Deriving Pipeline Models for Timing Analysis from High-Level HDL Processor Designs

Samira Ait Bensaid, Mihail Asavoae, Farhat Thabet and Mathieu Jan
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

***Abstract*—Static worst-case timing analysis is important in the context of safety-critical systems as it is one approach that could be used to validate the required timing bounds. In order to derive accurate bounds, the worst-case timing analysis is performed under (micro)-architecture consideration, consequently, these bounds are expressed in processor cycles. The required (micro)-architecture models are usually constructed by hand, from processor manuals and validated through testing. Recent advances in hardware design promote open hardware initiatives and high-level Hardware Description Languages (HDLs), revisiting the perspectives to automatically construct (micro)-architecture models for worst-case timing analysis. In this paper, we present an approach concerning the construction of pipeline datapath models from processor designs described in high-level HDLs. We propose a methodology based on the Chisel/FIRRTL Hardware Compiler Framework which we apply on several open-source RISC-V processors.**

***Index Terms*—processor design, WCET analysis, pipeline datapath, HDL languages.**

I. Introduction

Design and implementation of safety-critical systems is standardized in order to identify and address potential hazardous events [1]. For example, events like missing timing deadlines would be deemed as unacceptable since these deadlines are mandatory in safety-critical systems. Thus, 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. Static worst-case execution time (WCET) analyses are able to compute safe and precise timing bounds while reasoning about the executions of an input program on an underlying architecture. In this setting, both aspects, that of the program and the architecture, become equally important. Whereas the program-level infrastructure required by the WCET analysis consists of a standard control-flow graph (i.e. the input program is the binary), at architecture-level, the analysis infrastructure is more diverse. For example, the same control-flow graph is used for cache analyses [2] whereas more specialized, cycle-accurate models are necessary for pipeline analyses [3].

There are several state of the art static timing analyzers: the industrial-strength tool aiT [4] and the academic ones Otawa [5], Heptane [6] and Chronos [7]. All these propose (micro)-architecture models of caches and pipelines, relying on static analysis to characterize their timing behaviors. These (micro)-architecture models are usually developed by hand and sometimes validated against hardware simulators for conformance (e.g. for aiT [8] or Chronos [9]). A closer code inspection of their pipeline models expose 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. Second, these models focus on how an instruction progresses through the pipeline and not its actual semantics (i.e. the correctness of program execution is assumed in the context of static WCET analysis). Third, these models are designed to “execute” basic blocks (i.e. straight-line code) and hence, certain pipeline circuitry, e.g. forwarding is not explicitly encoded in the pipeline model but handled at the code-level through arbitrary timing values. So, developing such models at a proper abstraction level is essential to obtain correct and precise WCET bounds.

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 [10] or SpinalHDL [11]), sitting at the top of specialized hardware compilation frameworks (e.g. FIRRTL [12]). 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.

In this paper, we propose an approach to construct pipeline models directly from RISC-V processor code developed within the Chisel/FIRRTL compilation framework. These pipeline models should adhere to all (previously mentioned) design characteristics required by a standard WCET analyzer. Since we aim to apply our analysis on any freely available, Chisel-based processor design, we also need to consider how the processor is coded as the accuracy of the constructed pipeline model depends on that. High-level HDLs, like Chisel promote design modularity and reusability, impacting the way a pipeline is coded. For example, a pipeline could be developed over several software-level modules connected through interfaces and/or using various kinds of internal structures to define the pipeline circuitry. Hence, a pipeline construction analysis should be able to handle all these design language issues.

In order to construct a datapath model out of high-level HDL code, our approach determines the pipeline depth (i.e. the number of stages) and identify how to connect these stages. As such, our approach is designed around the pipeline registers, as these are mapped to pipeline stages, similarly to the ‘flat‘ pipeline models of the WCET analyzers. For each software-level module of the pipeline design, our approach (1)

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

extracts the registers and their intra-module relations and (2) constructs input-output interfaces (i.e. through ports) for the inter-module register connections. Our approach relies on a register-to-pipeline stage assignment function to determine the datapath model. In this way, our approach exposes the register forwarding which, from a WCET analysis point of view, creates a separation of concerns between (micro)-architecture and program models. As such, our datapath model would be suited for a wider range of timing properties than those related to worst-case timing (e.g. timing anomalies or timing leakage). We apply our approach on several open-source RISC-V processors of various degree of complexity, i.e. from single to multi-modular pipeline designs. A work-in-progress version of this approach, limited to single module pipeline designs is presented in [13].

