Modelling and Verification of Superscalar Micro-architectures
Functional Approach

S. MERNIZ, and M. BENMOHAMMED
LIRE Laboratory
Computer Science Department
Mentouri University, Constantine
ALGERIA

Abstract: Most proof approaches verified a pipelined Micro-Architectural (MA) implementation against an Instruction-Set-Architecture (ISA) specification, and consequently, it was impossible to find a meaningful point where the implementation state and the specification state can be compared easily. An alternative solution to such problem is to verify a pipelined micro-architectural implementation against a sequential multi-cycle implementation. Because both models are formalised in terms of clock cycles, all synchronous intermediate states represent interesting points where the comparison could be achieved easily. Furthermore, by decomposing the state, the overall proof decomposes systematically into a set of verification conditions more simple to reason about and to verify. A major advantage of this elegant choice is the ability to carry out the proof by induction within the same specification language rather than by symbolic simulation through a proof tool which remains very tedious. Also, because both models relate to the MA level, there is no need for a data abstraction function, only a time abstraction function is needed to map between the times used by the two models. The potential features of the proposed proof methodology are demonstrated over the pipelined and the superscalar pipelined MIPS processors within Haskell framework.

Key-Words: Formal specification, formal Verification, RISC designs, Micro-architectures, State functions, Haskell.

1 Introduction
1.1 Motivation
Most proof approaches attempt to validate processor micro-architectural implementations against their corresponding ISA specifications. However, if a sequential MA implementation (which reveals the state after completing each instruction) could be easily verified against an ISA specification through a commutative diagram, this is not the case for a pipelined MA implementation because of latency of pipeline events. At any time, there may be several partially executed instructions in the pipe, that make it difficult to define a data abstraction function to map the partial results into a meaningful visible state. In other words, it is impossible to find a meaningful point where the comparison between the pipelined MA implementation and the ISA specification can be made easily. Burch and Dill [1] solved such problem by simulating the effect of completing every instruction in the pipe before doing the comparison. So, the natural way to complete every instruction is to flush the pipe. After flushing, they project the synchronised implementation state to the specification state to extract only the observables. In their original work, they proved the pipeline correctness diagram by symbolically simulating the pipelined machine design in their logic of uninterpreted functions with equalities.

Although the flushing method enhanced verification techniques by using an automated decision procedure, it presents on the other hand many drawbacks which are clearly stated in many papers [2, 3]. Particularly, it makes the size of the abstraction function and the number of examined cases very large for deeper pipelines. The technique has been extended thereafter by many researchers to handle more complex designs such as superscalar [3, 4], and Out Of Order execution [5, 6] designs. Unfortunately, the same correctness criterion (proving the commutative diagram with respect to an ISA specification) has been adopted by the extenders, and consequently the same drawbacks persist. Moreover, as new implementation features are introduced, such variants are flawed. Other notions of correctness such as the one step theorem [7, 8] and Well-founded Equivalence Bisimulation [9], also, have been used to verify complex processor designs. Both approaches prove the commutative diagram with respect to an ISA specification.

This work suggests verifying a pipelined implementation against a sequential multi-cycle implementation rather than against an ISA specification. Because both models are formalised in terms of clock cycles, all synchronous intermediate states represent useful points where the comparison between the two models could be achieved easily. Furthermore, because
both models relate to the MA level, there is no need for a data abstraction function (which remains very difficult to define for most approaches), only a time abstraction function is needed to map between the times used by the two models. One positive consequence of this elegant choice is the ability to carry out the proof by induction within the same specification language rather than by symbolic simulation through a proof tool which remains very tedious.

To practically show the usefulness of our approach, we have applied it to RISC processors within a functional framework. RISC architectures are well structured and so, they can be hierarchically built from the core architecture implementing the basic instruction set to highly optimised architectures [10]. Therefore, they suit elegantly the incremental design approach. On the other hand, functional frameworks provide beside their formal semantics definition (to support formal reasoning), powerful features (function composition, parallelism, polymorphism, higher order functions, etc) that demonstrated their viability with respect to complex hardware designs [11, 12].

1.2 Design Approach

Our view of formal verification of microprocessors follows the vertical-horizontal layered design approach depicted in figure.1. 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: It describes the structural features of the microarchitectures implementing the processor’s operations. All MA designs (which could be hierarchically built one over the other) represent different implementations for the same ISA specification. In this work, we will be interested on three MA designs; the Sequential MA design (SMA), the pipelined MA design (PMA), and the superscalar MA design (SSMA). The SMA design whose proof could be easily performed against an ISA specification1 represents the reference core architecture over which will be hierarchically developed both the PMA and the SSMA designs, and against which will be verified as well (unlike other approaches where the PMA and the SSMA designs are proved against an ISA specification). The lower layers represent successive refinements.

