Formal Verification of Superscalar Micro-architectures: Functional approach

S. Merniz, and M. Benmohammed
LIRE Laboratory
Computer Science Department, Mentouri University, Constantine, ALGERIA
s merniz@hotmail.com

Abstract: **-** Verifying a pipelined Micro-Architectural (MA) implementation against an Instruction-Set-Architecture (ISA) specification is a common approach which still requires considerable efforts because there is no meaningful point where the implementation state and the specification state can be compared easily. An alternative approach consists of verifying 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 useful points where the comparison could be achieved easily. 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. 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. Furthermore, by decomposing the state, the overall proof decomposes systematically into a set of verification conditions more simple to reason about and to verify. The proposed proof methodology is illustrated on both the pipelined and the superscalar pipelined MIPS processors within Haskell framework.

Keywords: **-** Formal specification, Formal verification, Micro-architectures, State functions.

I. INTRODUCTION

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

Manuscript received March 31, 2008: Revised version received July 28, 2008.

S. Merniz is with Computer Science department, Mentouri University, Constatntine, Algeria. (Phone: +213 772 85 84 64, Email:s_merniz@hotmail.com).

M. Benmohammed is with computer Science Department, Mentouri University, Constantine, Algeria. (Email: ibnm@yahoo).

incremental design approach. On the other hand, functional frameworks provide beside their formal semantics definition (to support formal reasoning), powerful features (function composition, higher order functions, parallelism, polymorphism, etc) that demonstrated their viability with respect to complex hardware designs [11], [12],[13].

II. 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 micro-architectures 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 specification 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 language Haskell [14].

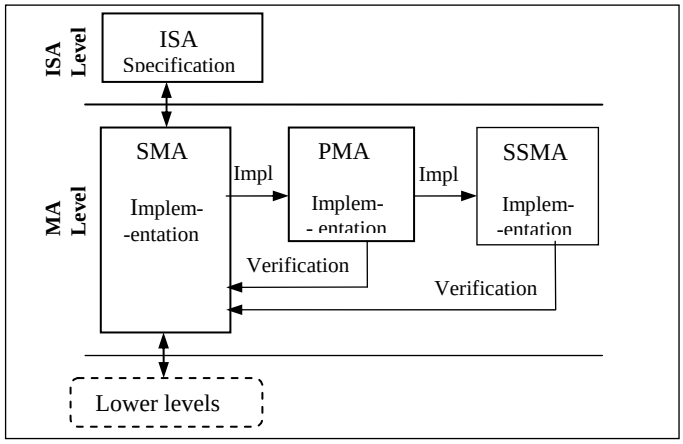

Fig.1.The layered Vertical-Horizontal design approach

III. PRELIMINARIES

A. 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: $F(n,c) = f^n(c)$

B. 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 $S = (S_1,...,S_k)$ be the state space distributed over k components (the observables) where S_i is the state of the i^{th} component, for $1 \le i \le k$. Thus, the state and the next-state functions will be decomposed as follows:

$F(n, c_1,.., c_k) = (F_1(n, c_1,.., c_k),....,F_k(n, c_1,.., c_k))$

And

$f(c_1,..., c_k) = (f_1(c_1,..., c_k),..., f_k(c_1,..., c_k))$

where,

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

$F_i(0, c_1,..., c_k) = c_i$

$F_i(n, c_1,.., c_k) = f_i(F_1((n-1), c_1,.., c_k),...,F_k((n-1), c_1,.., c_k))$

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

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.

C. The observational aspect of the state function

Redefining F_i as follows:

$F_i(n, c_1,..., c_k) =$

$f_i(F_1((n-1), c_1,.., c_k),...,F_k((n-1), c_1,.., c_k))$

$= f_i(F((n-1), c_1,..., c_k))$

Then,

$F(n, c_1,.., c_k) = (F_1(n, c_1,.., c_k),...,F_k(n, c_1,.., c_k))$

$= (f_1(F((n-1), c_1,.., c_k)),.., f_k(F((n-1), c_1,.., c_k)))$

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

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

$F(0, c_0^1,..., c_0^k) = (c_0^1,..., c_0^k)$

$F(n, c_0^1,..., c_0^k) = let \quad c_1^n = f_1(F((n-1), c_0^1,..., c_0^k))$

$$\vdots$$
$$c_k^n = f_k\,(F\,((n\text{-}1),\ c_0^1,\ldots,c_0^k))$$
$$\text{in}\ (c_n^1,\ldots,c_n^k)$$

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.

IV. 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 follow:

$$ma = [f_1,\ f_2\,\mathbf{o}\,(f_1,\ldots,f_1^s),\ \ldots,\ f_s\,\mathbf{o}\ldots\mathbf{o}\,(f_1,\ldots,f_1^s)] \quad \text{(m1)}$$

In such form, only the fi 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 fi 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.