This paper is organized as follows. Section II presents elements of the Chisel/FIRRTL Hardware Compiler Framework and introduces non-trivial case studies. Section III presents the approach towards automatically constructing abstract pipeline models for the WCET analysis. Section IV reports on the implementation and experimental results. Section V and Section VI address the related work and the conclusions, respectively.

II. Chisel/FIRRTL Framework and Illustration on RISC-V Cores

A. Hardware Compilation Framework: Chisel/FIRRTL

Chisel [10], is an open-source hardware construction language embedded in Scala programming 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. We briefly introduce in this section some features of Chisel with applications in hardware designs of RISC-V processors. We focus on those features which are particularly important to our pipeline datapath construction approach.

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 [14] is defined as follows:

```
class DatPath(width: Int) extends Module {
   val io = IO (new DpathIo ())
   val if_reg_pc = RegInit (io.reset_vector)
}

val d = Module(new DatPath())
d.io.reset_vector := io.reset_vector
```

The variable `if_reg_pc` is a register declared inside the module `DatPath` and initialized with a value `io.reset_vector` while `d` is an instance of `DatPath` which is further used to access and update this particular value.

Ports and Data Types. 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` that can be used as an IO interface with ports defined by `Input/Output` Chisel language constructs. Chisel also provides `Bundle` which is a compound type to define collections of values with named fields.

```
class DpathIo(width : Int) extends Bundle {
   val reset_vector = Input(UInt())
   val dec_inst = Output(UInt())
}
```

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, while the operator <> is a bidirectional (i.e. bulk) connection element-wise. The `when` construction defines a conditional block.

```
class Rocket(width : Int) extends Bundle {
   val ibuf = Module(new IBuf())
   ibuf.io.imem <> io.imem.resp
   when (!ctrl_killd) {
     ex_reg_pc := ibuf.io.pc
   }
}
```

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

Chisel lies at the top of a hardware compilation framework as it compiles into an intermediate language named FIRRTL (Flexible Intermediate Representation for RTL) [12] and further, into Verilog code. The FIRRTL representation is internally represented with an Abstract Syntax Tree (AST) structure. The FIRRTL AST 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. FIRRTL proposes different representations, also named forms (i.e. high, low), with each form using a stricter and simpler subset of the Chisel language features and defining different transformations to generate the next (lower) form. The high form supports high-level constructs such as bundle type and conditional statements, while in the low form these are simplified and flatten in order to be translated into Verilog.

B. RISC-V Cores Based Chisel

We introduce next several hardware design features, using code snapshots of RISC-V processors designed using the Chisel/FIRRTL compilation framework. We exemplify them with the multi-modular and hierarchical pipeline design of the Rocket processor [15] and the flat and parametric pipeline construction of the Fuxi processor [16]. Finally, we present two different implementations of the forwarding mechanism, as developed in RISC-V Sodor [14] and Fuxi [16] processors.

Rocket Pipeline. Rocket is an in-order, scalar processor based on the RISC-V Instruction Set Architecture (ISA) [17]. 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 separated 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 Fig. 1. 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. Fig. 1 also presents the connections between the pipeline design elements (e.g. registers or input/output interfaces); the corresponding code snippets are in Listing 1 in Chisel and in Listing 2, in FIRRTL high form. From a code organization point of view, the pipeline is developed in five modules: `Rocket`, `IBuf`, `Frontend`, `ShiftQueue` and `RocketTile`.

Listing 1: Simplified Chisel code of Rocket datapath pipeline.

```
1  class RocketTile:
2    val core = Module(new Rocket)
3    frontend.module.io.cpu <> core.io.imem
4  ----------------
5  class Rocket:
6    val ex_reg_pc = Reg (size)
7    /*decls: mem_reg_pc and wb_reg_pc*/
8    val ibuf = Module (new IBuf)
9    when (cond1) {
10     ex_reg_pc := ibuf.io.pc
11     /*updates: mem_reg_pc and wb_reg_pc*/ }
12   ibuf.io.imem <> io.imem.resp
13 ----------------
14 class Frontend:
15   val io = IO(.../*cpu field*/)
16   /* decls: s1_pc and s2_pc */
17   s2_pc := s1_pc
18   val fq = Module (new ShiftQueue)
19   fq.io.enq.bits.pc := s2_pc
20   io.cpu.resp <> fq.io.deq
21 ----------------
22 class IBuf:
23   val io = new Bundle {/*imem and pc fields*/}
24   val buf = Reg (io.imem.bits)
25   buf.pc := io.imem.bits.pc
26   io.pc := Mux (cond2, buf.pc, io.imem.bits.pc)
27 ----------------
28 class ShiftQueue :
29   io.deq.bits := io.enq.bits
```

