 SrcsReg ← IR_P.filter(dest == reg) ; /* filter only
                    the sources of reg  */
11                 if SrcsReg.size == 0 then
12                     Reg_updt = (reg, empty)
13                 else if SrcsReg.size == 1 then
14                     Updt = MapStage(Pregs(SrcsReg), SrcsReg)
15                     Reg_updt = (reg, Updt)
16                 else
                       /* define select conditions for multiple
                          sources of reg */
17                     foreach src ∈ SrcsReg do
                           /* Bypass conditions */
18                         IfThen ← TRUE
19                         if (Pregs(src) >= Stage_i ∧ (src, reg) ∉
                            PredRegsSel) then
20                             Updt = MapStage(Pregs(src), src)
21                             IfThen = IfThen ∪ (bypass_cond, Updt)
22                         else
                               /* Selection conditions */
23                             Updt = MapStage(Pregs(src), src)
24                             IfThen = IfThen ∪ (select_cond, Updt)
25                     Reg_updt = (reg, IfThen)
26             Stage_i_assign = Stage_upd(Stage_i, R(Reg_updt))
               StageUpdate = StageUpdate ∧ Stage_i_assign
27     return StageUpdate;
```

7.1.2 . Control and ISA Modeling TLA+ Specification

In our workflow described in Figure 4.1, the TLA+ formal models are based on the abstract pipeline models and the program executed on this pipeline. We add the control logic to the automatically constructed model of the datapath pipeline. The control path manages the control of data movement between components in the pipeline. Since we aim to build pipeline models for timing analysis, we should integrate how instructions advance through this pipeline model so as to prove the timing properties. Currently, the integration of the control path is mainly manual, as is the abstract Instruction Set Architecture (ISA) semantics for the same design. The *Program component*, defined in figure 4.1 and introduced at the beginning of Algorithm 5 as input, is responsible for defining ISA-level execution patterns on the constructed pipeline model, describing the necessary infrastructure to reason about timing properties of the design.

The control logic addresses all the signals that ensure that data flow into pipeline stages and that instructions are properly executed. It includes the selection signals of combinatorial and sequential circuitry: multiplexer, functional units, registers, etc. These signals are related to the instruction patterns and are designed to determine pipeline status such as stalling, branching logic etc. These execution patterns are specified as a set of control conditions defined in pipeline update `UpdatePipe`. While their placement is automatically generated, their actual semantics is added by hand. Pipeline status determines the progress status of the execution. Thus, in order to correctly add the control logic, we need to specify the ISA instructions of the processor, then analyze the implementation of each instruction to determine the placement of control points that affect the register transfer, and finally examine the input program to define pipeline status (stall, kill, etc).

We also need to introduce the abstract RISC-V ISA modeling which is specified also by hand. The abstract ISA consists of a set of instructions grouped in classes, sharing the same execution patterns, expressed in the form of a set of conditions `conds` as follows:

```
insts == [
[type |-> "alu", rd |-> "rd", rs1 |-> "rs1", rs2 |-> "rs2",
 conds = {cond_1, .../* Conditions of type select_cond */ }],
[type |-> "jal", rd |-> "rd", rs1 |-> "empty", rs2 |-> "empty",
 conds = {cond_2,.../* Conditions of type select_cond */ }],
[type |-> "lw", rd |-> "rd", rs1 |-> "rs1", rs2 |-> "empty",
 conds = {cond_3,... /* Conditions of type select_cond */ }],
 ...]
```

The example above describes three types of instruction classes. Adding to selection conditions specified in `conds`, the description of the instructions also exposes the source and destination registers, `rs1/2` and `rd` respectively, initialized with some default values. We specify in TLA+ these instruction

operands to express the dependencies between instructions and then identify the forwarding mechanism. Then, we define the bypass conditions `bypass_cond` by comparing the instruction operands defined in the registers rs1, rs2, and rd. We detail the semantics and the body of these conditions in the application case studies in Section 7.2.

Now for the pipeline status, we model the stalling logic. This stalling logic is also defined by hand and based on how the pipeline interacts with the memory system. For example, we could encode such a logic that stalls the pipeline for five cycles whenever there is memory access caused by load/store instructions. The predicates `stall` and `stall_cond` state exactly this behavior.

```
stall == stage_MemStage.MemStage_pc.inst.type = "lw"
        \/ stage_MemStage.MemStage_pc.inst.type = "sw"
stall_cond == stall /\ stall_delay < 5
```

The pipeline stages with the stalling logic are modeled in `StallPipe` operator and described in `formal_Stall` function in Algorithm 8. We specify the pipeline stall from the first stage until the stage for the memory access defined in `MemStage` variable, line 4. We stall the pipeline for the memory latency cycles for all the pipeline stages before the memory stage, lines 4-7, and we keep the other stage advancing through the pipeline, we update all the registers in this pipeline stage with `empty`, in $Stage_i\_assign$ line 8. Then, we output the pipeline stages on the stalling state, lines 9,10 in Algorithm 8.

Algorithm 8: `formal_Stall` function

1 **Function** `formal_Stall`($Pregs$, $State\_var$, $MemStage$, $IR_{\mathcal{P}}$)**:**
2 $\quad stageStall \leftarrow TRUE$
3 $\quad i \leftarrow 1$
4 $\quad$ **while** *i <= MemStage* **do**
5 $\quad\quad$ **foreach** $Stage_i \in Pregs$ **do**
6 $\quad\quad\quad stageStall = stageStall \wedge Stage\_upd(Stage_i, UNCHANGED)$
7 $\quad\quad\quad i = i + 1$
8 $\quad Stage_i\_assign = Stage\_upd(Stage_{MemStage+1}, R(Regs, empty))$
9 $\quad stageStall = stageStall \wedge Stage_i\_assign$
10 **return** $stageStall$;

We integrated the stalling logic into the pipeline definition via a TLA+ operator `Stall`, which is defined as a conjunction between the pipeline stalling stages and `currCycle`, `delay_cycle` variables. `stall_delay` is incremented until `stall_cond` becomes false.

```
Stall == ... /\ stall_cond
             /\ StallPipe
```

```
                    /\ stall_delay' = stall_delay + 1
                    /\ currCycle' = currCycle + 1
```

The update state `Next` integrates the pipeline updates without stalling in `TransPipe` and with the stalling logics in `Stall`. The pipeline update is defined as follows:

```
TransPipe == ... /\~stall_cond
                 /\ UpdatePipe
                 /\ currCycle' = currCycle + 1
                 /\ stall_delay' = 0
```

The state transformer of `Spec` is refined with the `Stall` as follows:

```
Spec == Init /\ [][TransPipe \/ Stall]_<< state_vars >>
```

As such, we distinguish between two types of pipeline progress: pipeline progress when there is stalling, via `Stall` and pipeline progress without stalling, using `Next`. This modeling accounts for the hardware part of the specification. Then, in this modeling we map the input program, given as a sequence of instructions (specified by the abstract ISA) and having the conditions to guide their execution through the pipeline. This is the software aspect of the formal model.

7.2 . Application to RISC-V Processors Case Studies

RISC-V Sodor 5-stage Pipeline. We detail next our formal model for the pipeline of the Sodor 5-stage processor; this model combines an automatically generated datapath with manual extensions for the control and ISA-level semantics. We refer to each aspect in the following description. Our TLA+ specification `Spec` is defined with an initial state `InitPipe` and a transition states defined with `TransPipe` and `StallPipe` as follows:

```
Spec == InitPipe /\ [][TransPipe \/ StallPipe]_state_vars
```

The state variables include the pipeline depth of RISC-V Sodor 5-stage and the cycle level granularity, which corresponds to:

```
state_vars = << stage_1, stage_2, stage_3, stage_4, stage_5,
               currCycle >>
```

The initial state is an empty pipeline. Thus `InitPipe` contains the initialization of the pipeline stages. We present next the registers of stage 3, which correspond to the registers described in Figure 6.6. As described in `formal_Init` function in Algorithm 6. The record `R` defined in $Stage_i\_assign$ corresponds to the initialization of the stage 3 registers with `empty`. The initialization for `stage_3` is as follows:

```
InitPipe == ... /\ stage_3 = [ ex_pc |-> empty,
                               ex_alu_op1 |-> empty,
                               ex_alu_op2 |-> empty,
                               ex_rs2_data |-> empty ]
            ... /\ currCycle = 0
```

The TransPipe models the progress of the pipeline where there is no stalling.