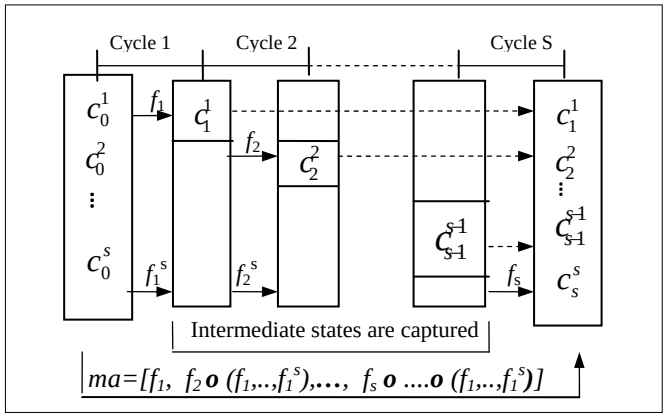

Fig. 2. MA-Step decomposition capturing the intermediate states

$$ma::W \rightarrow W$$
$$ma(c_0^1,\ldots,c_0^s) = \text{let}$$
$$c_1^1 = f_1\,(c_0^1,\ldots,c_0^s)$$
$$\vdots$$
$$c_s^s = (f_s\,\mathbf{o}\ldots\mathbf{o}\,(f_1,\ldots,f_1^s))\,(c_0^1,\ldots,c_0^s)$$
$$\text{in}\ (c_1^1,\ldots\,c_s^s)$$

Fig. 3. MA-Step Implementation capturing the intermediate states

V. 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).

$$SMA::(Int, W) \rightarrow W$$
$$SMA(0, c_0^1,\ldots,c_0^s) = (c_0^1,\ldots,c_0^s)$$
$$SMA\,(n, (c_0^1,\ldots,c_0^s)) = ma(SMA((n\text{-}1), c_0^1,\ldots,c_0^s))$$

By infolding the *ma* function, the SMA definition rewrites as follows:

$$SMA(\mathbf{n},\ c_0^1,\ldots,c_0^s) =$$
$$\textbf{let}\ \ c_n^1 = f_1(SMA((\mathbf{n\text{-}1}),\ c_0^1,\ldots,c_0^s)$$
$$\vdots$$
$$c_n^s = (fs\ \text{o}\ ..\text{o}\ (f_1,\ldots,f_1^s))\ (SMA((\mathbf{n\text{-}1}),\ c_0^1,\ldots,c_0^s))$$
$$\textbf{in}\ \ (c_n^1,\ldots,c_n^s)$$

Fig. 4. Functional Specification of the SMA model

VI. 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.

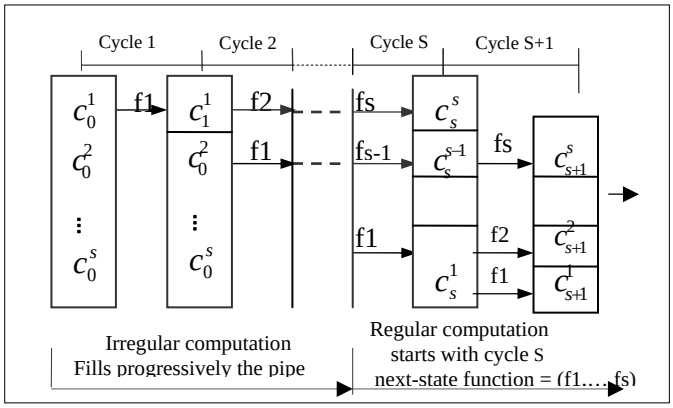

Fig. 5. Pipelined Model diagram

Again, the key solution for constructing the PMA model consists of decomposing the PMA state. Let *S*, be the number of pipelining stages, *fi*, for 1≤ i ≤ *S*, be the component function that performs the functionality of the stage i, and *W=(W1,…,Ws)*, 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: $c_0^1,\ldots,c_0^s$, the state at cycle *S*-1, is computed as follows:

$$PMA((s\text{-}1),\ c_0^1,\ldots,c_0^s) = [(f_{s-1},\ldots,f_1)\ o \ldots o\ (f_2, f_1)\ o\ f_1](c_0^1,\ldots,c_0^s)$$

- 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,…,f_s), which establishes automatically after S cycles. So, the PMA state at cycle k≥S, is computed as follows:

$$PMA(k,\ c_0^1,\ldots,c_0^s) = (f_1,\ldots,f_s)\ (PMA((k\text{-}1),\ c_0^1,\ldots,c_0^s))$$

Figure 6 shows the functional implementation of this regular form

$$\begin{aligned} &PMA(k,\ c_0^1,\ldots,c_0^s)\ = \\ &let\ \ c_k^1 = f_1\ (PMA\ ((k\text{-}1),\ c_0^1,\ldots,c_0^s)) \\ &\quad\vdots \\ &\quad c_k^s = fs\ (PMA\ ((k\text{-}1),\ c_0^1,\ldots,c_0^s)) \\ &in\ (c_k^1,\ldots,c_k^s) \end{aligned}$$