Listing 2: High-form FIRRTL for Listing 1.

```
1  module RocketTile:
2   inst frontend of Frontend
3   inst core of Rocket
4   core.io.imem.resp.bits.pc <=
5          frontend.io.cpu.resp.bits.pc
6  ---------------
7  module Rocket:
8   inst ibuf of IBuf
9   when cond1 :
10    ex_reg_pc <=ibuf.io.pc
11    /*updates: mem_reg_pc and wb_reg_pc*/
12  ibuf.io.imem.bits.pc <=
13         io.imem.resp.bits.pc
14  ibuf.io.imem.bits.data <=
15         io.imem.resp.bits.data
16 ---------------
17 module Frontend:
18  output io : {.../*cpu field*/}
19  s2_pc <=s1_pc
20  inst fq of ShiftQueue
21  fq.io.enq.bits.pc <=s2_pc
22  /* explicit update, field by field, of <>*/
23 ---------------
24 module IBuf:
25  output io : {/*imem and pc fields*/}
26  reg buf.pc : UInt<40>
27  buf.pc < -io.imem.bits.pc
28  node _io_pc_T_1 =
29     mux(cond2, buf.pc, io.imem.bits.pc)
30  io.pc <=_io_pc_T_1
31 ---------------
32 module ShiftQueue:
33  io.deq.bits.pc <=io.enq.bits.pc
```

Listing 1 illustrates 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 Fig. 1 and implemented using the bulk operator <> in Chisel. The connections of this operator which are described also in line 12 are compiled, in FIRRTL, into partial connections of each construction field, in lines 12-15 in Listing 2. 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 2.

The registers of decode and fetch 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 2. 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 2. 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 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. 1, the pipeline design of Fuxi is flatter, in Fig. 2. 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`.

Listing 3 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 3. The connection of two pipeline

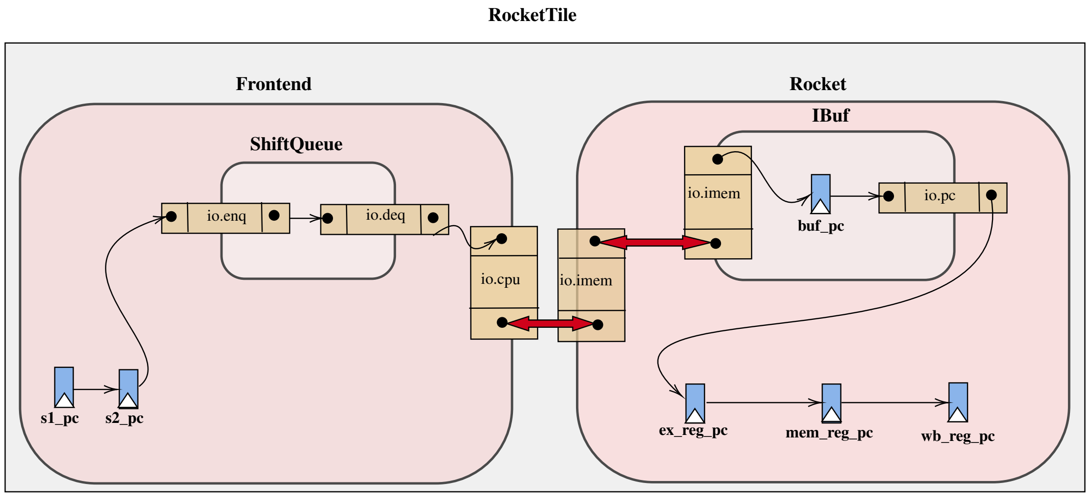

Fig. 1: Datapath pipeline modules of Rocket core.

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.

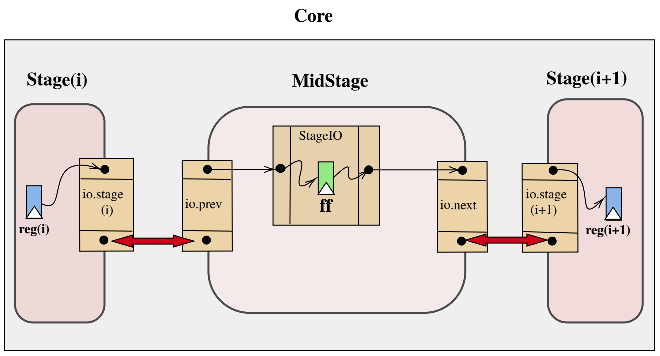

Fig. 2: Datapath pipeline modules of Fuxi.

Listing 3: 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, 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`.

Listing 4: 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 5. 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.