```
TransPipe == ..../\~stall_cond
                 /\ update_stage_3(regs) /\ ...
                 /\ currCycle' = currCycle + 1
```

The operator update_stage_3(regs) stands for the following partial update of stage_3 (i.e., only 2 registers, ex_pc and ex_alu_op1 out of 13 registers are shown):

```
stage_3' = [ ex_pc |-> stage_2.dec_pc,
   ex_alu_op1 |-> IF bypass_3_5 THEN stage_5.wb_wbdata
                  ELSE IF bypass_3_4 THEN stage_4.mem_alu_out
                  ELSE IF bypass_3_3 THEN stage_3.ex_alu_op2
                  ELSE IF cond_1 THEN stage_5.wb_wbaddr
                  ELSE IF cond_2 THEN stage_4.mem_wbaddr
                  ELSE IF cond_3 THEN stage_2.ex_wbaddr
                  ELSE IF cond_4 THEN stage_2.dec_pc
                  ELSE stage_2.dec_inst ... ]
```

The update of register ex_alu_op1 is defined with respect to the set of dependency relations between registers, where each possible update is guarded by a condition bypass_cond or select_cond as specified in the Algorithm 7. Each condition is associated with the corresponding source register defined in Updt with the function MapStage in the Algorithm 7. For example, for the update of register ex_alu_op1 with the source register wb_wbdata, it corresponds to the value stage_5.wb_wbdata. The condition bypass_cond is defined for each register as bypass_x_y, while select_cond as cond_z. These conditions correspond to the blue and, respectively, black edges of the datapath from Figure 6.8. More precisely, the ISA model defines, for each instruction class, an execution pattern on the pipeline in the form of a set of conditions cond_z. Now, the semantics of cond_z is as follows: given an instruction i in a pipeline stage, if the progression of i is conditioned by cond_1, then cond_1 evaluates to true only if it is among the stated conditions of the instruction type of i. The semantics of bypass_x_y is also given by hand and states the true data dependencies, using the registers rs1, rs2 and rd of instructions found in the pipeline stages x and y as follows:

```
bypass_3_3 == stage_2.dec_pc.inst.RS1 = stage_3.exe_pc.inst.RD
bypass_3_5 == stage_2.dec_pc.inst.RS1 = stage_5.wb_pc.inst.RD
bypass_3_4 == stage_2.dec_pc.inst.RS1 = stage_4.mem_pc.inst.RD
```

For the stalling logic, it is integrated with the operator `StallPipe` with `stage_4` as the assumed memory access stage in the case of the Sodor 5-stage pipeline. The memory stage is added manually. We stall the pipeline stages from `stage_1` to `stage_4` with `UNCHANGED`, as defined in `stageStall` variable in the Algorithm 8. The write-back happens in `stage_5` where instructions can leave the pipeline and the register `wb_wbdata` is updated with `empty` as explained in the Algorithm 8, line 8.

```
StallPipe == /\ stall_cond
             /\ stage_5' = [ wb_wbdata |-> empty ]
             /\ stall_delay' = stall_delay + 1
             /\ currCycle' = currCycle + 1
             /\ UNCHANGED << stage_1, stage_2, stage_3, stage_4>>
```

Rocket Pipeline. We illustrate now the formal model for the pipeline of the Rocket processor. The pipeline depth of Rocket is 6 stages. Thus, the state variables are defined as follows:

```
state_vars = << stage_1, stage_2, stage_3, stage_4, stage_5,
              stage_6, currCycle >>
```

We model the pipeline update where each stage updates the registers according to their dependencies. As detailed in Algorithm 5, line 3, we consider the corresponding graph of the register dependencies IRp with only the longest path in the `max_case` $\boxed{1}$ dependencies. Rocket processor presents the `max_case` $\boxed{1}$ to assign the register `ex_reg_pc` as detailed in Figure 6.2. The update of this register in stage 4 is as shown:

```
      stage_4' = [ ... ex_reg_pc |-> buf_pc ...]
```

We next illustrate the ISA model for two instruction classes to map the input program into the pipeline models to finish integrating both software and hardware aspects of formal models. We model the update of stage 6 with only one register. The following TLA+ description corresponds to the datapath pipeline model in Figure 6.8.

```
stage_6' = [ wb_reg_wdata |->
                        IF cond_1 THEN stage_5.mem_reg_pc
                        ELSE IF cond_2 THEN stage_5.mem_reg_wdata
                        ELSE stage_5.mem_reg_inst
             ...]
```

We refer to the ISA model of instruction classes to define their execution patterns and to evaluate the corresponding conditions to true.

```
insts == [
[type |-> "add", rd |-> "rd", rs1 |-> "rs1", rs2 |-> "rs2",
```

```
  conds = {cond_2} ],
 [type |-> "jal", rd |-> "rd", rs1 |-> "empty", rs2 |-> "empty",
  conds = {cond_1} ],
  ...]
```

The input program is a sequence of instructions. Thus, we follow the progression of each instruction on the pipeline. For example, when an instruction `i` is in a pipeline stage, if it is a jump instruction, then `cond_1` is evaluated to true since `cond_1` is among the stated conditions of the jump instruction type.

Thus, we semi-automatically generate the formal models from the processor designs. Indeed, we automatically generate the datapath pipeline models and add the control logic by hand to guide the execution. Furthermore, we have also generated pipeline models for various RISC-V processor designs such as Sodor 3-stage, RISC-V Mini [28], Kyogen [29], Rocket [2] and Fuxi [30].

Table 7.1 – Statistics on TLA+ specification of RISC-V processor designs

	LOC(automatic)	*LOC(manual)*	*automatic*
Sodor 3-stage	40	16	71%
RISC-V Mini	55	16	77%
Sodor 5-stage	145	25	85%
KyogenRV	239	37	86%
Rocket	103	25	80%
Fuxi	297	32	90%

Table 7.1 presents statistics related to the manual and automatic lines contained in the TLA+ specification for each processor design. The automatic LOCs include the generated datapath model, while the manual LOCs include the control (stalling) and the map of the input program into the TLA+ specification. For instance, the TLA+ specification of the RISC-V Sodor 5-stage pipeline is around 170 lines out of which, 145 lines are automatically generated and only 25 lines are added by hand. Furthermore, the number of automatic lines is related to the size of the intermediate graph (i.e., pipeline registers). It increases for the designs with a large number of pipeline registers assigned in their pipeline stages, as for KyogenRV and Fuxi processor designs.

Next, we present the impact of control logic (i.e., the pipeline stalling) which was added manually on the TLA+ specification. We report in Table 7.2 the results for various processor designs, where we evaluate the impact of checking for stalling conditions, on the TLC running time with a property as a `currCycle` is lower than 1000. We design by `processor_complete` the formal specification of both datapath and control logics, while the `processor_datpath` reports the results for only the datapath formal model. The formal pipeline models are built to address the execution of basic blocks, basically, not very long sequences of instructions. Thus, the input program size on which we

experiment is relatively small, up to 200 instructions. We can observe that adding the stalling logic in the TLA+ specification increases the runtime for all the processor designs. The evaluation includes the stalling conditions and the pipeline progress in the case of the stalling.

Table 7.2 – Impact of stalling checks on TLA+ pipeline models.

	$\#Instr$	$Time$ (s)	$\#Instr$	$Time$ (s)
Mini_complete	100	02	200	03
Mini_datpath		01		02
Sodor3_complete	100	02	200	03
Sodor3_datpath		01		02
Sodor5_complete	100	16	200	23
Sodor5_datpath		13		18
Kyogen_complete	100	21	200	29
Kyogen_datpath		18		25
Rocket_complete	100	17	200	23
Rocket_datpath		15		17
Fuxi_complete	100	17	200	26
Fuxi_datpath		10		18

7.3 . Integration of Formal Pipeline Models in a Timing Anomaly Detection Procedure

We detail next the integration of the generated pipeline models into an existing model checking procedure for the detection of timing anomalies. We start with presenting the corresponding definition of timing anomalies and its TLA+ formal encoding. Then, we report the evaluation synthesizes of the integration through various benchmarks, on a number of RISC-V processor designs.

7.3.1 . Model Checking for Timing Anomalies

Static WCET analysis can be complicated by the presence of an undesired timing phenomenon called timing anomalies [106]. Indeed, they are a threat to the computation of safe bounds, thus it is necessary to explore all the paths

in order to compute bounds. A timing anomaly is a counter-intuitive timing behavior in the sense that the local worst-case timing behavior does not result in the global worst-case performance. Figure 7.1 illustrates the timing anomalies definition on two executions, in this case, it is a counter-intuitive timing anomaly.

Timing anomalies can be produced from shared resources between architecture components or an unpredictable timing behavior coming from the interaction with the memory system. Reasoning about timing anomalies means reasoning on pairs of different execution traces corresponding to the same input program. Thus formal models of both the architecture and the input program are required.

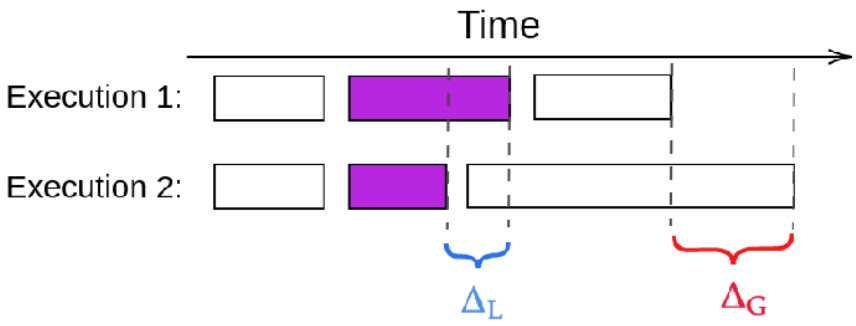

Figure 7.1 – Example of counter-intuitive timing anomalies

In our work, we consider the variabilities in execution latency resulting from the interaction between the pipeline and the memory system. Moreover, we integrate our semi-automatically generated pipeline model into an existing procedure for the detection of timing anomalies [6], which encodes a definition of timing anomalies based on the concept of locality, proposed in [104]. More specifically, this work defines a timing anomaly regarding local and global variations.

We illustrate next this integration of the semi-automatically generated state pipeline model. We define first a `state_vars` extension to capture local and global timing variations as part of a new state variable, `asi`: local variation defined by `delta`, global variation identified with `ET` specified for two traces. `pcid` and `pathid` rely to the program representation. We define their semantics later.

