Modelling and verification of parameterized architectures: A functional approach

Salah Merniz1 | Saad Harous2

1MISC Laboratory, Constantine2 University, Constantine, Algeria
2College of Information Technology, United Arab Emirates University, Al Ain, United Arab Emirates

Abstract
The merit of higher order functions for hardware description and transformation is widely acknowledged by hardware designers. However, the use of higher order types makes their correctness proof very difficult. Herein, a new proof approach based on the principle of partial application is proposed which transforms higher order functions into partially applied first-order ones. Therefore, parameterised architectures modelled by higher order functions could be easily redefined only over first-order types. The proof could be performed by induction within the same specification framework that avoids translating the higher order properties between different semantics, which remains extremely difficult. Using the notion of parameterisation where verified components are used as parameters to build more complex ones, the approach fits elegantly in the incremental bottom-up design where both the design and its proof could be developed in a systematic way. The potential features of the proposed methodological proof approach are demonstrated over a detailed example of a circuit design and verification within a functional framework.

1 | INTRODUCTION AND MOTIVATION

Within the framework of functional programming, the merit of higher order functions (HOFs) called combining forms is widely acknowledged by hardware designers. Their ability to make abstractions allows them to be used for many purposes such as: structuring and reusing circuit specifications [1,2], expressing regular designs [3,4], optimising designs [2,5], developing parallel pattern-based models [6–8], etc. However, if HOFs provide many advantages at the specification level, their correctness proof remains a tedious task. The main reason relates to the use of higher order types which make the process of showing that a design meets its specification for all input values, or defining standard higher order homomorphisms extremely difficult. Such obstacles lead some authors to consider rather weakening higher order homomorphism [9] (lacking the congruence relation that requires complicated logic calculus involving partial ordering on types), and some authors to simulate higher order types by first-order ones within an appropriate framework, such as higher order logic [10,11], term rewriting systems [12,13], functional programming [14–17]. Most of the proposed approaches use a proof tool to reason about the HOFs properties which still remain very difficult to translate and prove [18–21]. Moreover, some higher order cases are still difficult [22].

HOFs have been extensively used for building complex hardware designs whose proofs remain an unaccomplished goal. In Ref. [2], the author uses HOFs within CaSH (subset of Haskell) to develop complex regular architectures which are validated only by simulation before being translated into the VHDL code. The authors of Ref. [6], propose a reconfigurable architecture framework called Plasticine for developing, simulating, and implementing efficient pipelined SIMD architectures over HOFs (called parallel patterns) within a Scala-based HDL. Parallel patterns are implemented in Plasticine by a collection of low level Pattern Units which precisely capture the semantics of functional and memory hardware units. However, neither the pattern units nor the targeted designs are

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2021 The Authors. IET Computers & Digital Techniques published by John Wiley & Sons Ltd on behalf of The Institution of Engineering and Technology.
verified. In Ref. [8], the authors present a similar approach within a functional framework called Lift, which provides both high-level patterns for describing designs at high level of abstraction, and low-level patterns (as rewrite rules) for making optimisations and generating FPGA target implementations. However, such patterns which support both the design construction and its transformations are used without verification. The authors of Ref. [23], use HOFs within CλaSH for modelling Run-Time Reconfigurable designs, and use assertion-based verification to ensure their correctness. Assertions are described by functions and implemented on hardware designs. Therefore, their proof approach requires more space and time, and consequently, it remains critical with respect to the design scalability. In Ref. [24], the authors make use of full Haskell to develop high-order descriptions of processor designs which are defunctionalized (for the synthesis purpose) into first-order implementations in ReWire (subset of Haskell). The correctness of the high-level functional description is ensured by a theorem which is proved outside of ReWire. To show the design scalability of their approach, they applied it to the design of both single core and dual core (by instantiating two copies of the single core). While their approach scales well with the design complexity, it remains however, unclear with respect to the proof scalability.

This article proposes a new methodological approach within Haskell framework for the formal specification and verification of parameterised hardware designs (that are built over HOFs). The proposed approach combines both the notion of parameterisation (to structure the design) and the notion of partial application (to ease the proof) to incrementally build and prove complex hardware designs.

In our context, both first-order and higher order specifications represent functional programs that could be used for both formal verification and simulation (actual designs are validated by mixing these two techniques [25,26]). Moreover, the methodology carries out the proof by induction within the same specification framework, that is both the design and its proof could be carried out using the same semantics. The use of a separate proof tool requires translating the design properties from the specification framework to the proof framework, and so, the soundness property remains difficult.

The proposed proof methodology follows the layered vertical-horizontal design approach which has been proposed as one of the most innovative contribution in our previous work [27] on first-order models. Such work involves only the system behaviour while it says nothing about its structure. Complex systems are incrementally built one over the other and consequently, their structure (as well as their reliability) is very crucial. The objective of the present work is to show how such systems are built and proved. For this purpose, we introduce two basic concepts: the notion of parameterisation (implemented by the notion of HOF) which is the key point for structuring large designs, and the notion of partial application to simplify the proof. With respect to our previous work, this methodology also shares both the notion of state functions (to formally represent machine states) and the notion of time functions (to link synchronous points where the states of two different implementations could be compared).

Throughout this work, we provide an effective framework in which both the design and its proof could be developed in a systematic way.

Herein, Section 2 introduces the basic definitions underlying our proof methodology. Section 3 presents the principle of the proposed proof methodology through the layered vertical-horizontal design approach. Section 4 is dedicated to the modelling and verification of the different micro-architectures: Sequential, pipelined, and superscalar. Section 5 presents a detailed case study of our proof methodology with respect to MIPS processors. Finally, the conclusion section outlines the main contributions of this research work.

2 | PRELIMINARIES

Haskell [28] is a non-strict purely functional language that provides in addition to its formal semantics definition (to support formal reasoning) powerful features such as: function composition, parallelism, lazy evaluation, polymorphism, higher order functions, etc., which demonstrated its viability with respect to complex hardware designs. Throughout this work, we will use plain Haskell (to deeply exploit its powerful features) instead of a functional HDL (Hardware Description Language). General Haskell could be subsequently translated into a restricted intermediate representation [29,30] before translating it to an imperative HDL like VHDL, or Verilog.

