Work in Progress: Automatic Construction of Pipeline Datapaths from High-Level HDL Code

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

***Abstract*—Safety-critical systems rely on worst-case timing analysis under architecture considerations to ensure that their timing bounds could be guaranteed. Usually, such architecture models are constructed by hand, from processor manuals. However, with open hardware initiatives and high-level Hardware Description Languages (HDL), automation would and should be possible. In this paper, we present an approach for constructing pipeline datapath models from processor designs described in high-level HDLs. We propose a methodology based on the Chisel/FIRRTL Hardware Compiler Framework and we report preliminary results 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 regulated by comprehensive standards as a means to prevent hazardous events. For example, missing a timing deadline is an unacceptable event and a solution is to characterize the timing behavior of these systems with adequate timing bounds. Worst-case timing analyses are able to compute safe (and precise) timing bounds and to provide the necessary guarantees.

State of the art static timing analyzers such as aiT [6], Otawa [4], Heptane [7] or Chronos [10] consider architecture models (of caches or pipelines) and use static analysis to characterize their timing. These architecture models are usually developed by hand and sometimes validated against hardware simulators for conformance (e.g. for aiT [15]). A closer code inspection of their pipeline models expose their common attributes. First, the pipeline stages are represented with a single variable, reducing a pipeline stage to an identification attribute. Second, these models focus on instruction progression and not its actual computation (i.e. the correctness of program execution is assumed in the context of static timing analysis). Third, these models execute basic blocks (i.e. straight-line code) and hence, pipeline optimizations like forwarding are not explicitly encoded in the hardware pipeline model but handled at the code-level through arbitrary timing penalties.

Thanks to open hardware initiatives, many processor designs become available[1], supported by high-level HDLs, like Chisel [3], and associated extensible compilation chains, like FIRRTL [8]. These compilation chains come with configurable optimizations and a pass infrastructure, which facilitate the analysis of hardware designs (more difficult to address from Verilog/VHDL). In this paper, we thus propose an analysis of Chisel/FIRRTL-based processor designs, in order to determine pipeline models for static timing analysis. Our analysis addresses (1) how to determine the pipeline depth (i.e. the number of stages) and (2) how to connect these stages, to form a datapath, that can later easily be abstracted to match the ones of WCET analyzers. This analysis focuses on the pipeline registers to capture a cycle-accurate behavior, as required by a WCET analysis, while improving on the pipeline models of WCET analyzers. More precisely, our analysis could expose the register forwarding, creating a separation of concerns between architecture and program models and making this model suited for a wider range of timing properties (e.g. timing anomalies, security-related etc.). We apply our analysis on several processors and report preliminary results.

Related Work. Our work addresses code-level analysis (of hardware designs) to extract pipeline models for timing analysis, The works in [13], [14] address a similar goal, that of analyzing processor (Verilog/VHDL) design code [13] and derive architecture models for the WCET analysis [14]. Our approach considers designs developed with more expressive languages (Chisel/FIRRTL). In contrast to our approach, which engineers a solution and then aims to validate it, the work in [13] uses the abstract interpretation framework to construct sound processor abstract models. This particular work is applied in [14] to determine architecture models using a semi-automatic procedure, while our analysis is fully automated in constructing the pipeline model. Abstract pipeline models (including forwarding) are addressed in [5], from processor graphs (i.e. structures combining combinational and sequential logics and which could be generated from Verilog/VHDL code). The actual construction also relies on register placement, as our approach, but requires as input the pipeline depth. Besides, the goal of [5] is to formally verify the functional correctness of the pipeline optimizations, which is complementary to ours, that of preserving the timing behavior.

A related problem, addressed for example in [9], [11], is to synthesize pipelines from sequential representations. In comparison, our approach has an opposite goal, that of going from full-fledged pipeline implementations to simpler, abstract models. Also, bounding the pipeline depth is on of our goals whereas, for automated pipeline design, it is known. Moreover, our approach works on richer input representations (i.e. code). Finally, in [9], [11], the correctness of automated pipeline design is systematically ensured during its construction, while we verify the result against the original design as we rely on heuristics to produce a solution.

[1] A non-exhaustive list is provided later in the paper

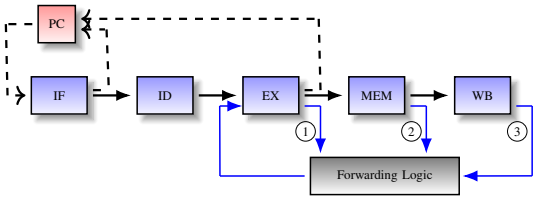

Fig. 1: Classical 5-stage pipeline with forwarding.