```
asi = [ delta |-> [n \in 1..2 |-> 0],
        ET |-> [n \in 1..2 |-> 0],
        pcid |-> Program.did,
        pathid |-> 1]
```

A timing anomaly is characterized by a pair of execution paths when we compare the local worst-case variations array (i.e., `delta`) with the global worst-cases array (i.e., `ET`). Figure 7.2 describes the timing anomalies intuition from the previous definition. Hence, the property is encoded as:

```
Prop == ~(asi.delta[1] < asi.delta[2] /\ asi.ET[1] > asi.ET[2])
```

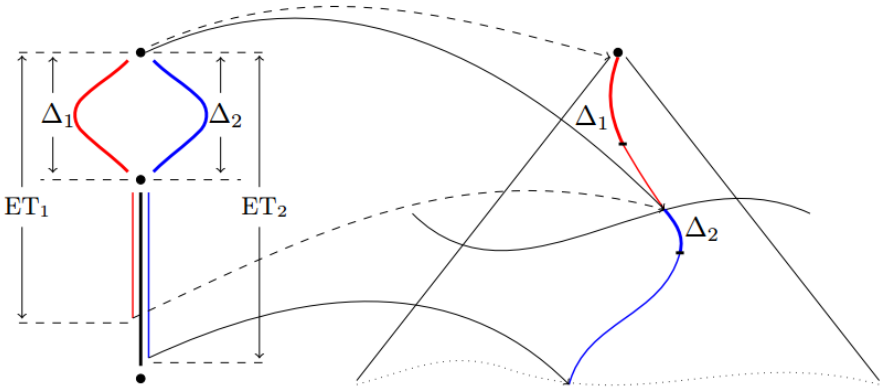

Figure 7.2 – Timing anomalies intuition, from [6]

The formal pipeline model should be evaluated on two traces, where each trace is a sequence of instructions, according to the `insts` definition provided in Section 7.1.2. Each instruction is represented by several parameters, the program counter `pc`, the instruction type in which we specify the operands `inst`, and the latency `lat` (to model the variability which is coming from the interaction with the memory system). Finally, the procedure for the detection of timing anomalies evaluates the pair of traces, given in `fst` and `snd` fields and specified in `asi.pathid` field to be initialized with the first trace, and with a locality defined whenever the instruction with `pc` given in field `did` is in a particular pipeline stage (e.g., `stage_3`). We define in the `asi.pcid` field the instruction identified in `did` program field.

```
Program <- [ fst |-> << [ pc |-> 1, lat |-> {1, 10},
                          inst |-> [type |-> "alu" ... ] ],
                        [ pc |-> 2, ... ], ... >>,
            snd |-> << [ pc |-> 1, lat |-> {1, 10},
                         inst |-> [type |-> "alu" ... ] ],
                       [ pc |-> 2, ... ], ... >>,
            did |-> 2 ]
```

The update state is extended to integrate the locality in instruction executions. The previously defined operator `TransPipe` of Sodor 5-stage is extended to fully explore the instruction latencies of an instruction in `stage_2`,

before entering the locality-related stage, i.e., `stage_3`. Moreover, `TransPipe` is also extended with the corresponding update of the `asi` state variable, based on the system state (i.e., the self-explanatory conditions `condFlushWB` and not `moreIns`) as follows:

```
TransPipe == \exists lat \in stage_2.dec_pc.lat :
               ...
             /\ asi' = IF condFlushWB /\ ~ moreIns
                       THEN [ asi EXCEPT!.ET[asi.pathid]=
                            currCycle + 1 ]
                       ELSE IF asi.pcid = stage_2.dec_pc.pc
                       THEN [ asi EXCEPT!.delta[asi.pathid] =
                            stage_2.dec_pc.lat +
                            asi.delta[asi.pathid] ]
                       ELSE asi
```

Once the first trace `fst` is explored, the procedure for the detection of timing anomalies reloads the second trace `snd`, extending the specification with the following straightforward operator `Reload`:

```
    Reload == condReload
              /\ ...
              /\ asi' = [ asi EXCEPT!.pathid = 2 ]
```

Where the corresponding conditions for Sodor 5-stage are defined as follows:

```
condFlushPip == stage_5.wb_pc.pc = 0 /\ stage_4.mem_pc.pc = 0
          /\ stage_3.exe_pc.pc = 0 /\ stage_2.dec_pc.pc = 0
          /\ stage_1.if_pc.pc = 0
condFlushWB == stage_4.mem_pc.pc = 0 /\ stage_3.exe_pc.pc = 0
         /\ stage_2.dec_pc.pc = 0 /\ stage_1.if_pc.pc = 0
condReload == /\ asi.pathid = 1
              /\ ~moreIns
              /\ condFlushPip
```

`condFlushWB` is used to update the global variation `ET` when the last instruction is in the last pipeline stage (write-back), and there is no more instruction in the input program to progress. `condFlushPip` is defined in order to reload the second program trace `snd`, where all the instructions of the first trace `fst` are already left the pipeline.

As a consequence, The TLA+ formal specification `Spec` is extended and becomes:

```
Spec == InitPipe /\ [][TransPipe \/ StallPipe
       \/ Reload]_<< state_vars >>
```

7.3.2 . Experimental Results

The integration of the generated formal pipeline models into an existing procedure for the detection of timing anomalies allows us to evaluate the effectiveness of our work. As such, we evaluate its impact on all of the previously considered RISC-V processor designs, with their different characteristics, including the number of datapath pipeline registers, the pipeline depth, and the execution patterns specified with the tuple (instructions, control conditions). Furthermore, we evaluate also the path/condition coverage of the models through various instruction classes. We also measure the model checking-based exploration of the state space in both its width and depth. The depth of the state space is expressed through a variation in the size of the input traces (i.e., instruction sequences), while the width of the state space is impacted by the latency variations of various instructions in the input traces. We detail next the evaluation of the processor designs.

We experiment with our integration procedure on two processor designs, RISC-V Sodor 5-stage and Rocket. We select these two due to their complexity and variety in terms of execution patterns and the pipeline structure, which is mono-module for RISC-V Sodor and multi-module for Rocket design.

The TLA+ specification of the RISC-V Sodor 5-stage pipeline, integrated into the procedure for the detection of timing anomalies, is about 320 lines, while it is composed of 250 lines for the Rocket processor. We then employ the TLC model checker to explore all the latency variations on the traces and detect timing anomalies. The input traces are synthetic benchmarks featuring variable latencies on various instructions and which are explainable by external static cache analyses [12] or measurement strategies [107].

We conduct experiments on three variants of each considered processor. First, a simple version where each pipeline stage is represented by a single register. This model is also used in [6] and named `simple` in our experimental evaluation. Second, the pipeline from Figure 6.6 and Figure 6.8, which are a subset of the (semi-)automatically generated pipeline model of Sodor RISC-V 5-stage and Rocket respectively, named `processor_subset` in the experiments. Finally, the complete pipeline model which is presented in Section 7.2 and named `processor_complete` in the following.

The experimentation is conducted on a quad-core `Intel Core i7` at a frequency of 1.90GHZ and 16GB RAM memory and with the TLA+ Toolbox using the TLC model checker version 2.16. We summarize in Tables 7.3-7.8 the experimental results for multiple test scenarios. The overall goal is to measure the impact of the generated pipeline `processor_complete` (over `processor_subset` and `simple` implementations) on synthetic traces with and

without timing anomalies. The general format of these tables is seven columns. We specify in the first column the pipeline models, in the second the corresponding size of pipeline registers, and in the third and fourth columns, some trace-related statistics namely, the trace size and number of different execution traces due to latency variations. The last three columns are model checking statistics, on the running time (the fifth column) and on the search space (the sixth column on the size of the longest behavior and finally the seventh column on the number of all states).

First, we conduct experiments on proving the absence of timing anomalies, shown in Tables 7.3, and 7.4, with respect to a given scenario, detailed next. We consider a synthetic trace of 200 instructions in which we specify a pair of latencies for the instruction of interest (i.e., corresponding to hit/miss cache activity) as well as other arbitrary latencies for the rest of the instructions. Then, we employ bounded model checking with a bound of 1000 cycles, roughly established in a worst-case scenario wrt. the provided latencies and instructions. Then, the TLC model checker fully explores the state space, proving the absence of timing anomalies up to the provided bound with respect to the considered scenario.

Table 7.3 – Experimental results for the absence of timing anomalies for RISC-V Sodor processor.

	$\#Regs$	$\#Instr$	$\#Execs$	$Time\ (s)$	$Diam\ -eter$	$States\ Found$
`simple`	5	200	65536	32	1170	801678
`sodor_subset`	11			40		
`sodor_complete`	37			54		

Table 7.4 – Experimental results for the absence of timing anomalies for Rocket processor.

	$\#Regs$	$\#Instr$	$\#Execs$	$Time\ (s)$	$Diam\ -eter$	$States\ Found$
`simple`	6	200	65536	38	1172	801707
`rocket_subset`	16			46		
`rocket_complete`	30			59		

Next, we present, in Tables 7.5-7.8, results when timing anomalies are detected. We have experimented with several synthetic benchmarks with variable execution trace size as described in Tables 7.5, and 7.6, and with variable instruction latencies, reported in Tables 7.7, and 7.8 for Sodor 5-stage and Rocket respectively. The trace size is gradually increased, from 40 to 200

instructions, corresponding to reasonably-sized basic blocks, producing differences in the running times between `simple` and `processor_complete` models for longer traces (i.e., measuring the impact on the depth of the state space, in Tables 7.5, 7.6). Then, we fix the trace size to 200 instructions and increase the number of variabilities due to instruction latencies, up to 4M different scenarios, which are explored by the TLC model checker. This experimentation also reports runtime differences between `simple` and `processor_complete`, which are proportional with respect to the number of exposed scenarios, measuring the impact on the width of the state space.

We can observe for all Tables 7.5-7.8 the variable trace sizes and, more precisely, instruction latencies significantly impact the running time for the three pipeline variants. Indeed, the processors differ in the number of pipeline registers that construct the pipeline stages as the state variables. However, the statistics on the diameter and number of states found by the TLC model checker are similar for the three pipeline variants. The reason is that we execute the same sequence of instructions for the same number of pipeline stages, and the state variables are pipeline stages that group registers and not individual registers (which could lead to different state updates). Furthermore, the stalling logic is also the same for the three variants of the same processors, focusing on blocking the pipeline up to the memory stage. Thus, the difference between the pipeline variants relies on the pipeline registers that are included in the pipeline stages as the state variables. Moreover, the state variables' size is not changed. Therefore, the variability directly impacts the runtime and keeps the same diameter and states found for the three variants.

We have experimented with various complete versions of the processor designs with different pipeline depths from 3 to 6 pipeline stages with different sizes of the intermediate graph representation from previous chapters. Table 7.9 presents the experimental result with 200 instructions in the input program and a variability of 4M scenarios explored by the TLC model checker. These benchmarks report runtime differences between all these pipeline designs, which are related to different parameters: the pipeline depth, 3 stages for Sodor 3-stage and RISC-V Mini, 5 stages for Kyogen, and Fuxi, and finally 6 for Rocket. Furthermore, it depends also on the number of registers containing the graph and their updates according to control conditions and bypassing dependencies. For instance, the TLA+ specification of the processors Kyogen and Fuxi takes longer to run than the other processors with 5 stages, this is due to their pipeline registers assigned in the graph as reported in Table 6.3. The depth of state space (i.e., the diameter) increases with the pipeline depth. For example, it can reach 828 for processors with 3 pipeline stages: RISC-V Sodor 3-stage and RISC-V Mini, 832 for Kyogen, Sodor 5-stage, and Fuxi pro-

Table 7.5 – Experimental results for timing anomalies detection with multiple program depth for Sodor 5-stage processor.

	$\#Regs$	$\#Ins$	$\#Exc$	$Time\ (m:s)$	$Diam\ -eter$	$States\ Found$
simple	5	40	1024	00:24	172	83288
sodor_subset	11			00:27		
sodor_complete	37			00:45		
simple	5	80	1024	01:09	332	167768
sodor_subset	11			01:23		
sodor_complete	37			01:54		
simple	5	120	1024	01:29	492	252248
sodor_subset	11			02:01		
sodor_complete	37			02:51		
simple	5	160	1024	01:45	652	336728
sodor_subset	11			02:07		
sodor_complete	37			02:59		
simple	5	200	1024	01:57	812	421208
sodor_subset	11			02:20		
sodor_complete	37			03:10		

Table 7.6 – Experimental results for timing anomalies detection with multiple program depth for Rocket processor.

	$\#Regs$	$\#Ins$	$\#Exc$	$Time\ (m:s)$	$Diam\ -eter$	$States\ Found$
simple	6	40	1024	00:02	174	83321
rocket_subset	16			00:03		
rocket_complete	30			00:04		
simple	6	80	1024	00:05	334	167801
rocket_subset	16			00:17		
rocket_complete	30			00:27		
simple	6	120	1024	00:09	494	252281
rocket_subset	16			00:42		
rocket_complete	30			00:52		
simple	6	160	1024	00:14	654	336761