2.1 | Polymorphic higher-order functions

A higher order function (HOF) is a function that takes functions as arguments or returns a function as a result. Polymorphic HOFs express elegant regular designs while enhancing their abstraction through their parametric polymorphism. Figure 1 shows both the definition and the implementation of (map-scan-right) mscan HOF [31]. It combines three standards HOFs: map, fold and scan in only one operation.
2.2 Curried and partially applied HOFs

Haskell allows a (higher order) function to be defined in a curried form; that is, it takes its arguments separately (juxtaposed). In this way, it can be partially applied (to fewer arguments than it was) to form a new function (called specialisation) defined only over first-order types. For instance, let us consider the polymorphic HOF \textit{mscarr} specified so far in Haskell.

Let data \texttt{Bit = Zero | One, and fadd be a function whose type matches the type of the functional argument f.}
\[ f :: a \rightarrow b \rightarrow (b, c) , \quad \text{\texttt{Fadd}} :: \texttt{(Bit, Bit)} \rightarrow \texttt{Bit} \rightarrow (\texttt{Bit, Bit}) \]


\[ \text{Where } a = (\texttt{Bit, Bit}), b = \texttt{Bit}, c = \texttt{Bit}. \]

Then, from the definition of mscarr, we can create a new partially applied function by applying \textit{mscarr} to \textit{fadd} as follows:
\[ f1 = \textit{mscarr fadd} :: \texttt{Bit} \rightarrow ([\texttt{Bit}, \texttt{Bit}]) \rightarrow (\texttt{Bit, Bit}). \]

Such a composite function whose type signature involves only first-order values, models the behaviour of a concrete design with a well-defined interface as shown in Figure 2.

2.3 State and next-state functions

Throughout this work we will use the notion of state functions for modelling complex processor microarchitectures involving memories.

Let \( S \) be a non-empty set, called the state space. A state function with an initial state \( c::S \), and a next-state function: \( f: S \rightarrow S \), is recursively defined as follows:
\[ F:: (\text{Int}, S) \rightarrow S . \]
\[ F(0, c) = c . \]
\[ F(n, c) = f ( F((n-1)c) ) . \]

The distributed aspect of a machine state space over its components requires decomposing the state and the next-state functions into coordinates.

Let \( S = (S_1,..,S_k) \) be the state space distributed over \( k \) components (the observables) where \( S_i \) is the state of the \( i^{th} \) component, for \( 1 \leq i \leq k \). Thus, the state function will be decomposed as follows:
\[ F(n,c_1,..,c_k) = (F_1(n, c_1,..,c_k),..,F_k(n,c_1,..,c_k)) \]
\[ = (f_1(F((n-1),c_1,..,c_k)),..,f_k(F((n-1),c_1,..,c_k))) \]

Taking the initial state into account, the distributed form of the state function \( F \) will be redefined as follows:
\[ F:: (\text{Int}, S) \rightarrow S \]
\[ F(0, c_1,..,c_k) = (c_1,..,c_k) \]
\[ F(n, c_1,..,c_k) = let \ c_i' = f_i ( F((n-1), c_1,..,c_k) ) \]

The distributed form of the next-state function \( f \) (transition function) is defined below:
\[ f: S \rightarrow S . \]
\[ f(c_1',..,c_k') = \text{let } c_1' = f_1( c_1',..,c_k') \]
\[ \vdots \]
\[ c_k' = f_k( c_1',..,c_k') \]

in \( (c_1',..,c_k') \)

Such form will be very important for the specification of designs that require performing only one step, such as the execution of one instruction.

3 Design METHODOLOGY

3.1 Layered design approach

The proposed proof methodology allows to incrementally model and prove state-of-the-art parameterised designs. The verification process follows the layered vertical-horizontal design approach shown in Figure 3. The highest level represents the Instruction-Set-Architecture (ISA) specification that describes the semantics of the processor’s operations. The Micro-Architectural (MA) level represents the top-level design implementing the ISA specification. All MA designs which are hierarchically built one over the other represent different implementations of the same ISA specification. Three MA designs will be considered; the sequential MA design (SMA), the pipelined MA design (PMA), and the superscalar MA design (SSMA). The SMA design which involves only first-order functions (and consequently its proof could be easily performed against an ISA specification) represents the reference core architecture over which will be hierarchically developed both the PMA and the SSMA designs. So, the PMA will be parameterised with the SMA over which it is built, and the SSMA will be parameterised with the PMA over which it is built as well. Both pipelined and superscalar designs will be proved against the reference SMA design. Other approaches such as: the flushing technique [32–34], the completion functions method [35], the one-step theorem [36] and the well-founded bisimulation [37] attempt to prove the Micro-Architecture/Register-Transfer design against an ISA specification. However, such approaches reveal many difficulties [38] to link the MA/RT level to the ISA specification.
level for pipelined and superscalar designs (instructions overlap) that require defining complex abstraction functions.

3.2 | Proof principle of the microarchitectural level

3.2.1 | Formal verification of the first-order implementation

Let $S$ and $W$ be two finite sets of first-order types, representing values at the ISA and MA levels, respectively.

Let $\text{isa} :: s_1 \rightarrow \ldots \rightarrow s_n \rightarrow S$ and $\text{sma} :: w_1 \rightarrow \ldots \rightarrow w_n \rightarrow W$, be the first-order type signatures of the ISA specification and the corresponding first-order implementation (SMA), respectively, where: $s_1, \ldots, s_n, S \in S$, and $w_1, \ldots, w_n, W \in W$.

Let $\text{abs} :: W \rightarrow S$ be an abstraction function that maps the MA level to the ISA level. Then, proving the correctness of the SMA implementation with respect to the ISA specification requires proving the following commutative property:

$$\forall v_1 :: w_1, \ldots, v_n :: wn$$

$$\text{isa}(\text{abs}(v_1), \ldots, \text{abs}(v_n)) = \text{abs}(\text{sma}(v_1, \ldots, v_n))$$

Such a property is depicted in Figure 4, by the corresponding commutative diagram. Throughout this work, the sequential microarchitecture implementation will be assumed correct. A detailed proof about sequential microarchitectures is given in our previous work [39].