II. Illustrating Example

Fig. 1 shows the stages of a standard in-order 5-stage pipeline, i.e. from instruction Fetch (*IF*) to register Write-Back (*WB*). 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 due to space constraints. The datapath, presented with solid lines, considers the usual left-to-right advancement through the pipeline stages, with an optimization, in the form of register forwarding, represented with blue lines. There is thus 3 forwarding paths ① - ③, from the pipeline stages Execute (*EX*), Memory (*MEM*) and Write-Back (*WB*) to *EX* respectively. The remainder of this paper illustrates our contribution over the Sodor RISC-V processor [2], whose pipeline strictly adheres to this representation shown in Fig. 1.

Listing 1 now presents a (simplified) snapshot of Sodor's pipeline[2], that is written in the Chisel high-level HDL [3]. Chisel hardware construction primitives (e.g. for wires, multiplexers, registers etc.) are embedded in the Scala programming language, which facilitates design reuse and automated generation of digital logic designs. A Chisel-based hardware design is a set of modules, each module defines input/output (wire) ports to connect its sequential and combinatorial logics to other modules. Chisel lies at the top of a hardware compiler framework, compiling into an intermediate language named FIRRTL and further, into Verilog code. The FIRRTL language mainly exposes hardware primitives while stripping out the Scala features and maintaining module, scoping and naming traceability. This language, the Abstract Syntax Tree (AST) and its visitors are described in [8].

Listing 1: Snapshot of the Chisel code of the Sodor pipeline.

```
1  val if_pc  = RegInit (size)
2  val dec_pc = RegInit (size)
3  val exe_pc = RegInit (size)
4  val exe_rs2_data = Reg (size)
5  val dec_rs2_data = Wire (size)
6  ...
7  dec_pc := if_pc
8  /* C1-C3: enable and selection signals */
9  dec_rs2_data := MuxCase (rf_rs2_data,
10   Array (C1 → exe_alu_out, /* ①*/
11          C2 → mem_wbdata, /* ② */
12          C3 → wb_wbdata /* ③ */)
13 ...
14 when (C4) /* C4: no stalling condition */ {
15   exe_pc := dec_pc
16   exe_rs2_data := dec_rs2_data }
```

[2]Names of registers have been simplified, i.e. `_reg_` have been removed.

First, notice the register `if_pc` implementing the PC shown in 1 (the `RegInit` object line 1 and updating another register at line 7). The forwarding to the `EX` stage is addressed through the Chisel wire `dec_rs2_data`, declared at line 5 (i.e. the `Wire` object). It is then updated between lines 10 to 12 through a cascade of multiplexers, conveniently represented by a mix of Chisel and Scala features, as `MuxCase` and `Array` objects respectively. Signals for wire enable/selection or stalling `C1-C4` are inputs to these multiplexers (not defined to simplify), as well as the important guarded updates using a `when` block (lines 14 to 16).

III. Pipeline Datapath Analysis

In order to construct abstract pipeline datapaths, we need to address two objectives: to determine the pipeline depth (i.e. number the pipeline stages) and to present how these stages are connected. We focus on the registers within the processor design in order to determine precedence relations between these registers. First, 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 $AST_{\mathcal{P}}$. Each module $M \in \mathcal{P}$ is defined by a set I/Os of input/output ports, a set $Regs$ of registers (or storage elements), a set $Combs$ of wires representing the combinatorial logic of M, a set $Ctxs$ of contexts where the elements of $I/Os \cup Regs \cup Combs$ are updated and finally a set $Exts$ of the sequential and combinatorial logics external to M (but referenced within M). $Ctxs$ captures all the scopes of M. An element $ctx \in Ctxs$ is defined as a mapping between the context condition and the respective register updates. For brevity, we assume that a pipeline $\mathcal{P}$ relies on a single module M, but our formalization addresses the more general case where it is made of a set of modules.

Example 1: Listing 1 presents a Chisel module with wires `dec_rs2_data`, `exe_alu_out` and `mem_wbdata` $\in Combs$ (the definition of the last two is omitted due to space constraints), and registers `if_pc`, `dec_pc`, `exe_pc`, `exe_rs2_data` $\in Regs$. Finally, {`C4` $\mapsto$ `[exe_pc, exe_rs2_data]`} $\in Ctxs$.