In our context, all MA designs will be modelled in terms of state functions (representing state machines), within a functional framework using the functional programming language Haskell [13].

---

1In [14], we performed the correctness proof of the sequential multi-cycle MIPS processor till the component level

---

2 Preliminaries

2.1 State Function

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:

$$ F :: (Int, S) \rightarrow S $$

$$ F(0, c) = c $$

$$ F((n+1), c) = f(F(n,c)) $$

Because the next state is always a function of the previous state, a system modelled by the notion of state function is deterministic. The transition between two adjacent observable states is called a step. For instance, $F(n,c)$ represents the state after $n$ steps, given an initial state $c$, and a next-state function $f$. Its value is given by: $F(n,c)=f^n (c)$

2.2 State decomposition

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

Let $S=(S_1,\ldots,S_k)$ 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 and the next-state functions will be decomposed as follows:

$$ F(n, c_1, \ldots, c_k) = (F_1(n, c_1, \ldots, c_k), \ldots, F_k(n, c_1, \ldots, c_k)) $$

And

$$ f_i(c_1, \ldots, c_k) = (f_{i_1}(c_1, \ldots, c_k), \ldots, f_{i_k}(c_1, \ldots, c_k) $$

where,

$$ F_i :: (Int, S) \rightarrow S_i $$

$$ F_i(0, c_1, \ldots, c_k) = c_i $$

$$ F_i(n, c_1, \ldots, c_k) = f_{i_1}(F_1(n-1, c_1, \ldots, c_k), \ldots, F_k((n-1), c_1, \ldots, c_k)) $$

And $f_i :: S \rightarrow S_i$, for $1 \leq i \leq k$

In this way, each coordinate $F_i$ computes only the $i^{th}$ component of the state function $F$, and each coordinate $f_i$ computes only the $i^{th}$ component of the next-state function $f$. \[\Box\]
2.3 The observational aspect of the state function

Redefining $F$, as follows:

\[ F_i (n, c_1, ..., c_k) = f_i (F_i ((n-1), c_1, ..., c_k), ..., F_k ((n-1), c_1, ..., c_k)) \]

\[ = f_i (F ((n-1), c_1, ..., c_k)) \]

Then,

\[ F(n, c_1, ..., c_k) = (F_1 (n, c_1, ..., c_k), ..., F_k ((n-1), c_1, ..., c_k)) \]

\[ = (f_1 (F ((n-1), c_1, ..., c_k)), ..., f_k (F ((n-1), c_1, ..., c_k))) \]

Taking the initial state into account, $F$ will be redefined more precisely as follows:

\[ F : (\text{Int}, S) \rightarrow S \]

\[ F(0, c_0^1, ..., c_0^k) = (c_0^1, ..., c_0^k) \]

\[ F(n, c_0^1, ..., c_0^k) = \text{let } c_n^1 = f_1 (F ((n-1), c_0^1, ..., c_0^k)) \]

\[ c_k^k = f_k (F ((n-1), c_0^1, ..., c_0^k)) \]

\[ \text{in } (c_n^1, ..., c_k^k) \]

Rewriting $F$ in such a form reveals many important advantages, in particular:

- It suits naturally the parallel computations: All the $f_i$ coordinates operate in parallel.
- It fits adequately the notion of observational equivalence (very useful for complex systems, where someone is interested to just some observations among many others)
- It fits also the incremental design approach: If we extend the design by extra observables, we just have to define extra next-state functions.

3 Modelling the MA-Step

At the micro-architectural level the notion of step, called MA-step, will be implemented in terms of clock cycles. To be able to observe the evolution of the state at each cycle, the MA-Step function will be decomposed as follows

\[ ma = [f_s, f_2 \circ (f_1, ..., f_i^1), ..., f_i \circ ... \circ (f_i, ..., f_i^i)] \] (m1)

In such form, only the $f_i$ coordinates are transformers, while all others are selectors (to read from one stage interface and write into the next). In this way, all the component states which are computed by the $f_i$ coordinates throughout the different stages are captured as depicted in figure.2. To be realistic, we have limited the observation to only one observable by stage. For example, the multi-cycle MIPS machine [15] updates the PC state at fetch stage, the memory state at memory access stage, and the register file state at write back stage. A functional implementation is shown in figure 3

4 Modelling the Sequential MA machine

A sequential MA machine will be defined by a recursive state function that returns the MA state after executing $n$ instructions (by applying MA-step $n$ times).