Fig. 6. Functional Specification of the PMA model for k≥S

VII. VERIFICATION OF THE PIPELINED MA MACHINE

A. 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.

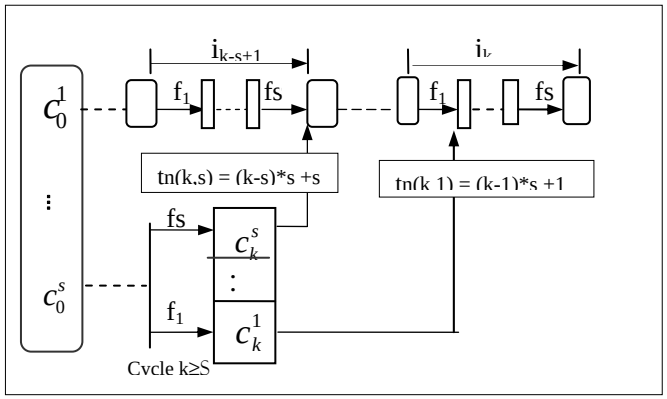

Fig 7. Synchronisation between pipelined and sequential models

In case of no stalls, the synchronization is performed using the following time function:

$$t_n(k,j) = (k\text{-}j)*s + j \qquad \textbf{(t1)}$$

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:

$$t_s(k,j,e) = ((k\text{-}j)\text{-}e)*s + j \quad \textbf{(t2)}$$

where e, is the number of stalls

B. 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≥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.

$$\begin{aligned} &CSMA(\boldsymbol{k},\ c_0^1,\ldots,c_0^s\ ) = \\ &Let\ \ c_k^1 = f1(SMA((\boldsymbol{k\text{-}1}),\ c_0^1,\ldots,c_0^s) \\ &\quad\vdots \\ &\quad c_k^s = (fs\ o,\ ..,o\ (f1,\ldots f_1^s))\ (\ SMA((\boldsymbol{k\text{-}s}),\ c_0^1,\ldots,c_0^s)) \\ &in\ (c_k^1,\ldots,c_k^s) \end{aligned}$$

Fig. 8. Functional Specification of the CSMA model for k≥S

C. Correctness criterion

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

$$\forall\ \ k :: Int,\ \ \forall\ c_0^1 :: W1, \ldots c_0^s :: Ws$$

$$PMA(k,\ c_0^1,\ldots,c_0^s)\ = CSMA(k,\ c_0^1,\ldots,c_0^s)$$

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

$$f_1\ (PMA\ ((k\text{-}1),\ c_0^1,\ldots,c_0^s)) = f_1\ (SMA((k\text{-}1),\ c_0^1,\ldots,c_0^s) \quad \textbf{(e1)}$$

$$\vdots$$

$$\wedge\ \ fs\ (PMA((k\text{-}1),c_0^1,\ldots c_0^s)) = (fs\ o\ldots o\ (f_1,..,f_1^s))(SMA((k\text{-}s),c_0^1,\ldots c_0^s)) \quad \textbf{(es)}$$

D. 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]
- Also, such equations can be instantiated for any particular architecture by just specifying the stage functions *fi*. Hence, the proof methodology scales well as designs get complex.
- The number of equations to prove depends to the number of observables. This means that, we can limit the proof only to

some observations on which someone is interested. This is very useful for complex pipelined designs involving many observations.

VIII. MODELLING THE SUPERSCALAR MA MACHINE

Superscalar designs extend pipelined designs by replicating pipeline stages such that to issue multiple instructions per cycle. In this work, we will limit ourselves only to in-order execution designs where instructions are issued in there original program order and results are written in the same order as well (the application of our methodology to out-of-order execution designs is still at hand). In this way, a SSMA model will be built upon a PMA model as follows:

Let $W=((W_1^1,\ldots,W_1^s),\ldots,(W_n^1,\ldots,W_n^s))$, be the SSMA state distributed over n pipelines (each one with S stages), and PMA_i for $1 \leq i \leq n$, be the function that performs the functionality of the pipeline i. Thus, given an initial state: $((c_0^{11},\ldots,c_0^{s1}),\ldots,(c_0^{1n},\ldots,c_0^{sn}))$, the state of a SSMA design at clock cycle k≥S, results from combining the states computed by the different pipelines as follows:

$$
\begin{aligned}
&SSMA(n,k,(c_0^{11},\ldots,c_0^{s1}),\ldots,(c_0^{1n},\ldots,c_0^{sn})) = \\
&\quad let\ \ (c_k^{11},\ldots,c_k^{s1}) = PMA_1(k,\ c_0^{11},\ldots,c_0^{s1}) \\
&\qquad\vdots \\
&\qquad (c_k^{1n},\ldots,c_k^{sn}) = PMA_n(k,\ c_0^{1n},\ldots,c_0^{sn}) \\
&\quad in\ \ ((c_k^{11},\ldots,c_k^{s1}),\ldots,(c_k^{1n},\ldots,c_k^{sn}))
\end{aligned}
$$