3.2.2 | Formal verification of the higher order implementation

Let $H$, be a higher order curried function modelling a system design as follows:

$$H :: t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t_{s_1} \rightarrow \ldots \rightarrow t_{s_n} \rightarrow (t_{y_1}, \ldots, t_{y_n})$$

$$H \ f_1 \ldots \ f_n \ x_1 \ldots \ x_n = (y_1, \ldots, y_n)$$

In our framework, such higher order definition describes the architecture of a system design (structure) and will be interpreted as follows: the function name $H$ denotes the system name, the functional parameters $f_i$ represent its components, the variables $x_i$ denote the inputs, and $y_i$ denote the outputs as depicted in Figure 5a. The polymorphic aspect of the parameters $f_i$ provides the ability of changing the functionality of the system components just by instantiating them with real design operations (technology implementation independent). However, to avoid dealing with functional types in the proof, our approach proposes deriving new partially applied first-order functions from HOFs. Notice that, HOFs must be defunctionalised into first-order functions during the synthesis process to make them translatable.

Let $F$ be a partial first-order function (specialisation) defined over $H$ as follows:

$$F :: tx_1 \rightarrow \ldots \rightarrow t_{s_n} \rightarrow (t_{y_1}, \ldots, t_{y_n})$$

$$F = H \ f_1 \ldots \ f_n$$

The function $F$ involves only first-order types and describes precisely the behaviour of the system design whose structure is described by $H$, as shown in Figure 5b. Therefore, our approach reduces the verification of the design modelled by $H$ to the verification of the design modelled by $F$. So, our approach assigns naturally a structural semantics to the H.O. Implementation and a behavioural (computational) semantics to the F.O. Implementation. Other approaches give a dual approach assigns a structural semantics to the same design: a structural semantics, to build it and a computational semantics to simulate it [2,6].

4 | MODELLING AND VERIFICATION OF THE MICROARCHITECTURAL DESIGNS

4.1 | Modelling sequential microarchitectural designs

4.1.1 | Modelling at the instruction level

At the microarchitectural level, the state of a sequential machine could be modelled by the SMA next-state function which is depicted in Figure 6b. It allows observing the evolution of the state $C^j_i$ produced by the stage functions $f_i$ during the instruction execution according to the diagram shown in Figure 6a.

4.1.2 | Modelling at the program level

Let $n$ be the number of instructions, then, the evolution of the state at the program level is captured by the SMA state function which is defined as follows:

$$\text{SMA} :: (\text{Int}, W \rightarrow W)$$

$$\text{SMA}(0, c_0^1, \ldots, c_0^j) = (c_0^1, \ldots, c_0^k)$$

$$\text{SMA} (n, (c_0^1, \ldots, c_0^j)) = \text{sma}(\text{SMA}(n-1), c_0^1, \ldots, c_0^j)$$
By unfolding the $sma$ function, the SMA definition rewrites, as shown in Figure 7.

$$sma : W \rightarrow W$$

$$sma(c_0, \ldots, c_s) =$$

let $c_1 = f(c_0, \ldots, c_s)$

in $(c_1, \ldots, c_s)$

By unfolding the $sma$ function, the SMA definition rewrites, as shown in Figure 7.

By unfolding the $sma$ function, the SMA definition rewrites, as shown in Figure 7.

4.2 | Modelling pipelined microarchitectural designs

4.2.1 | PMA higher order implementation

The PMA (Pipelined Microarchitectural) model will be formalised in terms of clock cycles at the program level. Its construction requires two steps, as depicted in Figure 8a

The first step which is an irregular computation fills progressively the pipe till the cycle $k = S - 1$, where $S$ is the number of stages. At each clock cycle a new stage function is activated that computes a new component state.

The second step which starts from the cycle $k \geq S$, is a regular computation: It allows to recursively compute the PMA state by applying the next state function $f_k$ defined as follows: $f = (f_1, \ldots, f_s)$. It involves $S$ parallel functions $f_i$, each one performs the task of a stage $S_i$.

Figure 8b shows the PMA higher order model parameterised with the stage functions $f_i$ of SMA.

4.2.2 | PMA first-order implementation

From the PMA higher order definition, we will derive a partial first-order definition that captures the behaviour of the same design and is easier to prove.

Let $FPMA$ be a partially applied first-order function defined as follows:

$$FPMA : (\text{Int}, W) \rightarrow W$$

$$FPMA \circ f_s \cdots f_1 (\text{Int}, W) \rightarrow W$$

Then, we get a simple recursive definition of a state function depicted in Figure 9.

4.3 | Verifying pipelined microarchitectural designs

4.3.1 | Time function

Because both pipelined and sequential MA models are formalised in terms of clock cycles, all synchronous intermediate
states represent useful points where the comparison could be achieved easily.

Let \( c^j_k \) denote the component state produced by the stage function \( f_i \) of a pipelined model with \( S \) stages, at clock cycle \( k \), for \( k \geq s: c^j_k = f_i(FPMA((k-1)c^j_1, ..., c^j_s)) \)

- In case of no stalls, the time function which links the pipelined and the sequential states is defined as follows: \( t_n(k,j) = (k-j)s + j \).

The corresponding sequential state is computed as follows:
\[
\hat{s}^j_{t_n(k,j)} = (f_1 \circ ... \circ f_1) (SMA((k-j)s)c^j_1, ..., c^j_s)
\]

In case of stalls, the time function rewrites as follows:
\[
t_n(k,j,e) = ((k-j)-e)s + j,
\]
where \( e \) is the number of stalls. Figure 10 shows such synchronisation.

4.3.2 Timed sequential model for pipelined designs

The TSMP (Timed Sequential Model for Pipelined designs) model against which the FPMA model will be verified is a first-order model developed over the SMA model and the time abstraction function. It inputs the same clock cycle \( k \) as the FMP A model, unlike the SMA model which inputs the number of instructions to execute. For each clock cycle \( k \), it builds \( S \) terms as shown in Figure 11, each one computes a partial result bound to an instruction within the pipe.