Listing 5: 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)
```

The code snippets in Listing 1 to Listing 5 present processor design choices addressed using 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.

III. Pipeline Datapath Analysis

In order to construct a pipeline datapath model for the WCET analysis, we need to determine the pipeline depth (i.e. number of pipeline stages) as well as the connections between these stages. A pipeline stage is to be identified by a set of registers. Intuitively, the construction of a pipeline datapath proceeds as follows: for each of the pipeline modules, given as input, and for each register in these modules, we perform a register analysis to determine precedence relations between registers. We also perform an output port analysis to determine how registers from different modules are to 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. Next, we introduce some notations and definitions, following a standard set-theoretic approach.

Notations. We consider $\mathcal{P}r = \bigcup_{i=1}^{n} M_i$, a processor design, defined by a set of n modules and $\mathcal{P} = \bigcup_{j=1}^{m} M_j$, 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 $\top$.

Example 1: The instance `ibuf` of the type module `IBuf`, defined in the module *Rocket*, as in Listing 2, line 8 is denoted by `(ibuf, IBuf)` $\in Insts_{Rocket}$. The module `RocketTile`, in Fig. 1 is the module $\top$ of $\mathcal{P}_{Rocket}$ whereas `Core`, in Fig. 2 is the module $\top$ of $\mathcal{P}_{Fuxi}$.

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.

Example 2: The ports `prev` and `next` of Listing 3, lines 14-15 are in $I/Os_{MidStage}$, the multiplexer `Mux(cond2, buf.pc, io.imem.bits.pc)` of Listing 1, line 26 is in $Combs_{IBuf}$, and registers `ex_reg_pc` and `buf.pc` of Listing 1, lines 6 and 24 are in $Regs_{Rocket}$ and $Regs_{IBuf}$ respectively.

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.

Example 3: The context `cond1` $\mapsto$ `ex_reg_pc` is in $Ctxs_{Rocket}$, as in Listing 1, lines 9-10. Similarly, `cond` $\mapsto$ `ff` is in $Ctxs_{MidStage}$, in Listing 3, lines 18-19. Finally, `C1` $\mapsto$ `read.data`, `C2` $\mapsto$ `read.data` etc. are in $Ctxs_{Fuxi}$, in Listing 5, lines 3-11.

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.

Example 4: Whereas the modules `RocketTile`, `Rocket`, `Frontend`, `IBuf` and `ShiftQueue` form the pipeline $\mathcal{P}$ of Rocket, in Listing 1 other modules like `ALU` or `CSR` (not represented) are in $Exts_{RocketTile}$.

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.

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 [12]. We name this operator

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

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

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 which is instantiated several times, e.g. `MidStage` of Fuxi, in Listing 3, our approach computes a module summary and uses it for all instances of this particular module. We denote by

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

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

Example 5: For the five pipeline modules of Rocket, shown in Fig. 1, `order_mods` produces the ordered set `ShiftQueue`, `Frontend`, `Ibuf`, `Rocket`, `RocketTile`.

Informally, a module summary has (1) an intra-modular component which focuses on the module's registers and (2) an inter-modular component which focuses on the module's interface (i.e. input-output ports).

We address the intra-modular 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 1: For a register $r \in Regs_M$, we denote by In_r the **input frontier** of r, defined by $\bigcup_{i=1}^{n} c_i$, 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 $\mathcal{P}$.

Definition 2: 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 1*.

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

Definition 3: For an output port $p \in I/Os_M$, we denote by Out_p the **output frontier** of p, defined by $\bigcup_{i=1}^{n} c_i$, where $c_i \in Regs_M$ or $c_i \in I/Os_M$.

An output frontier establishes, 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 4: 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 3*.

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 5: 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 register frontier of $r2$, and false otherwise. We denote by $Pred$ the set of $(r1, r2)$ with $r1, r2 \in Regs$, for which `prec`$(r1, r2)$ evaluates to true.

Definition 6: 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 2* and *Definition 4* respectively whereas I is the graph from *Definition 6*.

Definition 7: The operator $\texttt{ctx\_reg} : Regs \rightarrow 2^{Regs}$ is defined as $\texttt{ctx\_reg(r)} = R$ 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 $IR_{\mathcal{P}}$.

The abstract pipeline datapath of $\mathcal{P}$ is constructed by unfolding $IR_{\mathcal{P}}$ 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 1: Pipeline datapath construction

```
Input  : $\mathcal{P} = \bigcup_{i=1}^{m} \mathcal{M}p$ and $AST_{\mathcal{P}} = \bigcup_{i=1}^{m} AST_m$
Output: $Regs \mapsto Stages$ - pipeline datapath of $\mathcal{P}$
1 $o \leftarrow$ order_mods($\mathcal{P}$, $AST_p$) /* order modules
      based on instancing, $\top$ is last. */
2 foreach $m \in o$ do
3     ($I/Os_m$, $Regs_m$, $Ctxs_m$) $\leftarrow$
        process_mod($AST_m$)
4     $C_r \leftarrow C_r \cup$ regs_ctx($AST_m$, $Regs_m$, $I/Os_m$)
5     $C_{io} \leftarrow C_{io} \cup$ 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 1 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 last iteration, which is over the $\top$ module, is followed by the construction of the intermediate representation of $\mathcal{P}$, in line 6, used to generate the pipeline datapath, in line 7.