Fig. 9. Functional Specification of the SSMA model for k≥S

IX. VERIFICATION OF THE SUPERSCALAR MA MACHINE

A. Synchronisation diagram

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

Let $c_k^{ji} = f_j(\text{PMA}_i((\text{k-1}), c_0^{1i},\ldots,c_0^{si})$, be the SSMA component state produced by the stage function f_j, of the pipeline i, at cycle k≥S

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

$$t_n(k,j,i)=(n*(k-j)+(i-1))*s+j \quad (\mathbf{t3})$$

where n, is the number of pipelines. The corresponding sequential state is computed as follows:

$$s^j_{t(k,j,i)} = (f_j \circ \ldots \circ (f_1,\ldots,f_1^s))(\text{SMA}((n*(k-j)+(i-1)),\ c_0^{1i},\ldots,c_0^{si}))$$

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

$$t_s(k,j,i,e)=(n*(k-j)+(i-1))-e)*s+j \quad (\mathbf{t4})$$

where e, is the number of stalls.

B. 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≥ S

$$
\begin{aligned}
&CSSMA\ (n,\ k,\ (c_0^{11},\ldots,c_0^{s1}),\ldots,(c_0^{1n},\ldots,c_0^{sn})) = \\
&\quad let\ \ (c_k^{11},\ldots,c_k^{s1}) = CSMA_1(k,\ c_0^{11},\ldots,c_0^{s1}) \\
&\qquad\vdots \\
&\qquad (c_k^{1n},\ldots,c_k^{sn}) = CSMA_n(k,\ c_0^{1n},\ldots,c_0^{sn}) \\
&\quad in\ \ ((c_k^{11},\ldots,c_k^{s1}),\ldots,(c_k^{1n},\ldots,c_k^{sn}))
\end{aligned}
$$

Fig. 10. Functional Specification of the CSSMA model for k≥S

C. Correctness criterion

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

$$
\begin{aligned}
&\forall\ n,k :: Int,\ \forall\ c_0^{11} :: W_1^1,\ldots,c_0^{s1} :: W_1^s,\quad c_0^{1n} :: W_n^1,\ldots,c_0^{sn} :: W_n^s \\
&SSMA(n,k,(c_0^{11},\ldots,c_0^{s1}),\ldots,(c_0^{1n},\ldots,c_0^{sn})) = \\
&\qquad CSSMA(n,k,(c_0^{11},\ldots,c_0^{s1}),\ldots,(c_0^{1n},\ldots,c_0^{sn}))
\end{aligned}
$$

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

$$
\begin{aligned}
&PMA_1(k,\ c_0^{11},\ldots,c_0^{s1}) = CSMA_1(k,\ c_0^{11},\ldots,c_0^{s1}) \\
&\qquad\vdots \\
&\wedge\ PMA_n(k,\ c_0^{1n},\ldots,c_0^{sn}) = CSMA_n(k,\ c_0^{1n},\ldots,c_0^{sn})
\end{aligned}
$$