rocket_subset	16			01:30		
rocket_complete	30			01:45		
simple	6	200	1024	00:28	814	421241
rocket_subset	16			01:50		
rocket_complete	30			02:05		

Table 7.7 – Experimental results for timing anomalies detection with multiples latencies for Sodor 5-stage processor.

	$\#Regs$	$\#Ins$	$\#Exc$	$Time\ (m:s)$	$Diam-eter$	$States\ Found$
simple	5	200	1024	00:09	832	125999
sodor_subset	11			00:10		
sodor_complete	37			00:13		
simple	5	200	16384	04:43	832	3200254
sodor_subset	11			05:50		
sodor_complete	37			07:45		
simple	5	200	262144	15:36	832	7614589
sodor_subset	11			17:30		
sodor_complete	37			22:40		
simple	5	200	4194304	35:49	832	16921893
sodor_subset	11			41:27		
sodor_complete	37			45:15		

Table 7.8 – Experimental results for timing anomalies detection with multiples latencies for Rocket processor.

	$\#Regs$	$\#Ins$	$\#Exc$	$Time\ (m:s)$	$Diam-eter$	$States\ Found$
simple	6	200	1024	00:08	834	98927
rocket_subset	16			00:09		
rocket_complete	30			00:11		
simple	6	200	16384	02:49	834	3200343
rocket_subset	16			03:13		
rocket_complete	30			03:40		
simple	6	200	262144	19:15	834	7614726
rocket_subset	16			22:10		
rocket_complete	30			24:19		
simple	6	200	4194304	47:37	834	16922102
rocket_subset	16			53:36		
rocket_complete	30			58:13		

cessor designs with 5 stages, and 834 for Rocket processor with 6 stages as a pipeline depth. Finally, the state founds number depends on state space depth (traces) and width (variabilities) and increases with the pipeline stages. Indeed, it can reach 2K of difference between 3 and 5 pipeline stages.

Integrating our formal pipeline models into a procedure for the detection of timing anomalies has demonstrated the effectiveness of the automatic generation of pipeline models from the code of processor designs. We have man-

Table 7.9 – Experimental results for timing anomalies detection with various processor designs.

	$\#Regs$	$\#Ins$	$\#Execs$	$Time$ $(m:s)$	$Diam$ $-eter$	$States$ $Found$
RISC-V Mini	11	200	4194304	18:38	828	16710594
sodor 3stage	8			21:07	828	16710594
sodor 5stage	37			45:15	832	16921893
Kyogen	74			53:18	832	16921893
Rocket	30			58:13	834	16922102
Fuxi	53			48:05	832	16921893

aged to demonstrate this integration by proving the absence/presence of timing anomalies on all our use case design processors. The absence is obviously proved with respect to the considered traces. Furthermore, we can produce reliable coverage for each processor's formal model with various benchmarks featuring variable latencies and multiple traces. The code coverage relies on the coverage of all the execution patterns provided by the Instruction Set Architecture (ISA) abstraction as control conditions specified by hand in the corresponding ISA module. The code coverage is achieved at runtime when all the conditions (i.e., bypass and control conditions) are tested and evaluated through relevant benchmarks. For instance, with a trace size of 200 instructions and 4M execution paths due to latency variation, we can reach the coverage of all the instruction classes and pipeline status.

As observed, formal verification of properties on the provided models shows the impact of the automatic generation on the runtime and the state space size. The TLC model checker needs more time and faces the state space explosion proportionally with the complexity of the processor designs in terms of its number of pipeline stages, the pipeline structure, and the number of registers. Furthermore, the scalability of the automatic models remains reasonable compared to the simple ones.

7.4 . Conclusion

The timing analysis of safety-critical systems relies on hardware models (e.g., caches, pipelines, etc.); thus, generating pipeline models, wherever possible, is necessary. We have presented in this chapter the integration of such pipeline models into an existing model checking-based procedure for the detection of timing anomalies. Our proposed approach is based on a semi-automatic generation of pipeline datapath models from processor design code, which is then used to produce formal pipeline models. Then, we manually add to the datapath pipeline model the control logic (stalling), forming complete pipeline models that could capture the execution of instructions through a pipeline.

We have presented first the algorithms of the automatic generation of datapath formal models in TLA+. Then, we have illustrated the approach to various RISC-V processor designs. Thereafter, we experimented with their TLA+ formal specifications, integrated into a procedure for the detection of timing anomalies. We have reported preliminary experimental results in which we detailed the impact of the semi-automatic generation of pipeline models on the running time and state space explosion through several synthetic benchmarks.

8 - Conclusion and Perspectives

In this thesis work, we have presented our general workflow that addresses the formal verification of timing properties with the goal of automatic generation of formal pipeline models from high-level hardware designs. We have proposed an approach to automatically generate the pipeline datapath models for WCET analysis. It addresses the hardware design developed with Chisel language and integrated into the Chisel/FIRRTL hardware compilation framework. The approach considers as a starting point the datapath pipeline modules and relies on how the design is coded. The datapath pipeline modules can be an individual module defined as a mono-module pipeline design or is developed over several software-level modules as the multi-module pipeline, connected through interfaces and using various internal structures to define the pipeline circuitry. The analysis is structured into two main phases; the register analysis and then the construction of the abstract datapath pipeline model phases. Register analysis constructs an intermediate representation graph of the datapath pipeline as a dependency registers relation. The analysis is performed through two kinds of relations, the intra-module analysis and the inter-module analysis. The intra-module relations are performed at each module level for the register context. In contrast, the inter-module relations construct the input/output interfaces between pipeline modules in order to build the register connections. The second phase builds the abstract datapath pipeline model through a register-assignment procedure.

We have shown how a high-level hardware compilation toolchain can ease the adaptation of automatically generated formal abstract models to such safety properties. We have reported on a custom pass to generate an abstract pipeline model to be used therefore for the detection of timing anomalies. We have evaluated our approach on several in-order RISC-V processors, from the less to more complex range of processors, mono-module and multi-module of datapath pipeline. We have demonstrated the efficiency of our approach with various experiments that summarize the results and construct the synthesis.

We have also presented the automatic generation of formal models for the verification of timing predictability-related properties. These formal models are based on the generated datapath pipeline models in which we add control logic manually. We have translated the models into formal specifications using TLA+ language. This specification is further integrated into an existing procedure for the detection of timing anomalies. We experimented with TLA+ formal specification using the TLC model checking technique to explore the TLA+ specification while verifying the required timing properties. We have reported the experimental results in which we detailed the impact of the

semi-automatic generation of pipeline models on the running time and state explosion through several synthetic benchmarks.

As future work, we aim to validate the results of our algorithm that constructs the datapath pipeline models against processors so as to replace the current manual validation of these models with an automated procedure. Our model is a subgraph of the original pipeline processor. The objective is to validate the models against the original processor through the processor execution of assumed programs. Thus, we combine our static construction with a dynamic analysis through program execution. Several techniques can be used in order to establish this dynamic aspect. We aim to test the design and simulate the execution traces. Therefore, the reliability of our model can be determined by proving that such a subsumption relation between our model subgraph and the results of the processor design execution exists. We compare the register relations of the original processor execution traces with the abstract model relations.

We also intend to evaluate our approach on more complex processors as out-of-order RISC-V designs. Out-of-order processors add more complex pipeline constructions and mechanisms as structural hazards and speculative execution. As such, we need to include and treat all the Chisel language constructions as queues to correctly approximate the hardware designs' timing behavior. Furthermore, analyzing out-of-order processors allows us to add several design components as functional units on pipeline models. This feature enables extending the detection of timing anomalies analysis.