Next, we define an intermediate representation of the processor pipeline $\mathcal{P}$ based on the set of registers $Regs$ and relations between them (i.e. to express the timing of $\mathcal{P}$) as well as the set of contexts $Ctxs$ (i.e. to cater for how the processor/pipeline 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 from $AST_{\mathcal{P}}$ using the standard visitor from [8]. The initial partition of the processor design $\mathcal{P}r$ into the pipeline design P is also sufficient to determine the set $Exts$. Next, we determine dependencies between registers through a visitor combinator which we name *register-context analysis*. It characterizes each register by an input-output relation w.r.t. the other elements of the processor. Given a register r in a module, a visitor

combinator iteratively collects the AST_P nodes which affect the inputs of r. This iterative process corresponds to a standard dataflow analysis which terminates in several situations, defined next.

Definition 1: For a register $r \in Regs$ within $\mathcal{P}$, the **input frontier** of r is $\bigcup_{i=1}^{n} c_i$, where $c_i \in Regs$ or $c_i \in I/Os$ or $c_i \in Exts$.

Such an input frontier contains three kinds of design elements: other registers (i.e. that precede r in the abstract pipeline datapath), the ports of the module[3], and finally, other design elements out of the considered pipeline $\mathcal{P}$.

Example 2: The input frontier of the (forwarding destination) register `exe_rs2_data`, in Listing 1 (i.e. between the stages `ID` and `EX`, in Fig. 1), contains itself (case ①, combinatorial input `exe_alu_out` partially shown due to space constraint) as well as the other two cases, e.g. `mem_wbdata` (case ②) and `wb_wbdata` (case ③), through the `dec_rs2_data` wire.

Currently, a register frontier contains only once each register fulfilling Definition 1. However, a Chisel `when-.elsewhen` construct could include updates of a same register, in the different condition paths. This enables a context-dependent analysis, but which is left as future work.

Definition 2: For a register $r \in Regs$, the **register context** of r, denoted by $\mathcal{C}_r$, is defined by the pair $\langle in_r, out_r \rangle$ with in_r and out_r being the input frontier of r and the output wire of r respectively, with $out_r \in Combs$.

Definition 3: For two register contexts $\mathcal{C}_{r1}$ and $\mathcal{C}_{r2}$ of $r1$ and $r2$ respectively, a predicate $\texttt{prec}(\texttt{r1}, \texttt{r2})$ 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 $\texttt{prec}(r1, r2)$ evaluates to true. $Pred$ is the set of edges of the intermediate representation which is further used to determine the abstract pipeline datapath.

Definition 4: 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 $E = Pred$.

Definition 5: The operator $\texttt{ctx\_reg} : Regs \rightarrow 2^{Regs}$ is defined as $\texttt{ctx\_reg}(\texttt{r}) = R$ 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 overview. The abstract pipeline datapath of $\mathcal{P}$ is constructed by unfolding $IR_{\mathcal{P}}$ and assigning nodes (i.e. registers) to pipeline stages using an operator `to_stage` : $Regs \rightarrow \mathbb{N}^+$. The procedure assumes, as a starting point, the program counter $PC \in Regs$ (see Fig. 1), located in the fetch stage of the pipeline (i.e. $\texttt{to\_stage}(\texttt{PC}) = 1$). We leave as future work the identification and thus the placement of this register. Our algorithm distinguishes two cases from a $IR_{\mathcal{P}}$ graph, a case 1 driven by register dependencies and a case 2 driven by register context dependencies.

Case 1 places a register r, checking if all its sources are already placed. If so, the stage assigned to r is strictly following the minimal stage assigned among its source registers. Otherwise r should be placed by case 2. Formally, case 1 has $\texttt{to\_stage}(\texttt{r}) = i$ if $\exists (r_1, r) \in Pred$ and $\texttt{to\_stage}(\texttt{r}_1) = i - 1$, and $\forall (r_2, r) \in Pred$, $r_2 \neq r_1$ $\texttt{to\_stage}(\texttt{r}_2) > \texttt{to\_stage}(\texttt{r}_1)$ (to account for registers already placed in later stages than r_1 and having backward edges to r). This case is applied first to the components connected to PC. Case 2 places a register r wrt. the already placed registers from the same context. Formally, case 2 has $\texttt{to\_stage}(\texttt{r}) = i$ if $\texttt{ctx\_reg}(\texttt{r}) = R$ such that $\exists r_1 \in R$ with $\texttt{to\_stage}(\texttt{r}_1) = i$.