In this model, each one computes a partial result of instructions to execute for each clock cycle bound to an instruction within the pipeline. The corresponding sequential state is computed as follows:

\[
\hat{s}^j_{(k,j)} = (f_1 \circ ... \circ f_1) (SMA((k-j)s)c^j_1, ..., c^j_s)
\]

In case of stalls, the time function rewrites as follows:

\[
t_n(k,j,e) = ((k-j)-e)s + j,
\]
where \( e \) is the number of stalls. Figure 10 shows such synchronisation.

4.3.3 Criterion of correctness

Proving the behavioural equivalence of the FPMA model with respect to TSMP model, requires proving the following equation:

\[
\forall k :: \text{Int}, \forall c^j_1 :: W_1, c^j_2 :: W_s, \text{FPMA}((k, c^j_1, ..., c^j_s)) \Rightarrow \text{TSMP}((k, c^j_1, ..., c^j_s))
\]

This means that, starting from the same initial state: \( c^j_1, ..., c^j_s \) both models must reveal the same states at each clock cycle \( k \), according to the timing diagram shown in Figure 10. The proof of such an equation decomposes systematically to the proof of the following equations which are defined only over first-order types and provable by induction over clock cycles.

\[
f_1(FPMA((k-1), c^j_1, ..., c^j_s)) = f_1 (SMA((k-1)s,c^j_1, ..., c^j_s))
\]

\[
^\wedge f_1(FPMA((k-1), c^j_1, ..., c^j_s)) = (f_1 \circ ... \circ f_1) (SMA((k-s)s,c^j_1, ..., c^j_s))
\]

4.4 Modelling superscalar microarchitectural designs

4.4.1 SSMA higher order implementation

Superscalar designs extend pipelined designs by replicating pipelines so that to issue multiple instructions per clock cycle [40]. For the lack of space, only in-order execution designs will be considered. Thereby, the SSMA (SuperScalar Micro-Architectural) model could be developed upon the PMA model as follows:

Let \( W = ((W^1_1, ..., W^1_n), ..., (W^n_1, ..., W^n_n)) \), be the SSMA state distributed over \( n \) pipelines (each one with \( S \) stages), and \( pma_i \), for \( 1 \leq i \leq n \), be the component function that performs the functionality of the pipeline \( i \), using the stages functions \( f_i \).

Thus, given an initial state: \( (c_{i_1}^{11}, ..., c_{i_1}^{1n}, c_{i_2}^{11}, ..., c_{i_2}^{1n}, ..., c_{i_n}^{11}, ..., c_{i_n}^{1n}) \), and \( n \) pipelines: \( pma_1, ..., pma_n \) the SSMA state at clock cycle \( k \geq S \), is captured by the SSMA H.O function which is shown in Figure 12.

As we can see the superscalar model is built in a modular way over the pipeline model and consequently, its proof is subject to the proof of the pipeline modules.
4.4.2 | SSMA first-order implementation

The partial first-order function modelling the superscalar microarchitecture could be straightforwardly derived from the SSMA function as follows:

Let $\text{FSMA} = \text{SSMA pma}_1 \ldots \text{pma}_n :: (\text{Int}, \text{Int}, W) \rightarrow W$

$\text{fpma}_i = \text{pma}_i f_1 \ldots f_i :: (\text{Int}, W) \rightarrow W$.

Then we get the FSMA definition, as shown in Figure 13.

4.5 | Verifying superscalar MA designs

4.5.1 | Time function

The time function for a SSMA model generalises the time function used for the PMA model. Let $c_{ji}^k$ denote the SSMA component state produced by the stage function $f_j$ of the pipeline $i$, with $S$ stages at cycle $k \geq S$.

$c_{ji}^k = f_j(PMA_i((k-1), c_{1i}^k, \ldots, c_{si}^k))$

Let $n$, be the number of pipelines. Thus, two cases will be considered:

- **case of no stalls**: The time function is defined as follows:
  
  $t_n(k, j, i) = (n^*(k-j) + (i-1))*s + j$

- **case of stalls**: The time function rewrites as follows:
  
  $t_n(k, j, i, e) = (n^*(k-j) + (i-1) - e)*s + j$

where $e$ is the number of stalls (Fig. 14).

4.5.2 | Timed Sequential Model for Superscalar designs

The TSMS (Timed Sequential Model for Superscalar designs) model against which the FSMA model will be verified is shown in Figure 15. It is a first-order model built in a modular way over the TSMP model developed so far (Section 3.5.2) for the verification of the pipelined model. By substituting the $\text{tsmp}$ by its definition, we get a TSMS definition expressed with SMA terms that reflect exactly the timing between superscalar and sequential models.

4.5.3 | Correctness criterion

Proving the correctness of the FSMA model with respect to the TSMS model requires proving the following equation:

$\forall n, k :: \text{Int}, \forall c_{1i}^{1i}, \ldots, c_{ni}^{ni} :: W_n^1, \ldots, c_{mi}^{ni} :: W_n^m$

$\text{FSMA} (n, k, (c_{1i}^{1i}, \ldots, c_{ni}^{ni}), \ldots, (c_{1n}^{1n}, \ldots, c_{nn}^{nn})) = $

$\text{TSMS} (n, k, (c_{1i}^{1i}, \ldots, c_{ni}^{ni}), \ldots, (c_{1n}^{1n}, \ldots, c_{nn}^{nn}))$

The proof of such an equation decomposes to the proof of the following equations:

$\text{fpma}_1(k, c_{1i}^{1i}, \ldots, c_{ni}^{ni}) = \text{tsmp}_1(k, c_{1i}^{1i}, \ldots, c_{ni}^{ni})$

$\vdots$

$\text{fpma}_n(k, c_{1n}^{1n}, \ldots, c_{nn}^{nn}) = \text{tsmp}_n(k, c_{1n}^{1n}, \ldots, c_{nn}^{nn})$

By further substituting both the FPMA (section 3.4.2) and the TSMP (section 3.5.2) functions by their definitions, the superscalar FSMA model will be naturally proved against the sequential model SMA.