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

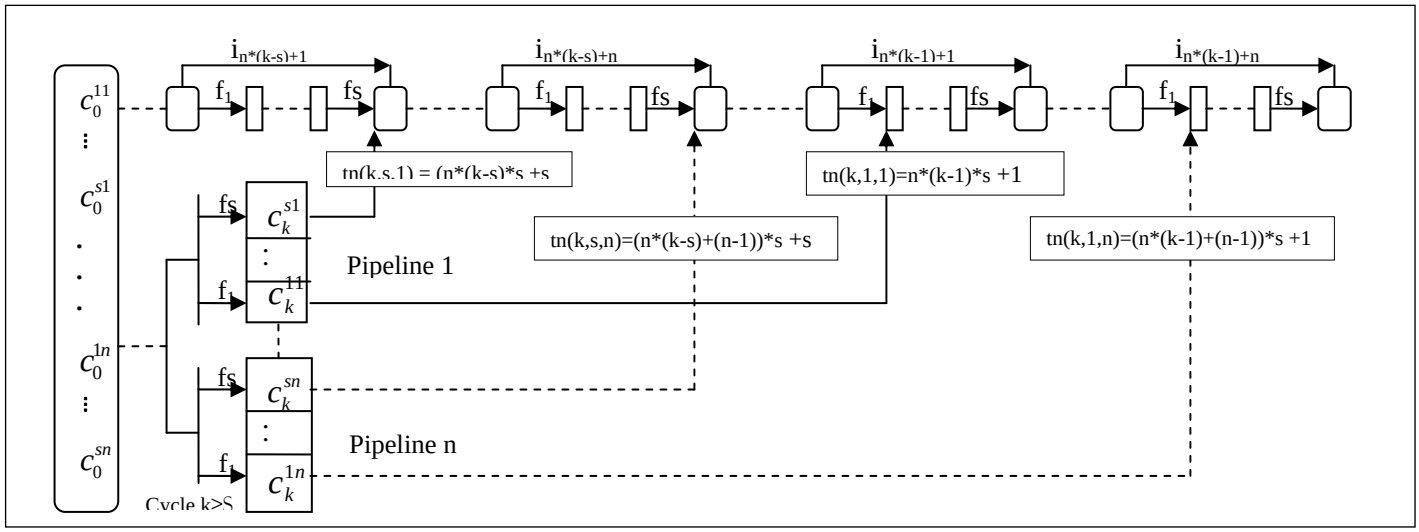

Fig 11. Synchronisation between superscalar and sequential models

X. 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 CSMAs 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. Further details are given below.

A. Definition of the MA State

The observation is limited to three components: The program counter, the register file, and the data memory (the instruction memory remains unchanged). Such components are typed as follows:

type Word = [Bit]**,** type PC = Word**,** type RegFile = [Word]**,** type Dmem = [Word]**,** type Imem = [Word]

The MA-state includes beside the observables, the pipeline registers which temporarily hold information between the different stages.

```
type PipeFD = (PC, IR)
type PipeDE = (PC, IR, RA, RB, RI)
type PipeEM = (IR, RB, Aluout, Cond)
type PipeMW = (IR, Aluout, Lmd)
type MA_state = (PipeFD, PipeDE, PipeEM, PipeMW, PC,
                 RegFile, Dmem, Imem)
```

B. Specification of the MA Stages

The interfaces specifications of the different stages are given below.

```
fe :: MA_state → PipeFD
de:: MA_state → PipeDE
ex:: MA_state → PipeEM
me:: MA_state → (Dmem, PipeMW)
wb:: MA_state → RegFile
```

To simulate a pipelined design, we also need selector functions to copy the remaining unchanged component states (which are needed later) from one pipe to the next.

```
fs:: MA_state → (PipeDE, PipeEM, PipeMW)
ds:: MA_state → (PipeFD, PipeEM, PipeMW)
es:: MA_state → (PipeFD, PipeDE, PipeMW)
ms:: MA_state → (PipeFD, PipeDE, PipeEM)
ws:: MA_state → ( PipeFD, PipeDE, PipeEM, PipeMW)
```

C. Pipelined model

We will limit ourselves only to the regular phase (for k≥ 5) involving all stages which are clocked in parallel. Of course, the system begins first by progressively filling the pipes before to stabilise. At the end of clock cycle k, the PMA model shows five partial results each one relates to a separate instruction within the pipe.

```
pma :: (Int, MA_state) → MA_state
pma(k, fd, de, em, mw, pc rf,, dm, im) =
Let (pc', ir1) = fe(pma((k-1), fd, de, em, mw, pc rf, dm,im))
 (pc2,ir2,ra,rb,ri)=de(pma((k-1),fd,de,em, mw, pc, rf, dm,im))
 (ir3,rb2,aluout,cond)=ex(pma((k-1),fd,de, em, mw, pc,rf, dm,im))
 (dm',ir4,aluout2,lmd)=me(pma((k-1),fd,de,em,mw,pc,rf,dm,im))
 rf ' = wb(pma ((k-1),fd, de, em, mw, pc,rf, dm,im))
in ( fd' = (pc', ir1), de' = (npc2, ir2, ra, rb, ri),
     em' = (ir3, rb2, aluout, cond'), mw' = (ir4, aluout2, lmd),
       pc', rf ', dm', im )
```

D. Sequential Model

The sequential model returns the state after executing k instructions

```
sma :: (Int, MA_state) → MA_state
sma (k, fd, de, em, mw ,pc, rf, dm,im) =
let (pc' ir1) =
        fe (sma((k-1), fd, de, em, mw, pc,rf, dm,im))
    (npc2,ir2,ra,rb,ri )=
        (de.(fe,fs))(sma((k-1),fd,de,em,mw,pc,rf,dm, im))

    (ir3, rb2, aluout, cond) =
      (ex . (de,ds).(fe,fs)) (sma((k-1),fd,de,em,mw,pc,rf, dm,im))

    (dm', ir4, aluout2, lmd) =
      (me .(ex, es) . (de,ds) . (fe,fs)) (sma((k-1),fd,de,em,mw, pc,rf,dm,im))

    rf '=
  (wb. (me,ms). (ex,es). (de,ds). (fe,fs))(sma((k-1),fd,de,em,mw,pc,rf,dm,im))

in ( fd' = (pc', ir1), de' =(npc2, ir2, ra, rb, ri),
   em' = (ir3, rb2, aluout, cond'), mw' = (ir4, aluout2, lmd),
   pc', rf ', dm', im )
```