Algorithm 2 presents the assignment of registers to pipeline stages. This algorithm starts with the program counter PC assigned in the first pipeline stage, i.e., `to_stage`(`PC`) $= 1$. We distinguish two cases: case $\boxed{1}$ driven by register dependencies and case $\boxed{2}$ driven by context dependencies, $Ctxs$. We further define three possibilities for case $\boxed{1}$: the `linear_case` $\boxed{1}$, the `min_case` $\boxed{1}$ and the `max_case` $\boxed{1}$. 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.

Algorithm 2: Register to pipeline stage assignment

```
1  Function assign_regs($i$, $reg$, $IR_{\mathcal{P}}$, $Regs$):
       /* C1_* and C2 identify $r$ in $IR_{\mathcal{P}}$ */
2      if C1_linear then
3          to_stage(r) = $i + 1$ ;          /* linear case $\boxed{1}$ */
4          assign_regs ($i+1$, $r$, $IR_{\mathcal{P}}$, $Regs$)
5      else if C1_min then
6          to_stage(r) = $i + 1$ ;          /* min case $\boxed{1}$ */
7          assign_regs ($i+1$, $r$, $IR_{\mathcal{P}}$, $Regs$)
8      else if C1_max then
9          to_stage(r) = $i + 2$ ;          /* max case $\boxed{1}$ */
10         assign_regs ($i+2$, $r$, $IR_{\mathcal{P}}$, $Regs$)
11     else if C2 then
12         to_stage(r) = $i$ ;              /* case $\boxed{2}$ */
13         assign_regs ($i$, $r$, $IR_{\mathcal{P}}$, $Regs$)
14     else
15         $(i', r') \leftarrow$ select_reg($IR_{\mathcal{P}}$, $Regs$)
16         assign_regs ($i'$, $r'$, $IR_{\mathcal{P}}$, $Regs$)
17     return to_stage;
```

Linear_case $\boxed{1}$ (lines 2-4) is applied when $C1\_linear = \exists!(reg, r) \in Pred$ is true. 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.

Min_case $\boxed{1}$ (lines 5-7) considers the case when several source registers are already assigned. The stage assigned to r strictly follows the 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) with no precedence relation between this source register of minimal stage and the other source registers.

Formally, the `min_case` $\boxed{1}$ is applied to assign r, preceded by $regs$, when the condition $C1\_min$ is true, where $C1\_min = \exists(reg, r) \in Pred \wedge \texttt{to\_stage(reg)} = i - 1 \wedge \forall(r_2, r) \in Pred,\ r_2 \neq reg \wedge \texttt{to\_stage(r}_2\texttt{)} >= \texttt{to\_stage(reg)} \ \wedge \ \nexists(reg, r_2) \in Pred.$

The Fig.3 illustrates this case. 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.

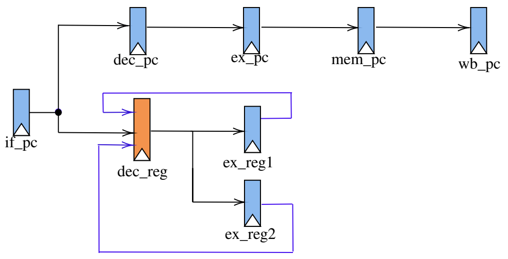

Fig. 3: Simple application of min case 1 ($C1\_min$).

Max_case $\boxed{1}$ (lines 8-10) also addresses the case of several source registers that are already assigned. It checks if the source register of 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` $\boxed{1}$ assigns r (preceded by reg) when $C1\_max$ is true, where $C1\_max = \exists(r_2, r) \in Pred \wedge \texttt{to\_stage(r}_2\texttt{)} = i \wedge \forall(reg, r) \in Pred \wedge r_2 \neq reg \wedge \ \texttt{to\_stage(r}_2\texttt{)} >= \texttt{to\_stage(reg)} \ \wedge \exists(reg, r_2) \in Pred.$

Fig. 4 illustrates this case, which is found in Rocket. 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).