FiguRe 13 | Partial first-order implementation of SSMA

FiguRe 14 | Synchronisation for a superscalar design with $n$ pipelines

The corresponding sequential state is computed as follows:

$s'_{(k,j,i)} = (f_1 \circ \ldots \circ f_i) (\text{SMA}(n^*(k-j) + (i-1)), c_{ji}^k, \ldots, c_{ni}^k)$

- **case of stalls**: The time function rewrites as follows:

  $t_n(k, j, i, e) = (n^*(k-j) + (i-1) - e)*s + j$,

where $e$ is the number of stalls (Fig. 14).
4.6 | Modelling and verification flow diagram

The Build-and-Prove flow diagram depicted in Figure 16 summarises the different steps of our proof methodology for developing (using parameterisation) and proving (using partial application) higher order models of microarchitectural designs.

Figure 14. Synchronisation between Superscalar and Sequential Models.

5 | APPLICATION TO THE MIPS PROCESSOR MICROARCHITECTURES

This section shows a practical aspect of our proof methodology involving the MIPS processor microarchitectures. MIPS processors are well structured and modular RISC architectures and consequently they fit adequately in the incremental design approach. Three processors will be considered as three categories: the sequential MIPS core processor (SMA), the pipelined MIPS processor (PMA) and the superscalar MIPS dual-issue processor (SSMA). These processors which are compatible with respect to the MIPS ISA are incrementally developed one over the other. So, the pipelined processor will be parameterised with the sequential core over which it is built and the superscalar will be parameterised with the pipelined over which it is built as well. Both pipelined and superscalar designs will be proved against the reference sequential core which is supposed to be correct. A detailed proof methodology about sequential processors is described in Ref. [25].

5.1 | Modelling the MIPS SMA

5.1.1 | Instruction set format

This case study considers the implementation of three MIPS instruction types: an R-type instruction (add), the memory
reference M-type instructions (load and store) and a branch B-type instruction (beq). Their formats are shown in Table 1.

5.1.2 | MIPS core microarchitecture

The simplified microarchitecture of the MIPS core processor implementing the above instruction set is drawn in Figure 17. Its datapath is divided into five stages which are clocked in a round Robin. Each stage performs its functionality within one clock cycle. The Fetch and Decode stages are common to all instructions while the remaining stages: Execute, Memory-access and Write-back are specific to each instruction. Table 2 shows an informal description of its stages.

5.1.3 | Functional implementation of the MIPS SMA stages

The functions \( F_i \) which implement the functionality (in terms of hardware components) of the different stages are described in Table 3: The outputs produced by a stage constitute the inputs of the next stage.

5.1.4 | Modelling the MIPS SMA at the instruction level

The state produced by the different stages of the MIPS sequential microarchitecture during the instruction execution is modelled by the \( \text{sma} \) function which is shown in Figure 18. It is built as follows:

\[
\text{sma} :: W \rightarrow W
\]

\[
\text{sma} \ (\text{pc}, \text{dm}, \text{rf}) = \text{let}
\]

\[
(\text{ir}, \text{npc}) = f_1(\text{pc}, \text{dm}, \text{rf})
\]

\[
(\text{cop}, \text{rs}, \text{rt}, \text{rd}, \text{im}, \text{ra}, \text{rb}, \text{npc}) = f_2(f_1(\text{pc}, \text{dm}, \text{rf}))
\]

\[
(\text{rd}, \text{aluout}, \text{cond}, \text{npc}) = f_3(f_2(f_1(\text{pc}, \text{dm}, \text{rf})))
\]

\[
(\text{rd}, \text{aluout}, \text{cond}, \text{npc}) = f_4(f_3(f_2(f_1(\text{pc}, \text{dm}, \text{rf}))))
\]

\[
\text{rf}^* = f_5(f_4(f_3(f_2(f_1(\text{pc}, \text{dm}, \text{rf})))))
\]

**FIGURE 18** Functional Model capturing the Instruction level

| TABLE 2 | Informal description of the MIPS MA stages |
| --- | --- |
| Fetch stage [ \( F_1 = \text{fetch} \) ] | Decode stage [ \( F_2 = \text{decode} \) ] | Execute stage [ \( F_3 = \text{execute} \) ] | Mem-Access stage [ \( F_4 = \text{Maccess} \) ] | Write-Back stage [ \( F_5 = \text{Wback} \) ] |
| Fetch instruct. \( IR = \text{1mem}[\text{pc}] \) Increment PC \( \text{Npc} = \text{pc} + 4 \) | | Case instruction of add Aluout = ra + rb Load Aluout = ra + Im Store Aluout = ra + Im Beq Aluout = Npc + Im Cond = (ra op rb) | | |
| | | | | |
| TABLE 3 | Functional implementation of the MIPS SMA stages |
| --- | --- | --- | --- | --- |
| \( F_1 = \text{Fetch} \) | \( F_2 = \text{Decode} \) | \( F_3 = \text{Execute} \) | \( F_4 = \text{M-Access} \) | \( F_5 = \text{W-back} \) |
| Fetch(\( \text{pc}, \text{dm}, \text{rf} \)) = \[ \text{Let} \ (\text{ir}, \text{npc}) = \text{read pc four Zero} \] \( \text{In} \ (\text{ir}, \text{npc}) \) | \[ \text{Let} \ (\text{ir}, \text{npc}) = \text{read pc four Zero} \] \[ \text{In} \ (\text{ir}, \text{npc}) \] \( \text{decode}(\text{ir}, \text{rs}, \text{rt}, \text{rd}, \text{im}) = \text{decode}(\text{ir}) \) | \( \text{Execute}(\text{cop}, \text{rd}, \text{aluout}, \text{cond}, \text{npc}) = \text{let} \) \( \text{Maccess}(\text{rd}, \text{aluout}, \text{cond}, \text{npc}) = \text{let} \) \( \text{writeback}(\text{rd}, \text{aluout}, \text{ln}) = \text{let} \) | \[ \text{let} \ (\text{ir}, \text{npc}) = \text{read pc four Zero} \] \( \text{In} \ (\text{ir}, \text{npc}) \) |
\[ \text{sma} = [f_1, f_2 \circ f_1, \ldots, f_5 \circ f_4 \circ f_3 \circ f_2 \circ f_1] \]

where \( f_1 = \text{fetch}, f_2 = \text{Decode}, f_3 = \text{Execute}, f_4 = \text{MAccess}, f_5 = \text{WBack} \). Each stage function is activated during one clock cycle. Throughout the execution of an instruction, we could selectively capture any intermediate component state to be observed and compared.