E. Component Sequential model

The CSMA model which is defined in terms of the SMA model returns the same five partial results computed by the PMA model. At clock cycle k, each stage i, computes the partial result relating to the instruction (k+1)-i. The specification of the CSMA model simulating the pipelining computation of the MIPS processor is given below.

```
csma :: (Int, MA_state) → MA_state
csma (k, fd, de, em, mw, pc, rf, dm, im ) =
Let (pc' ir1) =
        fe(sma((k-1), fd, de, em, mw,pc, rf, dm im))
  (npc2, ir2, ra, rb, ri ) =
        (de . (fe, fs)) (sma((k-2), fd, de, em, mw, pc,rf,dm, im))
  (ir3, rb2, aluout, cond) =
        (ex . (de, ds) . (fe,fs)) (sma((k-3), fd, de, em, mw, pc,rf,dm, im))
  (dm', ir4, aluout2, lmd) =
      (me . (ex,es) . (de,ds) . (fe,fs)) (sma((k-4),fd,de,em, mw, pc,rf,dm,im))
   rf ' =
 (wb . (me,ms) . (ex,es) . (de,ds) . (fe,fs)) (sma((k-5),fd,de,em,mw,pc,rf,dm,im))
in ( fd' = (pc', ir1), de' = (npc2, ir2, ra, rb, ri),
   em' = (ir3, rb2, aluout, cond), mw' = (ir4, aluout2, lmd),
     pc', rf ', dm', im )
```

F. Correctness criterion

According to section 7.3, the correctness proof of the pipelined MIPS processor is ensured if the following equations are satisfied:

fe(pma((k-1), ma_state)) =
fe(sma((**k-1**), ma_state) (**a1**)

∧ **de**(pma((k-1), ma_state)) =
(de . (fe, fs)**)** (sma((**k-2**), ma_state)) (**a2**)

∧ **ex**(pma((k-1),ma_state))=
(ex . (de, ds) **.** (fe,fs)**)**(sma((**k-3**),ma_state)) (**a3**)

∧ **me**(pma((k-1), ma_state))=
(me . (ex,*es*) **.** (de,ds) **.** (fe,fs)**)**(sma((**k-4**),ma_state)) (**a4**)

∧ **wb**(pma((k-1), ma_state)) =
(wb . (me,ms) **.** (ex,es) **.** (de,ds)**.**(fe,fs)**)** (sma((**k-5**),ma_state)) (**a5)**

Although it is possible to prove the functionalities of all stages, we will limit the correctness proof only to three observations: The program counter, the register file, and the memory. Therefore, we also need to project out the pc and the memory states, after being updated by the fetch and memory stages respectively. Such projection function is omitted for the register file state as it is the only component state that can be observed at the write back stage.

projPc :: MA_state $\rightarrow$ PC
projMr :: MA_state $\rightarrow$ Dmem

In case of absence of hazards or in case of presence of hazards that could be resolved using forwarding mechanisms (no stalls), the correctness proof of the pipelined MIPS processor is ensured if the following equations are satisfied.

projPc(fe(pma((k-1), ma_state))) =
projPc (fe(sma((k-1), ma_state))) (**b1**)

∧ projMr(me(pma((k-1),ma_state))) =
projMr **(**(me**.** (ex,es)**.** (de,ds)**.** (fe,fs))(sma((k-4),ma_state))**)** (**b2**)

∧ wb(pma ((k-1),ma_state)) =
(wb **.** (me,ms)**.** (ex,es)**.** (de,ds)**.** (fe,fs))(sma((k-5),ma_state)) (**a5**)

In case of taken branch (requiring stalls) the verification condition of the PC state rewrites as follows

projPc(fe(pma((k-1), ma_state))) =
projPc (fe(sma(((k-1)-i), ma_state))) **(b3)**

Where, *i*, is the number of clock cycles for which the system must be stalled.

Notice that, the equations (b1) and (b2) are consequent of the equations (a1) and (a4). If (a1) and (a4) are satisfied then (b1) and (b2) follow systematically.

Now, it becomes easier to reason about each verification condition separately. Furthermore, we can reason either about individual instructions or about groups of instructions such as register-type, memory-type, or branch-type instructions

G. Correctness proof

The proof will be carried out by induction over clock cycles. The base case is implicit because both models start from the same initial state. So, we will consider only the inductive case. To ease the proof we will consider for each stage function only the input parameters which are necessary for the computation of the corresponding output state (the remaining parameters are necessary only for the correct typing of the stage functions)

- *Pc state*

- *R-type and M-type instructions***:** To compute the next PC For these types of instructions, the fetch stage function of the pipelined model requires as active parameter, only the result previously produced by the same stage as shown in figure 12.