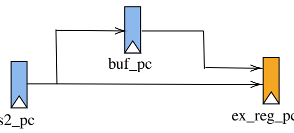

Fig. 4: Simple application of max case 1 ($C1\_max$).

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 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` preceeding `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: Simplified code for compressed instructions, in Rocket

```
class Rocket :
  val ibuf = Module (new IBuf)
  useCompressed: Boolean = true
  val fetchWidth: Int =
     if (useCompressed) then 2 else 1
  ex_reg_pc := ibuf.io.pc
-----------------
class IBuf
  val n = fetchWidth - 1
  val nBufValid =
     if (n == 0) then UInt(0) else Reg(fetchWidth)
  io.pc :=
     Mux(nBufValid > 0, buf.pc, io.imem.bits.pc)
```

Our approach 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` $\boxed{1}$. A complete construction of this model is presented in Section IV.

Case $\boxed{2}$ (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` $\boxed{2}$ is applied when $C2$ is true, where $C2 = \texttt{ctx\_reg(r)} = R \wedge \exists reg \in R \wedge \texttt{to\_stage(reg)} = i - 1.$

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.

IV. Implementation and Experiments

We implemented our approach in Scala, as a pass in the Chisel/FIRRTL hardware compiler framework and we evaluated its results on several open-source, Chisel-based RISC-V processors. Each processor design is considered *as is*, without manual modifications or 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 [18], Sodor [14], KyogenRV [19], Fuxi [16], and Rocket Chip [15]. We have briefly presented Sodor 5-stage, Fuxi and Rocket in Section II. RISC-V Mini [18] proposes a simple 3-stage pipeline designed to serve as an initial test case for the development of Chisel. We also consider a 3-stage pipeline of the Sodor processor as well as KyogenRV [19], a 5-stage pipelined processor targeting Intel FPGAs and developed for academic purposes. These processors are developed in different Chisel versions, from '3.2' for Fuxi to '3.5' for Rocket. We report the results of our analysis in Table I.

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 $\mathcal{P}$. Three of the processor designs are single-module and three are multi-module. 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 the column *Case 1* reports the aggregated results of these three cases and the 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 1), while the second corresponds to the register assignment phase (e.g., Algorithm 2). The runtime of the assignment phase is significantly lower than the one of the intra-modular phase. The latter increases with the number of modules, and is thus the highest for the Fuxi processor.

The depth of each pipeline has been correctly computed, namely our approach constructs a pipeline datapath of the same depth as in respective processor specification. Moreover, using the naming conventions w.r.t. register names and the pipeline stages (e.g., prefixes `dec_` and `ex_/exe_` for decode and execute stage respectively, as shown in Listing 1 or Listing 4), we could also verify that the registers are placed 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 Chisel `RegNext` construct. More specifically, the semantics of `RegNext` produces a one-cycle delayed version of the associated signal. 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. 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.

Our approach assigns all the registers of the pipeline modules only for two designs, Sodor 5-stage and RISC-V Mini. For KyogenRV, the unassigned registers are in fact not related to the datapath but to the control path; we also collected a vector register named `rv32i_reg[0-31]`, implementing the register file. For Rocket and Sodor 3-stage, 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, most of these registers are from the control path, updated in the external modules $Exts$, for instance the modules `CSR` or `BTB` of Rocket.

Next, we summarize in Fig. 5 the execution steps of the register assignment, i.e., Algorithm 2, for all the processor designs. The coding color of the algorithm traces is also displayed in Table I: green represents the *linear_case* $\boxed{\mathit{1}}$, gray and pink are *min_case* $\boxed{\mathit{1}}$ and *max_case* $\boxed{\mathit{1}}$ respectively, and orange represents *case* $\boxed{2}$. Furthermore, the multi-modular 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* $\boxed{\mathit{1}}$ in the beginning as it is driven by the register dependencies. We also notice that the transition between modules expose the way the implementation of the multi-modular 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. 2. 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. 1, where the registers are restricted in three modules `IBuf`, `Frontend` and `Rocket`. More precisely, the registers of the stages PC-generator and IF, are implemented in `Frontend` while the registers of stages from ID to WB are in `Rocket`.