Publications

- **International Conferences**

1. Samira Ait Bensaid, Mihail Asavoae, Farhat Thabet, and Mathieu Jan, “Deriving pipeline models for timing analysis from high-level HDL processor designs”. In 20th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2022, Shanghai, China, October 13-14, 2022. IEEE, 2022, pp. 1–8.

2. Samira Ait Bensaid, Mihail Asavoae, Farhat Thabet, and Mathieu Jan, “Work in progress : Automatic construction of pipeline datapaths from high-level HDL code”. In 28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2022, Milano, Italy, May 4-6, 2022. IEEE, 2022, pp. 305–308.

3. Benjamin Binder, Samira Ait Bensaid, Simon Tollec, Mihail Asavoae, Farhat Thabet, and Mathieu Jan, “Formal processor modeling for analyzing safety and security properties”. In 11 th European Congress Embedded Real-Time Systems, ERTS 2022, Toulouse-France, June 1-2.

- **Workshops/Talks**

1. Samira Ait Bensaid, Mihail Asavoae, Farhat Thabet, and Mathieu Jan, “Pipeline datapath models from risc-v based cores”. In RISC-V Spring Week, 2022.

2. Samira Ait Bensaid, Mihail Asavoae, Farhat Thabet, and Mathieu Jan, “Pipeline datapath models from risc-v based cores”. In TLA+ Community Event Conference, 2023.

9 - Annexe: Résumé Substanciel

Les systèmes embarqués sont conçus avec des fonctionnalités spécifiques et intégrés dans des systèmes plus complexes. Les systèmes embarqués disposent d'une infrastructure matérielle (basée sur un processeur) et sont capables d'exécuter des applications qui répondent à ces fonctionnalités spécifiques. Ces systèmes sont utilisés dans divers contextes allant des appareils portables aux systèmes critiques. La validité de ces systèmes critiques dépend non seulement de leur correction fonctionnelle, mais aussi du temps dans lequel ils sont exécutés. La conception de ces systèmes repose sur des réglementations et des normes pour aider à identifier et à traiter les événements dangereux potentiels. Par exemple, des événements tels que le non-respect des délais seraient jugés inacceptables pour ces systèmes.

Les analyses statiques de pire temps d'exécution sont parmi les approches utilisées pour garantir les délais requis pour les systèmes critiques.
Afin d'estimer des bornes précises sur ces temps d'exécution, ces analyses nécessitent de modéliser très finement le comportement temporel de la microarchitecture du ou des processeurs utilisés dans la plateforme matérielle. Ainsi, ces limites sont exprimées en cycles de processeur. Ces modèles de micro-architecture sont aujourd'hui développés de manière manuelle, à partir d'éléments de documentation des processeurs qui peuvent être face à des erreurs. Cependant, avec l'émergence du design des processeurs open-source, dont les descriptions au niveau RTL sont disponibles, et les langages de description de matériel de haut niveau (HCLs), la génération automatique de ces modèles de micro-architecture et, plus spécifiquement, des modèles de pipeline permet d'envisager une automatisation de cette étape.

Nous proposons un workflow pour la construction de modèles de chemins de données de pipeline à partir de conceptions de processeurs décrites dans des langages de construction de matériel (HCLs). Nous intéressons aux conceptions exprimées avec le langage Chisel qui est supporté par la chaîne de compilation matériel Chisel/FIRRTL. La conception exprimée en Chisel est compilée par la suite en une représentation intermédiaire FIRRTL par le biais des transformations dont cette infrastructure dispose. Ces chaines de compilation facilitent l'intégration des analyses des conceptions matérielles. Notre workflow est basé sur cette chaîne de compilation Chisel/FIRRTL. Nous déployons des analyses qui visent à construire automatiquement au niveau de la représentation intermédiaire FIRRTL des modèles de pipeline du chemin de données.

L'approche prend donc comme un point d'entrée les modules de pipeline du chemins de données et s'appuie sur la manière dont la conception est codée. Nous visons par la suite l'utilisation de ces modèles pour des pro-

priétés temporelles. Nous nous intéressons donc à l'aspect temporel au niveau du pipeline pour construire ces modèles. De ce fait, on construit le modèle de chemin de données de pipeline autour des registres. Nous présentons les résultats préliminaires de l'application sur plusieurs processeurs RISC-V open-source. Ces conceptions des processeurs sont caractérisées par leur diversité au niveau de nombre d'étages de pipeline et leur complexité de la structure du chemin de données du pipeline.

Nos travaux de thèse vise à appliquer ces modèles générés pour prouver des propriétés temporelles, notamment des propriétés liées à la prédictibilité temporelle. Pour prouver ces propriétés, diverses méthodes peuvent être utilisées. Nous adoptons la vérification formelle qui est une méthode exhaustive qui garantit que des résultats corrects sont produits dans toutes les conditions d'exécution possibles, et qui nécessite la construction de modèles formels. Les modèles générés sont suffisamment complets et précis pour être utilisé comme entrée pour la partie formelle. Nous les traduisons donc en modèles formels. Nous illustrons par la suite cette génération sur différents modèles de processeurs RISC-V.

Par ailleurs, nous les intégrons dans une procédure existante de détection des anomalies temporelle comme cas d'utilisation. Cette procédure est basée sur la vérification formelle. Nous employant le langage de modélisation et de vérification TLA+ et le model checking, et expérimentons notre analyse avec plusieurs processeurs RISC-V open-source. Nous évaluons cette intégration pour plusieurs processeurs comme cas d'étude. Nous examinons également l'impact de la génération automatique de modèles par le biais d'une série de critères synthétiques.

Bibliography

[1] "Risc-v sodor," https://github.com/ucb-bar/riscv-sodor,.

[2] "Rocket chip," https://github.com/chipsalliance/rocket-chip,.

[3] M. Schlickling and M. Pister, "A framework for static analysis of VHDL code," in *WCET*, ser. OASICS, vol. 6, 2007.

[4] ——, "Semi-automatic derivation of timing models for WCET analysis," in *LCTES*. ACM, 2010, pp. 67–76.

[5] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, "The program dependence web: A representation supporting control, data, and demand-driven interpretation of imperative languages," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, B. N. Fischer, Ed. ACM, 1990, pp. 257–271.

[6] M. Asavoae, B. B. Hedia, and M. Jan, "Formal executable models for automatic detection of timing anomalies," in *WCET 2018*, vol. 63, 2018, pp. 2:1–2:13.

[7] D. Kaestner and C. Ferdinand, "Safety standards and wcet analysis tools," in *ERTS*, 2012.

[8] T. Lothar and W. Reinhard, " Design for Timing Predictability - Real-Time Systems," 2004.

[9] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: an open toolbox for adaptive WCET analysis," in *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, ser. Lecture Notes in Computer Science, S. L. Min, R. G. P. IV, P. P. Puschner, and T. Ungerer, Eds., vol. 6399. Springer, 2010, pp. 35–46.

[10] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*, ser. OASIcs, J. Reineke, Ed., vol. 57. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 8:1–8:12.

[11] X. Li, L. Yun, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 56–67, 2007.

[12] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Trans. Embed. Syst.*, vol. 3, no. 1, pp. 05:1–05:48, 2016.

[13] J. Engblom and B. Jonsson, "Processor pipelines and their properties for static WCET analysis," in *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, ser. Lecture Notes in Computer Science, A. L. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491. Springer, 2002, pp. 334–348.

[14] M. Langenbach, S. Thesing, and R. Heckmann, "Pipeline modeling for timing analysis," in *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, ser. Lecture Notes in Computer Science, M. V. Hermenegildo and G. Puebla, Eds., vol. 2477. Springer, 2002, pp. 294–309.

[15] R. Wilhelm, M. Pister, G. Gebhard, and D. Kästner, "Testing implementation soundness of a WCET analysis tool," in *A Journey of Embedded and Cyber-Physical Systems*. Springer, 2021, pp. 5–17.

[16] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[17] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," Tech. Rep. UCB/EECS-2014-146, Aug 2014.

[18] J. Bachrach, H. Vo, B. C. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in *DAC'12*, 2012, pp. 1216–1225.

[19] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC*, 2012, p. 1216–1225.

[20] C. Papon, "SpinalHDL," https://github.com/SpinalHDL.

[21] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: a multi-level intermediate representation for hardware description languages," in *PLDI*, A. F. Donaldson and E. Torlak, Eds., 2020, pp. 258–271.

[22] B. Binder, M. Asavoae, F. Brandner, B. B. Hedia, and M. Jan, "The role of causality in a formal definition of timing anomalies," in *RTCSA 2022*, 2022, pp. 91–102.

[23] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *ICCAD*, 2017, p. 209–216.

[24] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of bluespec: A core language for rule-based hardware design." New York, NY, USA: Association for Computing Machinery, 2020.