Now, we assume that equation (a1) holds for cycle k, and we try to prove it for cycle k+1, as well.

fe (pma(k, ma_state)) =
fe(fe(pma(k-1),ma_state)) *definition of pma (one active parameter)*
= fe(fe (sma(k-1), ma_state)) *inductive case*
= fe (sma(k, ma_state)) *definition of sma*

Consequently,

projPc(fe (pma(k, ma_state))) = projPc (fe (sma(k, ma_state)))

Which, quickly terminates the proof

- *Branch Instructions:* For the branch instructions, the fetch stage activity depends on the result produced by the execute stage (see [23], p.A33).

Let (ir3', rb',aluout',cond') = **ex**(pma((k-1), ma_state))

Two cases will be discussed

- *Case 1***:** cond = False (untaken Branch)
In this case the execution continues in sequence, and the PC is updated (incremented) using only the fetch stage result. Therefore the proof is straightforward because this case is similar to the one discussed above.

Case 2: cond = True (taken branch)
In this case, the pc is updated using the execute stage result (as active parameter) of cycle k. Hence,

fe(pma(**k**,ma_state))
= fe(ex(pma(k-1),ma_state)) *definition of pma*
= fe ((ex **.** (de,ds) **.** (fe,fs))**.** (sma((k-3),ma_state))) *inductive case*
= fe(sma((**k-2**),ma_state)) *sma with time function **t2**, (2 stalls)*

Consequently,
projPc(fe (pma(**k**, ma_state))) = projPc(fe(sma((**k-2**), ma_state)))

This means that we need to stall for two cycles before to update the pc with the branch address (the two last instructions are ignored as shown in figure 13). According to the equation (b3), the approach gives us the right result.

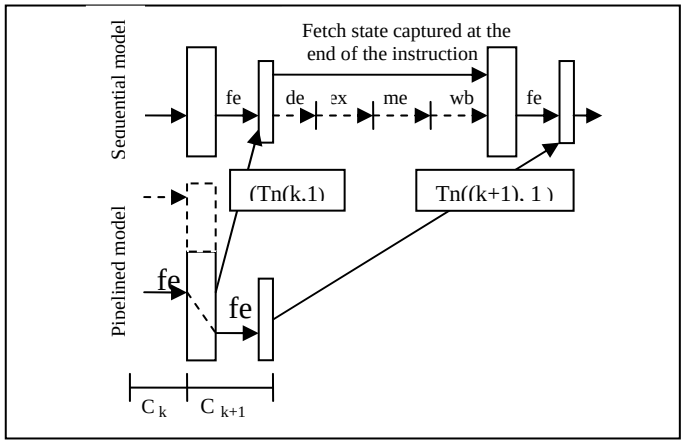

Fig 12. Correctness diagram of the fetch stage for R-type and M-type

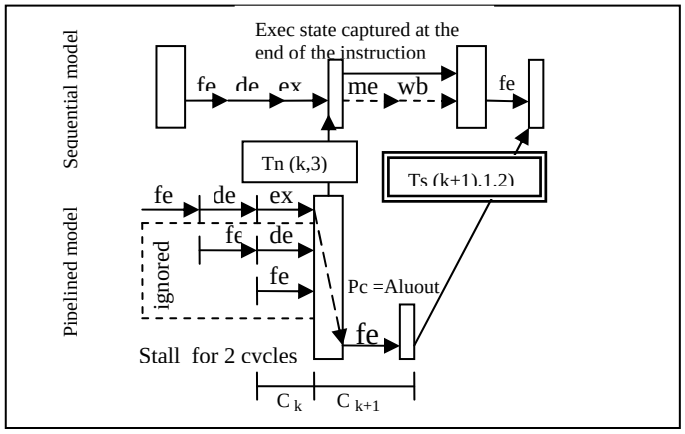

Fig 13. Correctness diagram of fetch stage for taken branch
In this case the system Stalls for 2 cycles

- *Data memory state*

- *R-type instructions:* For R-type instructions, the memory-access stage function inputs as active parameter only the execute stage result and just passes it from the EX/MEM pipeline register to the MEM/WB pipeline register. So, the proof is very easy.

