Academic Journal of Computing & Information Science
ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

Improvement of Algebraic Models of Abstract Pipelines for Formal Verification

Feng Zhang, Wensheng Niu

School of Computer Science and Engineering, Beihang Univerisity, Beijing, China

ABSTRACT. *A systematic approach to model microprocessors and their correctness is useful and necessary for practical projects of formal verifications. We extend existing models to support non-superscalar pipelines with dynamic stalling. We introduce a set of algebraic tools and methods to model the specification, implementation and verification, to define formal correctness condition in formal verification and guide the actual work of microprocessors formal verification. This method is a general basis of a uniform theoretical framework for modeling microprocessors, not limited to specific reasoning systems. We consider the microprocessors determined by iterated maps that data abstractions evolve over time from some initial state, at different levels of temporal and data abstraction. We apply this method to model a pipelined microprocessor with dynamic stalling and verify it using algebraic equations.*

KEYWORDS: *Algebraic models, a uniform theoretical framework, time and data abstraction, formal verification*

1. Introduction

This paper analyses the nature of initialization, data and time abstraction in pipelined microprocessors. With this basic of pipelines and algebra preliminary, the task of this paper is to present an effective algebraic model of correctness for non-superscalar pipelined microprocessors design and formal verification, but not restricted to specific reasoning software (such as some term rewriting systems, theorem prover etc.). We do not concern the specific works on microprocessors formal verification using specific software tools. This algebraic model is a general method, and may be represented in a range of machine reasoning systems. It forms a basis of uniform theoretical frameworks for modeling microprocessors, and simplifies the actual processes of formal verification.

We apply this algebraic method to the specification, implementation and verification of a system. The microprocessors can be seen a system determined by iterated maps that data abstractions evolve over time from some initial states, at different levels of temporal and data abstraction.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

Our interest is algebraic models of time and data abstraction, and complex temporal relations when a system evolving from states to states at different levels of abstraction. This involves temporal logic and state machine. We emphasize on the application of this method to an abstract non-superscalar pipeline with dynamic stalling.

In this paper, we introduce the method of correctness for modeling and verification of microprocessors with algebraic tools, and revise some points with vague meaning. Our work is based on the theory presented in [1]. In [3], Harman started to use algebraic method to model digital system, and emphasize on the specifications for digital systems. Previous to this, they have done many works on temporal logic and formal specification in [4] [5] [6]. In [6], they present the theory of time-consistency which is applied in the correctness proof. In [7], they introduce a model of temporal logic and abstraction for synchronous digital hardware. [8] [9] [10] [11] systematically present the algebraic method for modeling microprocessors, and the correctness equation for proof. This method introduced the temporal abstraction, the relationship between time at different levels of abstraction, and the concept of the correctness of the implementation with respect to its specification. In addition, they applied their theory to a case study of a microprocessor. [12] [13] present an application of this method for formal verification of superscalar computers. To prove the correctness of this method, [14] used HOL to prove an application. [15] introduces an overview of progress on the formal specification and verification of a commercial processor – ARM6 with the application of algebraic theory of this method, using HOL proof system. [16] applies Maude to a simple pipelined microprocessor. [17] introduces an algebraic framework for the verification of correctness of hardware with input and output, using HOL. In [18] and [19], Harman extend its model of correctness for non-superscalar microprocessors to SMT and CMT processors and multithreaded and multi-core processors respectively.

The above is about the algebraic method in formal verification for digital system. There are many other interesting works on pipeline microprocessors. The concept of [20] [21] is from [22] whose verification is on a simple pipelined processor. In [22], specification and implementation evolve as states stream, but time is not explicitly present; multiple copies of states of specification should be inserted to synchronize the specification and implementation.

[23] presents a new HOL4 formalization of the current ARM instruction set architecture, ARMv7, which is a modern RISC architecture with many advanced features. [24] presents a direction in ISA for producing detailed models of Instruction Set Architectures.