[25] O. Port and Y. Etsion, "Dfiant: A dataflow hardware description language," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[26] "chisel-bootcamp," https://github.com/freechipsproject/chisel-bootcamp.

[27] P. Bernardi, R. Cantoro, L. M. C. Brasca, B. Du, E. Sánchez, M. S. Reorda, M. Grosso, and O. Ballan, "On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors," in *14th International Workshop on Microprocessor Test and Verification, MTV 2013, Austin, TX, USA, December 11-13, 2013*. IEEE Computer Society, 2013, pp. 52–57.

[28] "Risc-v mini," https://github.com/ucb-bar/riscv-mini,.

[29] A. Saitoh, "Kyogenrv: simple 5-staged pipeline RISC-V," https://github.com/panda5mt/KyogenRV.

[30] "Fuxi, a 32-bit pipelined risc-v processor written in chisel3." https://github.com/MaxXSoft/Fuxi,.

[31] A. Waterman, Y. Lee, R. Avizienis, H. Cook, D. A. Patterson, and K. Asanovic, "The RISC-V instruction set," in *2013 IEEE Hot Chips 25 Symposium (HCS), 2013*. IEEE, 2013, p. 1.

[32] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Wiley, 1987.

[33] E. M. Clarke, T. Filkorn, and S. Jha, "Exploiting symmetry in temporal logic model checking," in *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, ser. Lecture Notes in Computer Science, C. Courcoubetis, Ed., vol. 697. Springer, 1993, pp. 450–462.

[34] Y. Yu, P. Manolios, and L. Lamport, "Model checking tla$^+$ specifications," in *CHARME*, vol. 1703, 1999, pp. 54–66.

[35] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, 1994.

[36] ——, *The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[37] G. J. Holzmann, *Explicit-State Model Checking*, 2018.

[38] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 1020 states and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[39] K. L. McMillan, "Symbolic model checking: an approach to the state explosion problem," 1992.

[40] C. Eisner and D. A. Peled, "Comparing symbolic and explicit model checking of a software system," in *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, 2002, Proceedings*, ser. Lecture Notes in Computer Science, D. Bosnacki and S. Leue, Eds., vol. 2318. Springer, 2002, pp. 230–239.

[41] H. R. Andersen, “An introduction to binary decision diagrams,” 1997.

[42] L. Charvát, A. Smrcka, and T. Vojnar, “HADES: microprocessor hazard analysis via formal verification of parameterized systems,” in *MEMICS*, ser. EPTCS, vol. 233, 2016, pp. 87–93.

[43] A. Mir, S. Balakrishnan, and S. Tahar, “Modeling and verification of embedded systems using cadence smv,” in *2000 Canadian Conference on Electrical and Computer Engineering. Conference Proceedings. Navigating to a New Era (Cat. No.00TH8492)*, 2000.

[44] R. Mukherjee, D. Kroening, and T. Melham, “Hardware verification using software analyzers,” in *2015 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2015, Montpellier, France, July 8-10, 2015*. IEEE Computer Society, 2015, pp. 7–12.

[45] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann, “New developments in WCET analysis,” in *Program Analysis and Compilation, Theory and Practice, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, ser. Lecture Notes in Computer Science, T. W. Reps, M. Sagiv, and J. Bauer, Eds., vol. 4444. Springer, 2006, pp. 12–52.

[46] Z. Bai, H. Cassé, M. D. Michiel, T. Carle, and C. Rochange, “Improving the performance of WCET analysis in the presence of variable latencies,” in *Proceedings of the 21st ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2020, London, UK, June 16, 2020*, J. Xue and C. Jung, Eds. ACM, 2020, pp. 119–130.

[47] X. Li, A. Roychoudhury, and T. Mitra, “Modeling out-of-order processors for WCET analysis,” *Real Time Syst.*, vol. 34, no. 3, pp. 195–227, 2006.

[48] D. Kroening and W. J. Paul, “Automated pipeline design,” in *DAC*. ACM, 2001, pp. 810–815.

[49] E. Nurvitadhi, J. C. Hoe, T. Kam, and S. Lu, “Automatic pipelining from transactional datapath specifications,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 30, no. 3, pp. 441–454, 2011.

[50] J. Gorius, S. Rokicki, and S. Derrien, “Spechls: Speculative accelerator design using high-level synthesis,” *IEEE Micro*, vol. 42, no. 5, pp. 99–107, 2022.

[51] K. McMillan, “Cadence smv,” Cadence Berkeley Labs, CA, 2000.

[52] D. Burger and T. M. Austin, “The simplescalar tool set, version 2.0,” 1997.

[53] A. Mycroft and R. Sharp, “Hardware synthesis using SAFL and application to processor design,” in *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME*

2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings, ser. Lecture Notes in Computer Science, T. Margaria and T. F. Melham, Eds., vol. 2144. Springer, 2001, pp. 13–39.

[54] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," in *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, M. Felleisen, P. Hudak, and C. Queinnec, Eds. ACM, 1998, pp. 174–184.

[55] M. J. C. Gordon, "The Semantic Challenge of Verilog HDL," in *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 1995, pp. 136–145.

[56] P. O. Meredith, M. Katelman, J. Meseguer, and G. Rosu, "A formal executable semantics of verilog," in *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*. IEEE Computer Society, 2010, pp. 179–188.

[57] M. J. C. Gordon, "Relating event and trace semantics of hardware description languages," *Comput. J.*, vol. 45, no. 1, pp. 27–36, 2002.

[58] D. J. Greaves, "Layering rtl, safl, handel-c and bluespec constructs on chisel hcl," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015, pp. 108–117.

[59] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 24:1–24:30, 2017.

[60] L. Truong and P. Hanrahan, "A golden age of hardware description languages: Applying programming language techniques to improve design productivity," in *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA*, ser. LIPIcs, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., vol. 136. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 7:1–7:21.

[61] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "C?ash: Structural descriptions of synchronous hardware using haskell," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010.

[62] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: a language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 296–311.

[63] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 804–817.

[64] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: a multi-level intermediate representation for hardware description languages," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 258–271.

[65] F. Heilmann, C. Brugger, and N. Wehn, "Investigate the high-level hdl chisel," 2013, p. 2.

[66] P. Pan, S. Jiang, Y. Ou, and C. Batten, "Towards gradually typed hardware description languages," *Rem*, vol. 504, p. 0, 2023.

[67] J. Im and S. Kang, "Comparative analysis between verilog and chisel in RISC-V core design and verification," in *18th International SoC Design Conference, ISOCC 2021, Jeju Island, South Korea, Republic of, October 6-9, 2021*. IEEE, 2021, pp. 59–60.

[68] P. Lennon and R. Gahan, "A comparative study of chisel for fpga design," in *2018 29th Irish Signals and Systems Conference (ISSC)*, 2018, pp. 1–6.

[69] J. Bruant, P. Horrein, O. Muller, T. Groléat, and F. Pétrot, "(system)verilog to chisel translation for faster hardware design," in *International Workshop on Rapid System Prototyping, RSP 2020, Hamburg, Germany, September 24-25, 2020*. IEEE, 2020, pp. 1–7.

[70] Ken Roe, "sifive/kami," https://github.com/sifive/Kami.

[71] I. Neamtiu, J. S. Foster, and M. W. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*. ACM, 2005.

[72] W. Yang, "Identifying syntactic differences between two programs," *Softw. Pract. Exp.*, vol. 21, no. 7, pp. 739–755, 1991.

[73] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," in *International Symposium on Programming, 6th Colloquium, Toulouse, France, April 17-19, 1984, Proceedings*, ser. Lecture Notes in Computer Science, M. Paul and B. J. Robinet, Eds., vol. 167. Springer, 1984, pp. 125–132.

[74] K. Kim and B. R. Moon, "Malware detection based on dependency graph using hybrid genetic algorithm," in *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010*, M. Pelikan and J. Branke, Eds. ACM, 2010, pp. 1211–1218.

[75] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing of hardware description languages," in *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, ser. Lecture Notes in Computer Science, L. Pierre and T. Kropf, Eds., vol. 1703. Springer, 1999, pp. 298–312.

[76] F. Tip, "A survey of program slicing techniques," *J. Program. Lang.*, vol. 3, no. 3, 1995.

[77] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival, "Static analysis by abstract interpretation of embedded critical software," *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, pp. 1–8, 2011.

[78] K. Y. Xavier Rival, *Introduction to Static Analysis An Abstract Interpretation Perspective*, 2019.

[79] S. Vasudevan, E. A. Emerson, and J. A. Abraham, "Improved verification of hardware designs through antecedent conditioned slicing," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 1, pp. 89–101, 2007.

[80] R. Sharp and A. Mycroft, "Soft scheduling for hardware," in *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, ser. Lecture Notes in Computer Science, P. Cousot, Ed., vol. 2126. Springer, 2001, pp. 57–72.