- *M-type instructions:* The memory stage inputs the execute stage and the memory stage results.

me (pma(k, ma_state))

= me **[me**(pma((k-1),ma_state)), **ex**(pma((k-1), ma_state))

= me**[(**me**.**(ex,es)**.**(de,ds)**.**(fe,fs)) (sma((**k-4**),ma_state))**,** *1st parameter*
(ex**.**(de,ds)**.**(fe,fs)) (sma((k-3),ma_state))**]** *2d parameter*

= me**[(**es.(de,ds)**.**(fe,fs))(sma((**k-3**),ma_state))**,** *1st parameter at end*
(ex**.**(de,ds)**.**(fe,fs)**)**(sma((k-3),ma_state))**]** *2d parameter, the same*

= me**[**((ex,es) **.**(de,ds)**.**(fe,fs))(sma((k-3),ma_state))**]** *factoring*

Consequently,
projMr(me (pma(k, ma_state))) =
projMr**(**(me**.**(ex,es) **.**(de,ds)**.**(fe,fs)) (sma((k-3),ma_state))**)**

which establishes the proof

- *Register file state*

- R-type and load instructions: The write-back stage activity is the same for both R-type and load instructions [23, p.A32]. It inputs two active parameters: the result produced by the memory stage of the same instruction and the result produced by the write-back stage of the previous instruction, and updates the register file**.**

wb(pma(k,ma_state)

= wb **[me** (pma((k-1),ma_state)), **wb**(pma ((k-1),ma_state))**]**

= wb **[(**me**.**(ex,es)**.**(de,ds)**.**(fe,fs)**)** (sma((k-4), ma_state))**,** 1st *param*
(wb**.**(me,ms)**.**(ex,es)**.**(de,ds)**.**(fe,fs)**)**(sma((k-5),ma_state))**]** *2d param*

= wb **[(**me**.**(ex,es)**.**(de,ds)**.**(fe,fs)) (sma((k-4), ma_state)), *1st param, same*
(ms**.**(ex,es)**.**(de,ds)**.**(fe,fs)) (sma((k-4),ma_state))**]** *2d param,at end*

= wb **[(**(me,ms)**.**(ex,es)**.**(de,ds)**.**(fe,fs)**)** (sma((k-4), ma_state))**]** *factoring*

which terminates the proof.

XI. 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 (real designs are validated by mixing these two techniques [17], [18], [19]). 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

[1] J.Burch, and D.Dill. Automatic Verification of pipelined microprocessor control. *CAV'94*, LNCS 818, June 1994.

[2] R.M.Hosabettu, M.Srivas, and G.Gopalakrishnan, Decomposing the Proof of Correctness of Pipelined Microprocessors, *CAV'98*, LNCS 1427, June1998.

[3] M.Velev and R.Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction, *DAC'00*, June 2000

[4] J. R. Burch, Techniques for Verifying Superscalar Microprocessors. DAC'96, June 1996.

[5] M.N.Velev, Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. *DATE'02*, March 2002

[6] S. K. Lahiri, S. A. Seshia, and R. E. Bryant, Modeling and verification of out-of-order microprocessors, *FMCAD'02*, LNCS 2517, Nov 2002.

[7] A.C.J. Fox. An algebraic framework for verifying the correctness of hardware with input and output: A formalization in HOL, *CALCO 2005*, LNCS, 3629, 2005.

[8] A.C.J Fox. Verifying the ARM Block Data Transfer Instructions, *DCC 2004*, Barcelona, 2004.

[9] P. Manolios, Correctness of pipelined machines. *In W.A. Hunt, and S. D. Johnson editors, FMCAD 2000*.

[10] S. Tahar and R. Kumar, A Practical Methodology for the Formal Verification of RISC Processors, *Formal Methods in Systems Design, 13(2),* September 1998.

[11] K.C. Claessen, *An Embedded Language Approach to Hardware Description and Verification*, PhD thesis, Chalmers University of technology. 2001.

[12] J.R. Matthews: *Algebraic specification and verification of processor Micro-architectures* PhD thesis, Oregon Graduate Institute of science and Technology, 2000.

[13] S. Merniz, M. Benmohammed, A Methodology for the Formal Verification of RISC Microprocessors A Functional Approach, *AICCSA'07*, Amman, Jordan, May 2007

[14] S L. Peyton Jones et al, *Haskell 98: A non strict,, purely functional language*. Revised; February 1999. Available at: http://www.haskell.org/onlinereport

[15] J.L.Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*. 3rd Edition, Morgan Kaufmann Publishers Inc, San Francisco CA, 2003.

[16] R.E. Bryant, S.K. Lahiri,, S.A. Seshia. Convergence testing in term level bounded model checking. *CHARME 2003*.

[17] G. Fabbri, E. Nistico, E. Santini, Building Simulation Modeling Environments, *WSEAS transactions on circuits and systems*, Issue 9, Volume 4, September 2005

[18] L. Jie, H. Kewei, Z. Shengxian, F. Ningjun, H. Guanglin, Design and Simulation of a New Isolated Feedback Circuit for Flyback Charging Circuit, *WSEAS transactions on circuits and systems*, Issue 2, Volume 6, February 2007

[19] T.H. Chiang, L.R. Dung, Hybrid Verification Technique for High-Level Synthesis of Dataflow Algorithms, *WSEAS transactions on circuits and systems*, Issue 3, Volume 6, March 2007