5.1.5 Modelling the MIPS SMA at the program level

The functional model that captures the evolution of the MIPS SMA state at the program level (\( n \) instructions) is drawn as a state function in Figure 19.

5.1.6 Selecting the observables

To be realistic, let us consider only three component states to be observed: the Program counter state at the fetch stage, the memory state at the memory access stage and the registerfile state at the write back stage. The registerfile is directly computed by the function \( f_5 \), while the Pc and the Mr will be selected using the \( \text{proj} \) function.

\[
\text{projPc} :: Wstate \rightarrow Wpc, \text{projPc}(pc, dm, rf) = pc.
\]

\[
\text{ProjMr} :: Wstate \rightarrow Wram, \text{ProjMr}(pc, dm, rf) = dm.
\]

These three observables are computed and captured at the instruction level and the program level as shown in Figure 20a and b, respectively.

5.2 Modelling the MIPS pipelined microarchitecture

5.2.1 MIPS pipelined microarchitecture

The MIPS pipelined microarchitecture extends the MIPS sequential multicycle one by adding pipeline registers and an appropriate control logic to clock the different stages in parallel, as shown in Figure 21.

5.2.2 PMA higher order implementation

The function modelling the pipelined MA design will be parameterised with the stage functions \( f_i \) of the MIPS sequential microarchitecture, as shown in Figure 22.

The functions \( f_i \) describing the internal structure of the stages operate in parallel. At each cycle, each one returns a partial result. Thereafter, we could capture any component state. At this level, the stage functions are polymorphic and therefore, the microarchitecture model is technology independent.

5.2.3 PMA first-order implementation

The PMA partial first-order implementation that describes the behaviour of the MIPS pipelined microarchitecture is represented in Figure 23, by the FPMA function which is defined as follows:

\[
\begin{align*}
\text{FPMA} :: (\text{Int}, \text{State}) & \rightarrow \text{State}, \\
\text{FPMA} & = \text{PMA} f_1 f_2 f_3 f_4 f_5 \\
\text{FPMA} (k=5, pc, dm, rf) & = (pc, dm, rf)
\end{align*}
\]

5.3 Verifying the MIPS pipelined microarchitecture

5.3.1 TSMP model

The FPMA model will be verified against the TSMP model which is drawn in Figure 24.

5.3.2 Correctness Criterion

Let us limit the proof to three observables: The Program counter (pc), the data memory (dm) and the registerfile (rf).
Thus, proving the behavioural equivalence of these two first-order models requires proving the following equation.

\[ \forall k :: \text{Int}, pc :: \text{Wpc}, dm :: \text{Wram}, rf :: \text{Wrf} \]
\[ \text{FPMA}(k, pc, dm, rf) = \text{TSMP}(k, pc, dm, rf) \]

Such equation decomposes systematically to the following equations:

\[ \text{projPc}(f_1(\text{FPMA}((k-1), pc, dm, rf))) = \text{projPc}(f_1(\text{SMA}((k-1)*5, pc, dm, rf))) \]
\[ ^\land \text{projMr}(f_2(\text{FPMA}((k-1), pc, dm, rf))) = \text{projMr}(f_2(f_3(f_1(\text{SMA}((k-4)*5, pc, dm, rf))))) \]
\[ ^\land f_3(\text{FPMA}((k-1), pc, dm, rf)) = f_3(f_4(f_5(f_1(\text{SMA}((k-5)*5, pc, dm, rf))))) \]

Such equations involve only first-order types and so, are easily provable by induction over clock cycles.

5.4 Modelling the MIPS Dual-Issue Microarchitecture

5.4.1 MIPS Dual-Issue Microarchitecture

The dual-issue MIPS processor uses the same five-stage structure which is used by the single issue pipelined MIPS processor. Instructions are paired and aligned on a 64 bit length: It fetches a 64-bit Instruction, decodes two individual instructions (2*32 bit instruction), executes two operations, accesses memory to load/store 64-bit word, and finally writes back two results. Its microarchitecture is shown in Figure 25, and a modular representation of such microarchitecture is depicted in Figure 26.

5.4.2 SSMA higher order implementation

The SSMA function modelling the dual-issue microarchitecture will be parameterised with the PMA function which will be distributed over two identical pipelines operating in parallel, as shown in Figure 27.

5.4.3 SSMA first-order implementation

The partial first-order implementation of the MIPS dual issue is represented by the FSMA function which is derived from the higher order SSMA function as follows:
Let \( \text{FSMA} = \text{SSMA.pma} :: (\text{Int, Int, } W) \rightarrow W \)
\[
\text{fpma} = \text{pma } f_1 f_2 f_3 f_4 f_5 :: (\text{Int, } W) \rightarrow W
\]

We get the following definition which is represented in Figure 28.

5.5 | Verifying the MIPS dual issue microarchitecture

5.5.1 | TSMS model

The FSMA first-order model which represents the behaviour of the MIPS dual-issue MA will be verified against the TSMS first-order model which is shown in Figure 29.