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

\[ SMA(0, c_0^1, ..., c_0^k) = (c_0^1, ..., c_0^k) \]

\[ SMA(n, c_0^1, ..., c_0^k) = ma(SMA((n-1), c_0^1, ..., c_0^k)) \]

By infolding the $ma$ function, the SMA definition rewrites as follows

\[ SMA(n, c_0^1, ..., c_0^k) = \]

\[ \text{let } c_n^1 = f_1 (SMA((n-1), c_0^1, ..., c_0^k)) \]

\[ ... \]

\[ c_n^k = (f_i \circ ... \circ (f_i, ..., f_i^i)) (SMA((n-1), c_0^1, ..., c_0^k)) \]

\[ \text{in } (c_n^1, ..., c_n^k) \]

5 Modelling the Pipelined MA Machine

Because the instruction level is not observable (instructions are overlapped), the PMA model will be formalised at the program level but still in terms of clock cycles. It starts naturally from a flushed state, fills progressively the pipe and then proceeds interminably (unlike the flushing approach), as depicted in figure 5.
Again, the key solution for constructing the PMA model consists of decomposing the PMA state. Let $S$, be the number of pipelining stages, $f_i$, for $1 \leq i \leq S$, be the component function that performs the functionality of the stage $i$, and \( W= (W_1, ..., W_s) \), be the PMA state distributed over $S$ observables. Therefore, the construction of the PMA model consists of two steps:

- The first step is an irregular computation: It allows to progressively filling the pipe till cycle $S-1$. Thus, given an initial state: $C_0$, the state at cycle $S-1$, is computed as follows:\

\[
PMA((s-1), C_0, ..., C_0) = (f_0 \circ \cdots \circ f_s) (C_0, ..., C_0) = C_{s-1}
\]

- The second step which starts from the cycle $S$, is a regular computation: It allows to recursively compute the PMA state by repeatedly applying the next state function: $f=(f_1, \ldots, f_s)$, which establishes automatically after $S$ cycles. So, the PMA state at cycle $k \geq S$, is computed as follows:

\[
PMA(k, C_0, ..., C_0) = (f_1, ..., f_s) (PMA((k-1), C_0, ..., C_0))
\]

The functional implementation of this regular form is shown in figure 6.

\[
PMA(k, C_0, ..., C_0) = \\
\text{let } C_1 = f_1 (PMA((k-1), C_0, ..., C_0)) \\
\vdots \\
\text{let } C_s = f_s (PMA((k-1), C_0, ..., C_0)) \\
\text{in } (C_1, ..., C_s)
\]

6 Verification of the Pipelined MA Machine

6.1 Synchronisation diagram

Because both the PMA and the SMA models are formalised in terms of clock cycles, all synchronous intermediate states represent useful points where the comparison could be achieved easily. Indeed, at the end of each clock cycle, a PMA design with $S$ stages reveals $S$ partial results; each one relates to an instruction within the pipe. So, we can construct a variant of the SMA model - called Component SMA Model - which simulates the effect of computing the same results sequentially as shown in figure 7.

In case of no stalls, the synchronization is performed using the following time function: $t_n(k,j) = (k-j)*s + j$ (11) This means that we need $(k-j)*S$ clock cycles to execute $(k-j)$ instructions sequentially by the SMA model, and we need $j$ clock cycles over, to reach the desired sequential state.

In case of stalls, the time function rewrites as follows: $t_n(k,j,e) = ((k-j-e)*s + j$ (12), where $e$, is the number of stalls.