The solution produced by our algorithm is a subgraph of $IR_{\mathcal{P}}$ with nodes, i.e. registers, for which pipeline stages have been assigned. Some registers may not have been assigned to a pipeline stage due to the heuristic nature of our algorithm, coming from the combination of cases 1 and 2.

To validate the results of our algorithm, we rely on a correctness criterion by defining a subsumption relation $\mathcal{S} : P_{IR_{\mathcal{P}}} \subset Q_{\mathcal{P}r}$ relating the original processor design $Q_{\mathcal{P}r}$ 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. Preliminary results

The results of our pipeline datapath construction are presented in Table I for 3 Chisel-based in-order RISC-V processors: RISC-V mini [1], Sodor [2] and KyogenRV [12]. RISC-V mini [1] features a simple 3-stage pipeline designed to serve as an initial testcase when designing last versions of Chisel itself. Sodor is a family of processors coming with different pipeline depths and with and without forwarding mechanism (noted FW and WFW respectively in Table I). We consider its 5-stage version. Finally, KyogenRV [12] is also a 5-stage pipeline processor targeting Intel FPGAs and developed for academic purposes. The first column of Table I reports the code size of each pipeline ($\mathcal{P}$), the next three columns present statistics related to the numbers of registers ($\#Regs$), of contexts ($\#Cxts$), the largest input frontier within pipelines (#Frt). The last two columns detail the number of registers successfully placed by the 2 cases of our algorithm.

TABLE I: Experimental results on RISC-V processor designs.

	LOC	$\#Regs$	$\#Cxts$	#Frt	Case 1	Case 2
RISC-V Mini	241	15	3	8	5	10
Sodor (WFW)	594	48	14	37	36	12
Sodor (FW)	646	48	14	88	34	14
KyogenRV	4567	93	59	35	47	36

The depth of each pipeline has been correctly computed. Note that our analysis initially identified the KyogenRV processor as a 7-stage pipeline. This inaccurate result was due to the use of the Chisel `RegNext` hardware construction primitives within this processor. The semantics of `RegNext` is to produce a one-cycle delayed version of associated signal. At the FIRRTL level, it is translated into an additional register

[3]Which are useful to address multi-module pipeline designs not presented here due to space constraint.

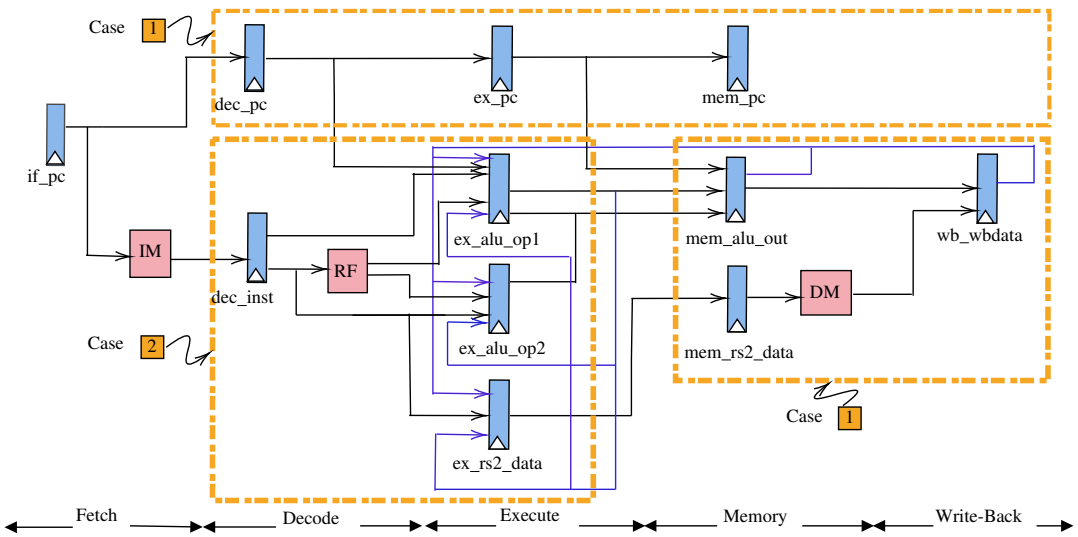

Fig. 2: RISC-V Sodor 5-stages pipeline datapath model.

for each delay and thus an additional pipeline stage. However, these registers can be easily identified based on their names, allowing us to discard them when computing abstract pipeline datapaths. It can also be noticed that all the registers of Sodor (both FW and WFW) and RISC-V Mini have been successfully placed, while it is not the case for KyogenRV. The 10 remaining registers that were not placed within its pipeline are in fact not related to the datapath but instead to the control path and the register file. Finally, it can be noticed an expected difference in the size of the largest input frontier (#Frt) for Sodor between the versions with and without forwarding.