5.5.2 | Correctness Criterion

Again, let us limit the observation only to the component states \((\text{pc, dm, rf})\), that have been considered for the proof of the MIPS PMA. Thus, proving the correctness of the FSMA model against the TSMS model with respect to these observables requires proving the following equation:

\[
\forall k :: \text{Int}, \forall (\text{pc}_1, \text{dm}_1, \text{rf}_1), (\text{pc}_2, \text{dm}_2, \text{rf}_2) :: \text{Address, dm1, dm2 :: Memory, rf1, rf2 :: RegFile}
\]

\[
\text{FSMA}(2,k,((\text{pc}_1, \text{dm}_1, \text{rf}_1), (\text{pc}_2, \text{dm}_2, \text{rf}_2))) =
\]
\[
\text{Let } (\text{pc}', \text{dm}', \text{rf}') = \text{pma } f_1 f_2 f_3 f_4 f_5 ((k-1), \text{pc}_1, \text{dm}_1, \text{rf}_1))
\]
\[
(\text{pc}', \text{dm}', \text{rf}''') = \text{pma } f_1 f_2 f_3 f_4 f_5 ((k-1), \text{pc}_2, \text{dm}_2, \text{rf}_2))
\]
\[
\text{ln } ((\text{pc}', \text{dm}', \text{rf}'), (\text{pc}', \text{dm}', \text{rf}'''))
\]

\[
\text{TSMS}(2,k,((\text{pc}_1, \text{dm}_1, \text{rf}_1), (\text{pc}_2, \text{dm}_2, \text{rf}_2))) =
\]
\[
\text{Let } (\text{pc}', \text{dm}', \text{rf}') = \text{tsmp}((k-1), \text{pc}_1, \text{dm}_1, \text{rf}_1))
\]
\[
(\text{pc}', \text{dm}', \text{rf}''') = \text{tsmp}((k-1), \text{pc}_1, \text{dm}_2, \text{rf}_2))
\]
\[
\text{ln } ((\text{pc}', \text{dm}', \text{rf}'), (\text{pc}', \text{dm}', \text{rf}'''))
\]
The proof of such equation decomposes to the proof of the following equations:

\[ \text{fpm} (k, (pc_1, dm_1, rf_1)) = \text{tmp}(k, (pc_1, dm_1, rf_1)) \]
\[ \text{fpm} (k, (pc_2, dm_2, rf_2)) = \text{tmp}(k, (pc_2, dm_2, rf_2)) \]

By substituting both fpm and tmp by their definitions, the above equations could further decompose to the following equations:

--- Pipeline-1 ---
\[ \text{projPc}(\text{f1}((\text{FPMA}(k-1), pc_1, dm_1, rf_1))) = \text{projPc}(\text{f1}((\text{SMA}((2^*(k-1)*5 + 1), pc_1, dm_1, rf_1))) \]
\[ \text{projMr}(\text{f4}(\text{FPMA}(k-1), pc_1, dm_1, rf_1))) = \text{projMr}(\text{f4}(\text{fpma}(k, (\text{pc} _1, dm_1, rf_1)))) \]
\[ \text{f}_5((\text{FPMA}(k-1), pc_1, dm_1, rf_1))) = \text{f}_5((\text{fpma}(k, (\text{pc} _1, dm_1, rf_1)))) \]

--- Pipeline-2 ---
\[ \text{projPc}(\text{f1}((\text{FPMA}(k-1), pc_2, dm_2, rf_2))) = \text{projPc}(\text{f1}((\text{SMA}((2^*(k-1)*5 + 1), pc_2, dm_2, rf_2))) \]
\[ \text{projMr}(\text{f4}(\text{FPMA}(k-1), pc_2, dm_2, rf_2))) = \text{projMr}(\text{f4}(\text{fpma}(k, (\text{pc} _2, dm_2, rf_2)))) \]
\[ \text{f}_5((\text{FPMA}(k-1), pc_2, dm_2, rf_2))) = \text{f}_5((\text{fpma}(k, (\text{pc} _2, dm_2, rf_2)))) \]

Now, we can use the definitions of the FPMA and the SMA models developed so far, to easily solve such equations which involve only first-order types. A detailed proof by induction over clock cycles is given in Ref. [27].

6 | CONCLUSION

A methodological design approach based on functional techniques for the formal modelling and verification of digital circuits using higher order functions has been presented. The approach exploits both the mechanism of parameterisation and the notion of partial application to incrementally develop both the design and its proof in a systematic way. With respect to alternative approaches, the proposed proof approach brings many contributions, particularly:

- It develops accurate functional models useful for both formal verification and simulation (actual designs combine both techniques).
- It scales well with the design complexity while it abstracts from its implementation details. Hence, more complex designs could be straightforwardly tackled.
- It reduces the design proof to the first-order case by simple partial application. Thereafter, it refines such proof (down till the component level) into a set of verification properties which are separately provable. Consequently, we can limit the proof only to some observations in which we are interested. This is very helpful for debugging complex designs. Furthermore, it carries out the proof process by induction within the same specification framework that avoids translating the design properties between different semantics (from the specification framework to the proof framework) which remains very difficult.

- It does not require a data abstraction function for the verification of pipelined and superscalar microarchitectural designs which remains difficult to construct (its complexity increases with the increase of the design complexity). Only a time abstraction function is needed to map between the models involved in the proof.
- Furthermore, the stalling case (due to hazards) is captured by the proof as well.
- However, a functional graph rewriting framework such as Clean [41] is more adequate for mechanical reasoning than a purely functional framework such as Haskell.