6.2 CSMA Model for a pipelined MA design

The CSMA model that we propose here, inputs the same clock cycle $k$, as the PMA model, unlike the SMA model which inputs the number of instructions to execute (see sect 4). For each clock cycle $k \geq S$, it constructs $S$ terms (upon the SMA model); each one computes a partial result for one instruction within the pipe as shown in figure 8.

\[
\text{CSMA}(k, C_0, \ldots, C_0) = \\
\text{let } C_1 = f_1 (\text{SMA}(k-1), C_0, \ldots, C_0) \\
\vdots \\
\text{let } C_s = f_s (\text{SMA}(k-1), C_0, \ldots, C_0) \\
\text{in } (C_1, \ldots, C_k)
\]

6.3 Correctness criterion

Proving the correctness of the PMA model with respect to the CSMA model requires proving the following equation:

\[
\forall k :: \text{Int, } \forall C_0 :: W_1, ... C_0 :: W_s \\
PMA(k, C_0, C_0, \ldots, C_0) = \text{CSMA}(k, C_0, C_0, \ldots, C_0)
\]

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

\[
f_1 (PMA((k-1), C_0, \ldots, C_0)) = f_1 (\text{SMA}(k-1), C_0, C_0, \ldots, C_0) \quad (\text{el})
\]

\[
\land f_s (PMA((k-1), C_0, \ldots, C_0)) = (f_s \circ \cdots \circ f_s) (\text{SMA}(k-1), C_0, \ldots, C_0) \quad (\text{es})
\]

6.4 Discussion

- The above equations are separately provable by induction over clock cycles. This avoids the use of symbolic evaluation which remains very tedious and insufficient for complex designs [16].
8 Verification of the Superscalar MA Machine

8.1 Synchronisation diagram

The synchronisation diagram for a SSMA model generalises the synchronisation diagram used for the PMA model. Fig. 11 depicts such synchronisation for a superscalar design involving an arbitrary number of pipelines.

Let \( c_{\text{SSMA}}^{i,j} = f_{\text{CSSMA}}((k-1), c_{0}^{i,1}, ..., c_{0}^{i,n}) \), be the SSMA component state produced by the stage function \( f_{j} \), of the pipeline \( i \), at clock cycle \( k \geq S \).

In case of no stalls, the time function is defined as follows:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1))*s + e
\]

where \( n \) is the number of pipelines

The corresponding sequential state is computed as follows:

\[
s_{i}(k,j,i) = (f_{j} \circ \circ \circ f_{1})(\text{SSMA}(n*(k-j)+(i-1), c_{0}^{i,1}, ..., c_{0}^{i,n}))
\]

In case of stalls, the synchronisation is performed using the following time function:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1)-e)*s + j
\]

where \( e \) is the number of stalls

8.2 CSMA model for a superscalar MA design

The component sequential model for a superscalar design will be built over the CSMA model used for a pipelined design. We call it; CSSMA model. It inputs the same parameters as the SSMA model, and outputs the expected state against which the SSMA model will be compared. Figure 10 shows such model for \( k \geq S \).

The synchronisation diagram for a SSMA model generalises the synchronisation diagram used for the PMA model. Fig. 11 depicts such synchronisation for a superscalar design involving an arbitrary number of pipelines.

In case of no stalls, the time function is defined as follows:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1))*s + e
\]

where \( n \) is the number of pipelines

The corresponding sequential state is computed as follows:

\[
s_{i}(k,j,i) = (f_{j} \circ \circ \circ f_{1})(\text{CSSMA}(n*(k-j)+(i-1), c_{0}^{i,1}, ..., c_{0}^{i,n}))
\]

In case of stalls, the synchronisation is performed using the following time function:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1)-e)*s + j
\]

where \( e \) is the number of stalls

The synchronisation diagram for a SSMA model generalises the synchronisation diagram used for the PMA model. Fig. 11 depicts such synchronisation for a superscalar design involving an arbitrary number of pipelines.

In case of no stalls, the time function is defined as follows:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1))*s + e
\]

where \( n \) is the number of pipelines

The corresponding sequential state is computed as follows:

\[
s_{i}(k,j,i) = (f_{j} \circ \circ \circ f_{1})(\text{CSSMA}(n*(k-j)+(i-1), c_{0}^{i,1}, ..., c_{0}^{i,n}))
\]

In case of stalls, the synchronisation is performed using the following time function:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1)-e)*s + j
\]

where \( e \) is the number of stalls

The synchronisation diagram for a SSMA model generalises the synchronisation diagram used for the PMA model. Fig. 11 depicts such synchronisation for a superscalar design involving an arbitrary number of pipelines.

In case of no stalls, the time function is defined as follows:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1))*s + e
\]

where \( n \) is the number of pipelines

The corresponding sequential state is computed as follows:

\[
s_{i}(k,j,i) = (f_{j} \circ \circ \circ f_{1})(\text{CSSMA}(n*(k-j)+(i-1), c_{0}^{i,1}, ..., c_{0}^{i,n}))
\]

In case of stalls, the synchronisation is performed using the following time function:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1)-e)*s + j
\]

where \( e \) is the number of stalls

8 Verification of the Superscalar MA Machine

8.1 Synchronisation diagram

The synchronisation diagram for a SSMA model generalises the synchronisation diagram used for the PMA model. Fig. 11 depicts such synchronisation for a superscalar design involving an arbitrary number of pipelines.