We illustrate the construction of the pipeline datapath model for the RISC-V Sodor 5-stage processor. The model is made of 48 registers and we present, in Fig. 2, some that we consider relevant. Our analysis generates the $IR_{\mathcal{P}}$ graph and proceeds to register placement. We specify that `if_pc` (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 $IR_{\mathcal{P}}$ (case 1, shown in yellow in Fig. 2). Then, our algorithm places the register `dec_inst` as it is updated in the same context as `dec_pc` (case 2). Case 1 indeed does not apply as the register frontier of `dec_inst` contains a reference to an external design element, the module IM (instruction memory). The next registers to be placed are `ex_alu_op1`, `ex_alu_op2` and `ex_rs2_data`. Their respective input frontiers contain registers not currently placed (i.e. `mem_alu_out` and `wb_wbdata`), due to the backward edges in blue coming from the data forwarding semantics. However, the context of register `ex_pc` contains these non placed registers enabling the use of the case 2 of our algorithm. Finally, our algorithm places the remaining registers (i.e. `mem_alu_out`, `mem_rs2_data`, and `wb_wbdata`) based on the register dependencies (case 1).

V. Future Work

We are currently working on expanding this analysis to address more complex, multi-module pipeline designs, but also out-of-order pipelines. We also plan to generate abstract (formal) pipeline models in order to plug them into WCET analyzers or detect timing anomalies.

References

[1] Risc-v mini. https://github.com/ucb-bar/riscv-mini,
[2] Risc-v sodor. https://github.com/ucb-bar/riscv-sodor,
[3] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: Constructing hardware in a scala embedded language. In: DAC. p. 1216–1225 (2012)
[4] Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: an open toolbox for adaptive WCET analysis. In: SEUS. LNCS, vol. 6399, pp. 35–46 (2010)
[5] Charvát, L., Smrcka, A., Vojnar, T.: HADES: microprocessor hazard analysis via formal verification of parameterized systems. In: MEMICS. EPTCS, vol. 233, pp. 87–93 (2016)
[6] Ferdinand, C., Martin, F., Cullmann, C., Schlickling, M., Stein, I., Thesing, S., Heckmann, R.: New developments in WCET analysis. In: Program Analysis and Compilation, Theory and Practice. LNCS, vol. 4444, pp. 12–52 (2006)
[7] Hardy, D., Rouxel, B., Puaut, I.: The heptane static worst-case execution time estimation tool. In: WCET. OASICS, vol. 57, pp. 8:1–8:12 (2017)
[8] Izraelevitz, A.M., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., Kim, D., Schmidt, C., Markley, C., Lawson, J., Bachrach, J.: Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In: ICCAD. pp. 209–216 (2017)
[9] Kroening, D., Paul, W.J.: Automated pipeline design. In: DAC. pp. 810–815. ACM (2001)
[10] Li, X., Yun, L., Mitra, T., Roychoudhury, A.: Chronos: A timing analyzer for embedded software. Sci. Comput. Program. **69**(1-3), 56–67 (2007)
[11] Nurvitadhi, E., Hoe, J.C., Kam, T., Lu, S.: Automatic pipelining from transactional datapath specifications. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **30**(3), 441–454 (2011)
[12] Saitoh, A.: KyogenRV: simple 5-staged pipeline RISC-V. https://github.com/panda5mt/KyogenRV,
[13] Schlickling, M., Pister, M.: A framework for static analysis of VHDL code. In: WCET. OASICS, vol. 6 (2007)
[14] Schlickling, M., Pister, M.: Semi-automatic derivation of timing models for WCET analysis. In: LCTES. pp. 67–76. ACM (2010)
[15] Wilhelm, R., Pister, M., Gebhard, G., Kästner, D.: Testing implementation soundness of a WCET analysis tool. In: A Journey of Embedded and Cyber-Physical Systems. pp. 5–17. Springer (2021)