The pipeline datapath construction of Rocket. Table II details the execution of the register assignment algorithm (i.e., Algorithm 2) on the Rocket processor. Since it is made of 240 registers, we only present a subset of the pipeline datapath registers. We start 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* $\boxed{\mathit{1}}$ 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

TABLE I: 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

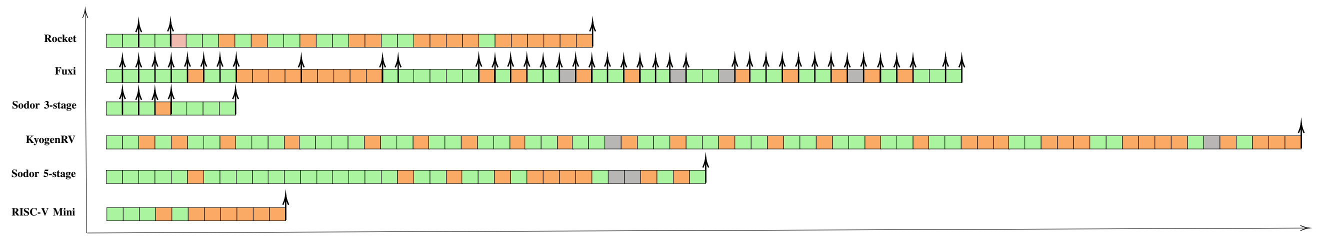

Fig. 5: Execution of the register assignment algorithm, i.e., Algorithm 2.

`max case` $\boxed{1}$ 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`.s Then, the *linear case* $\boxed{1}$ 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`. Their respective input frontier contains a reference to an external design element, the module `Inst`. So, these registers are assigned using case $\boxed{2}$ as they are updated in the same context as `ex_reg_pc`. Then, our algorithm applies the *linear_case* $\boxed{1}$ 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 $\boxed{2}$ is applied to register `mem_reg_wbdata`, which is connected to an external design element, the module `ALU` and which is updated in 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 $\boxed{2}$ 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 `RF`. So these registers are assigned using the case $\boxed{2}$, as they are updated in the same context as the registers already assigned to the execute stage. Fig. 6 shows the resulting pipeline datapath of Rocket, omitting those registers which are not on the path from the initial PC register.

TABLE II: 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` $\rightarrow$ `2`, `s2_pc` $\rightarrow$ `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

V. Related Work

The works in [20], [21] address a similar goal as ours, that of analyzing processor design code [20] and determining microarchitecture models for the WCET analysis [21]. These approaches analyze proprietary Verilog and VHDL code, whereas we consider open-source processor designs developed with more expressive HDL languages like Chisel/FIRRTL. In contrast to our approach, which engineers a solution and then aims to validate it a posteriori, the work in [20] uses the abstract interpretation framework to construct a sound approximation of the processor behavior. The results of [20] is applied in [21] to determine microarchitecture models using a semi-automatic procedure based on program slicing, while our analysis is fully automated towards constructing a pipeline datapath model. Both our approach and that of [21] rely on simplifications of the original design: ours is fed a given set of pipeline design modules while the work in [21] uses a combination of program slicing with hard-coded initialization. More precisely, program slicing is used to eliminate parameterized features of the design (i.e. for conditional compilation) and

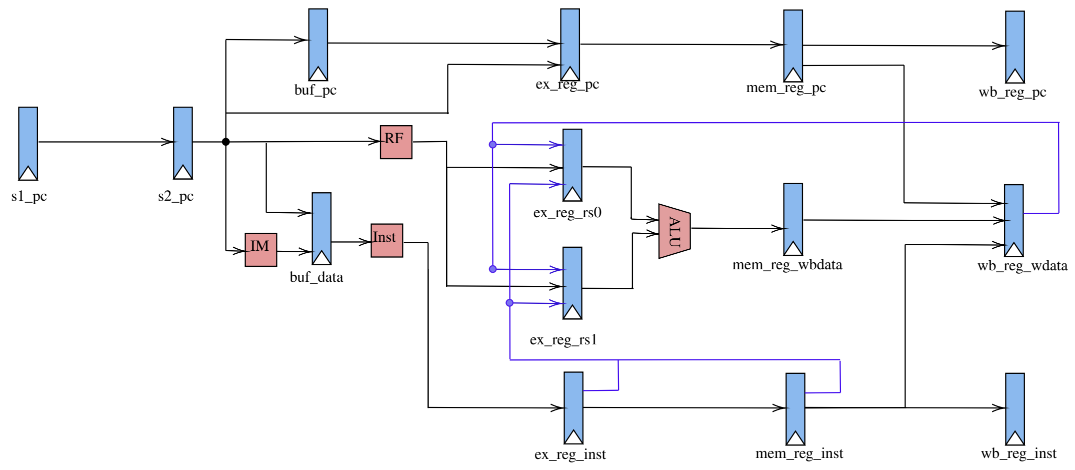

Fig. 6: Rocket core pipeline datapath model.

certain signals (e.g. for interrupts) are manually initialized. Regarding the control path, our approach does not differentiate between control and datapath registers as long as they are in the considered modules. The approach in [21] performs value analysis to approximate register and memory contents, aiming to simplify the original VHDL design (i.e. adding pseudo VHDL processes) to a datapath representation with accurate timing. The approach depends on the correct identification of the control path information, which is not fully automated. Abstract pipeline models (including forwarding) are addressed in [22], from processor graphs (i.e. structures combining combinational and sequential logics and which could be generated from Verilog/VHDL code). The actual datapath construction also relies on assigning registers to pipeline stages, but requires, as input, the pipeline depth. Besides, the work in [22] aims further, to formally verify the functional correctness of the pipeline optimizations.