In case of no stalls, the time function is defined as follows:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1))*s + e
\]

where \( n \) is the number of pipelines

The corresponding sequential state is computed as follows:

\[
s_{i}(k,j,i) = (f_{j} \circ \circ \circ f_{1})(\text{CSSMA}(n*(k-j)+(i-1), c_{0}^{i,1}, ..., c_{0}^{i,n}))
\]

In case of stalls, the synchronisation is performed using the following time function:

\[
\tau_{i}(k,j,i,e) = (n*(k-j)+(i-1)-e)*s + j
\]

where \( e \) is the number of stalls

8.2 CSMA model for a superscalar MA design

The component sequential model for a superscalar design will be built over the CSMA model used for a pipelined design. We call it; CSSMA model. It inputs the same parameters as the SSMA model, and outputs the expected state against which the SSMA model will be compared. Figure 10 shows such model for \( k \geq S \).
8.3 Correctness criterion

Proving the correctness of the SSMA model with respect to the CSSMA model requires proving the following equation:

\[ \forall \ n, k :: Int, \ \forall \ c^{11}_0, \ldots, c^{11}_n :: W^1, \ldots, c^{ln}_0 :: W^l, \ldots, c^{mn}_0 :: W^m \]

\[ \text{SSMA}(n, k, (c^{11}_0, \ldots, c^{11}_n)) \ldots (c^{ln}_0, \ldots, c^{mn}_0)) = \]

\[ \text{CSSMA}(n, k, (c^{11}_0, \ldots, c^{11}_n), \ldots, (c^{ln}_0, \ldots, c^{mn}_0)) \]

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

\[ \text{PMA}_A(k, c^{11}_0, \ldots, c^{11}_n) = \text{CSMA}_A(k, c^{11}_0, \ldots, c^{11}_n) \]

\[ \text{PMA}_A(k, c^{11}_0, \ldots, c^{11}_n) \land \text{PMA}_A(k, c^{ln}_0, \ldots, c^{mn}_0) = \text{CSMA}_A(k, c^{ln}_0, \ldots, c^{mn}_0) \]

Now, we can use the definitions of the PMA and the CSMA models given so far to resolve such equations.

9 Case Study

As a case study, we applied the proposed proof methodology to the formal verification of two examples: the pipelined and the dual issue superscalar pipelined MIPS processors with respect to the non-pipelined version. All functional models (PMA, SMA, SSMA, and CSMA models) were developed within Haskell framework. The correctness proof was carried out manually (the proof is amenable to mechanisation) by induction and was limited only to three observations: The PC, The Memory and the register file states. Therefore, three equations (each one relates to an observable) have been stated. For each case, three types of instructions (Register-type, Memory-type, and branch-type) have been reasoned about and proved. Throughout the proof process, different types of hazards (particularly branch hazards) were discussed as well. The methodology gives each time the right result. For each case the models are executed to compare the results.

10 Conclusions

A methodological approach for the formal specification and verification of RISC processor micro-architectures within a functional framework has been presented. The approach brings many contributions with respect to previous works. It produces accurate functional MA models (representing functional programs) that could be used for both formal verification and simulation (to compare the results). Moreover, by decomposing the state, the overall proof decomposes systematically into a set of verification conditions more simple to reason about and to verify. In particular, we can reason about the inter-instruction dependency such as the different types of hazards that can occur during the execution, unlike the flushing technique where such reasoning is impossible. Furthermore, it is possible to reason either about individual instructions or about groups of instructions such as register-instructions, memory-instructions and branch-instructions.

- Because both the reference and the pipelined models relate to the MA level, there is no need for a data abstraction function, only a time abstraction function is used to map between the times used by the two models. Moreover, such synchronization requires few cases with respect to those used by alternative approaches [2, 4].
- The ability to instantiate the set of equations for any particular architecture, offers a better scalability for the verification of future highly-optimised designs
- The key strength of the proposed proof methodology is the ability to carry out the proof by induction over clock cycles, within the same specification language rather than by symbolic evaluation through a proof tool which still requires considerable efforts.

References