[81] V. Athavale, S. Ma, S. Hertz, and S. Vasudevan, "Code coverage of assertions using RTL source code analysis," in *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*. ACM, 2014, pp. 61:1–61:6.

[82] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: a language for hardware-level security policy enforcement," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, R. Balasubramonian, A. Davis, and S. V. Adve, Eds. ACM, 2014, pp. 97–112.

[83] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 109–120.

[84] F. E. Allen, "Control flow analysis," *Proceedings of a symposium on Compiler optimization*, 1970. [Online]. Available: https://api.semanticscholar.org/CorpusID:14740595

[85] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[86] P. Cousot, "Abstract interpretation of graphs," in *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems - Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday*, ser. Lecture Notes in Computer Science, P. López-García, J. P. Gallagher, and R. Giacobazzi, Eds., vol. 13160. Springer, 2023, pp. 72–96.

[87] A. Dalsgaard, M. Olesen, M. Toft, R. Hansen, and K. Larsen, "Metamoc: Modular execution time analysis using model checking," in *WCET*, vol. 15, 2010, pp. 113–123.

[88] A. Mangean, J. Béchennec, M. Briday, and S. Faucou, "WCET analysis by model checking for a processor with dynamic branch prediction," in *Verification and Evaluation of Computer and Communication Systems - 11th International Conference, VECoS 2017, Montreal, QC, Canada, August 24-25, 2017, Proceedings*, ser. Lecture Notes in Computer Science, K. Barkaoui, H. Boucheneb, A. Mili, and S. Tahar, Eds., vol. 10466. Springer, 2017, pp. 64–78.

[89] A. Metzner, "Why model checking can improve WCET analysis," in *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, ser. Lecture Notes in Computer Science, R. Alur and D. A. Peled, Eds., vol. 3114. Springer, 2004, pp. 334–347.

[90] R. Wilhelm, "Why AI + ILP is good for wcet, but MC is not, nor ILP alone," in *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 309–322.

[91] A. Gustavsson, A. Ermedahl, B. Lisper, and P. Pettersson, "Towards WCET analysis of multicore architectures using UPPAAL," in *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, ser. OASIcs, B. Lisper, Ed., vol. 15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010, pp. 101–112.

[92] V. Touzeau, C. Maïza, D. Monniaux, and J. Reineke, "Ascertaining uncertainty for efficient exact cache analysis," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10427. Springer, 2017, pp. 22–40.

[93] B. Rieder, P. P. Puschner, and I. Wenzel, "Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis," in *International Workshop on Intelligent So-*

lutions in Embedded Systems, WISES 2008, Regensburg, Germany, July 10-11, 2008, M. Kucera, R. Roth, and M. Conti, Eds. IEEE, 2008, pp. 1–7.

[94] B. Blackham and G. Heiser, "Sequoll: A framework for model checking binaries," in *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. IEEE Computer Society, 2013, pp. 97–106.

[95] M. Asavoae, I. Haur, M. Jan, B. B. Hedia, and M. Schoeberl, "Towards formal co-validation of hardware and software timing models of cpss," in *Cyber Physical Systems. Model-Based Design - 9th International Workshop, CyPhy 2019, and 15th International Workshop, WESE 2019, New York City, NY, USA, October 17-18, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, R. D. Chamberlain, M. E. Grimheden, and W. Taha, Eds., vol. 11971. Springer, 2019, pp. 203–227.

[96] M. Jan, M. Asavoae, M. Schoeberl, and E. A. Lee, "Formal semantics of predictable pipelines: a comparative study," in *ASP-DAC*, 2020.

[97] B. Binder, M. Asavoae, F. Brandner, B. B. Hedia, and M. Jan, "Scalable detection of amplification timing anomalies for the superscalar tricore architecture," in *FMICS*, 2020.

[98] M. Asavoae, B. B. Hedia, and M. Jan, "Formal modeling and verification for timing predictability," in *ERTS 2020*, 2020.

[99] B. Binder, M. Asavoae, B. B. Hedia, F. Brandner, and M. Jan, "Is this still normal? putting definitions of timing anomalies to the test," in *27th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2021, Houston, TX, USA, August 18-20, 2021*. IEEE, 2021, pp. 139–148.

[100] B. Binder, M. Asavoae, F. Brandner, B. B. Hedia, and M. Jan, "The role of causality in a formal definition of timing anomalies," in *28th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2022, Taipei, Taiwan, August 23-25, 2022*. IEEE, 2022, pp. 91–102.

[101] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen WCET benchmarks: Past, present and future," in *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, ser. OASIcs, B. Lisper, Ed., vol. 15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010, pp. 136–146.

[102] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *Proceedings of the 20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA, December 1-3, 1999*. IEEE Computer Society, 1999, pp. 12–21.

[103] I. Wenzel, R. Kirner, P. P. Puschner, and B. Rieder, "Principles of timing anomalies in superscalar processors," in *Fifth International Conference on*

Quality Software (QSIC 2005), 19-20 September 2005, Melbourne, Australia. IEEE Computer Society, 2005, pp. 295–306.

[104] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *WCET*, vol. 4, 2006.

[105] A. M. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *ICCAD*, 2017, pp. 209–216.

[106] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *RTSS*. IEEE Computer Society, 1999, pp. 12–21.

[107] L. Kosmidis, D. Compagnin, D. Morales, E. Mezzetti, E. Quiñones, J. Abella, T. Vardanega, and F. J. Cazorla, "Measurement-based timing analysis of the AURIX caches," in *WCET*, ser. OASIcs, vol. 55, 2016, pp. 9:1–9:11.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Sémantique Formelle d'une Infrastructure de Compilation Matériel
Mots clés: Modélisation des processeurs, Langages de Construction de Matériel (HCLs), Chemin de données du pipeline, Anomalies temporelles, Vérification de modèle

Résumé:

Les analyses statiques de pire temps d'exécution sont utilisées pour garantir les délais requis pour les systèmes critiques. Afin d'estimer des bornes précises sur ces temps d'exécution, ces analyses temporelles nécessitent des considérations sur la (micro)-architecture. Habituellement, ces modèles de micro-architecture sont construits à la main à partir des manuels des processeurs. Cependant, les initiatives du matériel libre et les langages de description de matériel de haut niveau (HCLs), permettent de réaborder la problématique de la génération automatique de ces modèles de micro-architecture, et plus spécifiquement des modèles de pipeline. Nous proposons un workflow qui vise à construire automatiquement des modèles de chemin de données de pipeline à partir de conceptions de processeurs décrites dans des langages de contruction de matériel (HCLs). Notre workflow est basé sur la chaine de compilation matériel Chisel/FIRRTL. Nous construisons au niveau de la représentation intermédiaire les modèles de pipeline du chemin de données. Notre travail vise à appliquer ces modèles pour prouver des propriétés liées à la prédictibilité temporelle. Notre méthode repose sur la vérification formelle. Les modèles générés sont ensuite traduits en modèles formels et intégrés dans une procédure existante basée sur la vérification de modèles pour détecter les anomalies de temps. Nous utilisons le langage de modélisation et de vérification TLA+ et expérimentons notre analyse avec plusieurs processeurs RISC-V open-source. Enfin, nous faisons progresser les études en évaluant l'impact de la génération automatique à l'aide d'une série de critères synthétiques.

Title: Formal Semantics of Hardware Compilation Framework
Keywords: Modeling of processor design, Hardware Construction Languages (HCLs), Pipeline datapath, Timing anomalies, Model checking

Abstract: Static worst-case timing analyses are used to ensure the timing deadlines required for safety-critical systems. In order to derive accurate bounds, these timing analyses require precise (micro-)architecture considerations. Usually, such micro-architecture models are constructed by hand from processor manuals. However, with the open-source hardware initiatives and high-level Hardware Description Languages (HCLs), the automatic generation of these micro-architecture models and, more specifically, the pipeline models are promoted. We propose a workflow that aims to automatically construct pipeline datapath models from processor designs described in HCLs. Our workflow is based on the Chisel/FIRRTL Hardware Compiler Framework. We build at the intermediate representation level the datapath pipeline models. Our work intends to prove the timing properties, such as the timing predictability-related properties. We rely on the formal verification as our method. The generated models are then translated into formal models and integrated into an existing model checking-based procedure for detecting timing anomalies. We use TLA+ modeling and verification language and experiment with our analysis with several open-source RISC-V processors. Finally, we advance the studies by evaluating the impact of automatic generation through a series of synthetic benchmarks.