A related problem is that of automated pipeline designs, addressed in [23], [24]. More specifically, their goal is to synthesize pipelines from sequential representations, starting from simple designs [23] to more complex variants [24]. In comparison, our approach has an opposite goal, that of going from full-fledged processor implementations to simpler pipeline datapath models, as used in the WCET analysis. Also, the automated pipeline design considers the pipeline depth whereas, our analysis aims to bound this value. Moreover, our analysis works on richer HDL languages used in processor designs. Finally, in [23], [24], the correctness of automated pipeline design is ensured during its construction, while we rely on techniques specific to software engineering to determine a solution.

Traditional HDLs like Verilog or VHDL are currently backends in complex hardware compiler frameworks like that of Chisel/FIRRTL or those in [25]–[27]. As such, developing automated techniques towards analyzing HDL designs (e.g. for the construction of datapath pipelines, as in the current work) becomes tantamount to work with both high-level HDL languages and language transformation (i.e. compilation) issues. For example, the work in [27] proposes an intermediate-language representation named LLHD, which is inspired from LLVM IR, thus, in the SSA form. LLHD plays a similar role as FIRRTL in a hardware compilation chain, to facilitate the integration of compilation passes. Our approach works on the AST of FIRRTL, where multiple updates of the same register are possible, whereas LLHD proposes a more analysis-friendly representation (i.e. SSA) with a single register update. Higher-level HDLs [25], [26] similar to Chisel also propose design modularity and abstraction, leveraging the functional programming paradigm. One of their common characteristics is to explicitly express guarded (register) updates, identifiable on the AST of the processor design and used in our analysis to determine contexts for the register updates. Our approach, which targets the Chisel/FIRRTL compilation framework, would also be extensible to other similar frameworks, as shown by respective language inclusions, in [25], for Chisel and in [27] for FIRRTL.

VI. Conclusions and Future Work

In this paper we proposed an approach to automatically generate pipeline datapath models for the WCET analysis, out of RISC-V processor designs. Our approach addressed processors developed with the Chisel/FIRRTL compilation framework and we applied it on several open-source processors of various degree of complexity. Our approach handled multi-modular pipelines, with forwarding mechanisms and designed with a wide range of Chisel features. Finally, we have evaluated our approach on several in-order RISC-V processors, with a single datapath pipeline module and multi-modular pipeline designs.

As for future work, we pursue two directions. First, we want to validate the constructed datapath models against processor simulators so as to replace the current manual validation of these models with an automated procedure. Second, we want to generate formal datapath models in order to integrate them in formal verification frameworks towards hardware/software co-verification of functional [28] or non-functional (e.g. timing) [29] requirements.

References

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

[2] 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.

[3] J. Engblom and B. Jonsson, "Processor pipelines and their properties for static WCET analysis," in *Embedded Software, Second International Conference, EMSOFT 2002*, ser. Lecture Notes in Computer Science, vol. 2491, 2002, pp. 334–348.

[4] 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*, ser. LNCS, vol. 4444, 2006, pp. 12–52.

[5] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: an open toolbox for adaptive WCET analysis," in *SEUS*, ser. LNCS, vol. 6399, 2010, pp. 35–46.

[6] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *WCET*, ser. OASICS, vol. 57, 2017, pp. 8:1–8:12.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] C. Papon, "Spinalhdl: An alternative hardware description language," in *FOSDEM*, 2017.

[12] 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.

[13] S. A. Bensaid, M. Asavoae, F. Thabet, and M. 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. [Online]. Available: https://doi.org/10.1109/RTAS54340.2022.00034

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

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

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

[17] 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.

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

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

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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 Proceedings*, ser. Lecture Notes in Computer Science, vol. 2144. Springer, 2001, pp. 13–39.

[27] 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.

[28] R. Mukherjee, S. Joshi, J. O'Leary, D. Kroening, and T. Melham, "Hardware/software co-verification using path-based symbolic execution," *CoRR*, vol. abs/2001.01324, 2020.

[29] 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*, ser. Lecture Notes in Computer Science, R. D. Chamberlain, M. E. Grimheden, and W. Taha, Eds., vol. 11971, 2019, pp. 203–227.