ORCID
Salah Merniz  https://orcid.org/0000-0001-6544-7063

REFERENCES
1. Luk, W.W., Jones, G.: From specifications to parameterised architectures. In: Proc. IFIP WG10. vol. 2, Glasgow July (1988)
2. Wester, R.: A transformation-based approach to hardware design using higher-order functions, Ph.D. thesis. University of Twente (2015)
3. Sheeran, M.: uFP: an algebraic VLSI design language, Ph.D. thesis. Oxford University (1985)
4. O’Donnell, J.: Hydra : hardware description in a functional language using recursion equations and higher-order combining forms. In: Milne, G. J. (ed) The Fusion of Hardware Design and Verification. North-Holland, pp. 309–328 (1988)
5. Baaij, C.P.R.: Digital circuits in CλASH: functional specifications and type-directed synthesis. Ph.D. thesis. University of Twente (2014)
6. Prabhakar, R., et al.: Plasticine, A reconfigurable architecture for parallel patterns. In: Proceedings of the ACM/IEEE 44th International Symposium on Computer Architecture, ISCA’17, pp. 389–402. Canada, Toronto, June (2017). https://doi.org/10.1145/3079856.3080256
7. Prabhakar, R., et al.: Generating configurable hardware from parallel patterns. In: Proceedings of the 21st Int. Conf. on architectural support for programing languages and operating systems, ASPLOS-2016 ACM, New York, pp. 651–665. (2016)
8. Kristien, M., Bodin, B.: High-Level Synthesis of Functional Patterns with Lift. In: Proc. ARAY’2019, vol. 22, pp. 35–45. June, Phoenix (2019)
9. Astesiano, E., Cerioli, R.: Relationships between logical frames. In: Recent Trends in Data Type Specification, vol. 655, pp. 126–143. Springer LNCS (1991)
10. Blanchette, J.C., Nipkow, T., Nitpick: A counter-example generator for Isabelle/HOL based on the relational model Finder Kodkod. ITP, LNCS. 6172, 131–146 (2010)
11. Callaway Burn, T., Luke Ong, C.H., Ramsey, S.: Higher-order constrained horn clauses for verification. In: Proc. of the ACM on Programming Languages, POPL, vol. 2, January., pp. 1–28. (2018)
12. Luke Ong, C.H., Ramsey, S.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: Proc. of POPL, pp. 587–598. ACM (2011)
13. Giesl, J., et al.: Automated termination proofs for Haskell by term rewriting. ACM Trans. Program. Lang. Syst. 33(21–7), 7–39 (2011)
14. Asada, K., Sata, R., Kobayashi, N.: Verifying Relational properties of functional programs by first-order refinement. In: Proc. of PEPM’15, ACM, pp. 61–72 (2015)
15. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Proceedings of SCALA’15, ACM, pp. 18–29 (2015)
16. Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Programming Languages and Systems, ESOP’13, Springer-LNCS 7792, Rome, Italy, March, pp. 209–228. (2013)
17. Jhala, R., et al.: HMC: verifying functional programs using abstract interpreters. CAV 2011, pp. 470–485, Springer LNCS 6806 (2011)
18. Neatherway, R.P., et al.: A traversal-based algorithm for higher-order model checking. In: Proc. 17th ACM SIGPLAN International Conference on Functional Programming, ICFP’12, pp. 353–364 (2012)
19. Braibant, T.: Coquet: a Coq Library for verifying hardware. In: Certified Programs and Proofs, Springer LNCS (7086), pp. 330–345.2011
20. Kanig, J., et al.: In: Proc. ACM SIGPLAN Workshop on ML, Edinburgh, August, pp. 39–48. (2009)
21. Giesel, J., et al.: Automated termination proofs with AProVE. In: International Conference, RTA 2004, Aachen, LNCS, 3091, pp. 210–220, 3–5 June (2004)
22. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: Proc. POPL13, Rome, pp. 75–86, 23–25 Jan (2013)
23. Uchevler, B.N., Svarstad, K.: Modelling and assertion-based verification of run-time reconfigurable designs using functional programming abstractions. International Journal of Reconfigurable Computing, Hindawi, pp. 1–25. (2018)
24. Procter, A., et al.: Semantics driven hardware design, implementation, and verification with ReWire. In: The 2015 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems. LCTES, Portland, Oregon, June 18–19 (2015)
25. Strauch, T.: Combining simulation and FPGA based verification to an affordable and ultra-fast multi-billion-gate verification system. Proc. RSP. 19, 22–28 (2019)
26. Grimm, T.: A Hybrid methodology to enable the verification of temporal properties at system-level. Ph.D. thesis. Ruhr-University, Bochum (2018)
27. Merniz, S., Benmohammed, M.: Formal verification of superscalar micro-architectures: functional approach. Int. J. Circ. Syst. Signal Process. NAUN J. 2(1), 56–65 (2008)
28. Haskell.org, Haskell. Report www.haskell.org/haskellwiki/ (2010)
29. Zhai, K., et al.: Hardware synthesis from a recursive functional language. Proc. CODES’15, 83–93 (2015)
30. Ablak, A.B., Damaj, I.: HTCC: Haskell to Handel-C Compiler. In: The 19th EUROMICRO Conference on Digital System Design. IEEE, Limassol, Cyprus, August 31–September, 192–199 (2016)
31. O’Donnell, J., From transistors to computer architecture: teaching functional circuit specification in Hydra. In: FPLE’95: Symposium on Functional Programming Languages in Education, Springer, vol. 1022, pp. 195–214. LNCS
32. Burch, J., Dill, D.: Automatic verification of pipelined microprocessor control. In: Proc. CAV’94, Springer-LNCS 818, pp. 68–80. (June 1994)
33. Velev, M., Bryant, R.: Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. SAVE Proc. 112–117 (2000)
34. Reid, A., et al.: End-to-end verification of ARM processors with ISA-formal. In: Computer Aided Verification, Canada, Toronto, pp. 17–23 (July 2016)
35. Hosabetu, R.M., Sivas, M., Gopalakrishnan, G.: Decomposing the proof of correctness of pipelined microprocessors. In: Proc. CAV’08, LNCS 1427, Springer Verlag, pp. 122–134 (June 1998)
36. Fox, A.C.J.: An algebraic framework for verifying the correctness of hardware with input and output: a formalization in HOL. In: CALCO, LNCS 3629(2005), 1–29 (2005)
37. Manolios, P.: Correctness of pipelined machines. In: Proc. FMCAD'00, Hunt Jr., W.A., Johnson, D.S. (eds.), vol. 1054, pp. 161–178. Springer-LNCS (2000)
38. Andraus, Z.S.: Automatic formal verification of control logic in hardware designs, Ph.D. thesis. University of Michigan (2009)
39. Merniz, S., Benmohammed, M.: A methodology for the formal verification of RISC microprocessors: a functional approach. In: Proc. ACS/ IEEE, AICCSA’07, Amman, Jordan, pp. 492–499 (May 2007)
40. Mondelli, A.: Revisiting wide superscalar microarchitectures, Ph.D. thesis. University of Rennes1 (Sept. 2017)
41. Clean Report: Version 2.2, 2011. https://clean.cs.ru.nl/Clean

How to cite this article: Merniz S, Harous S. Modelling and verification of parameterized architectures: A functional approach. IET Comput. Digit. Tech. 2021;1–14. https://doi.org/10.1049/ietcdt.12024