2. Algebra Preliminary

In this section, we present the basic algebraic theory for modeling time and computer systems. We omit the details of universal algebra for computer science, which can be referred to [25] [26] [27] [28] [29]. Computer systems are modelled by

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

an algebraic framework using primitive recursive functions. Clock algebra models time and state-stream algebra models computer systems.

First, we present the correct models of implementation with respect to a functionality specification.

[1] defines the concept of the correctness of mapping between microprocessors at two level of abstraction. The programmer's model **PM** can be regarded as a microprocessor's functional or requirements specification, or architecture. The abstract circuit **AC** is the implementation of a design and describes the main factors of an actual circuit. Definition 2.1 (see [7] [8]) introduces the correctness model for the implementation **AC** with respect to a functional or requirements specification **PM**. This definition of correctness with respective to data and temporal abstractions specifies exactly how an implementation is correct to a specific design.

Definition 2.1 A state function $G: S\times B\rightarrow B$ can be called a correct implementation of a state function $F: T\times A\rightarrow A$ with respect to data abstraction map $\psi: B\rightarrow A$ and a state-dependent retiming $\lambda \in Ret(B, S, T)$ if, and only if, for all $b \in B$ and $s = start(\lambda(b, s))$

$$F(\lambda(b, s), \psi(b)) = \psi(G(b, s)),$$

Illustrated as Figure 1

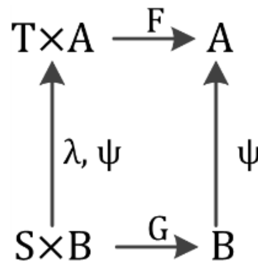

Figure. 1 A Correct Implementation Model

In a practical formal verification of pipelined microprocessors, $F: T\times A\rightarrow A$ is the state function of functionality specifications and $G: S\times B\rightarrow B$ is the state function of abstract circuit, with respect to A and B representing corresponding state set. The state-dependent retiming λ is a state-dependent time abstraction from a state with respect to time clock S to time clock T. In addition, we consider that S is faster than T, or S is as fast as T, because S represents the time clock cycles of abstract circuits and T represents the instruction clock cycles.

2.1 Temporal Abstraction: Retimings and Immersions

[29] defines a method of time abstraction and iterated mapping. It models time using clocks which divide time into discrete clock cycles, see definition 2.2.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

Definition 2.2 A clock is an algebra *(T|0, t+1)* where: (i) *T={0,1,2, ... }* is a set of clock cycles, which is natural number; (ii) *0* is the initial clock cycle; and (iii) *t+1* denotes the successor cycle function..

The purpose of clock is to denote the discrete time intervals or clock cycles. A clock may not represent a constant subdivision of time, but should denote an interval between significant states. For example, we might use an instruction cycle to represent the execution of instructions in a microprocessor. In reality, the length of each clock here are often different amounts of real time, because of variations of instruction execution times in many processor implementations.

In order to relate multiple clocks, method of retiming and immersion is introduced. We first introduce the retiming mapping, which has two properties: (i) cycle *0* of one clock is always mapped to cycle *0* of the other; (ii) the mapping is surjective and monotonic. The purpose of monotonic is to ensure there is never a discrepancy in the temporal ordering of states after abstraction, because for all s, $s' \in T$ *if* $s' > s$, then $\lambda(s') \geq \lambda(s)$, where λ is a retiming mapping. For convenience, we introduce the Definition 2.3:

Definition 2.3 A clock *S* is faster than clock *T*, or S is as fast as *T*, in a retiming map from *S* to *T*.

Definition 2.4 Let *S* and *T* be clocks. A retiming map $\lambda: S \rightarrow T$ is a subjective map with $\lambda(0) = 0$.The set of all retimings from clock S to clock *T* is represented by *Ret(S, T)*.

Definition 2.5 Immersion $\bar{\lambda}$ of a retiming $\lambda \in Ret\ (S,T)$, represented by *Imm (T, S)*, is defined by

$$\bar{\lambda}(t) = least\ s \in S\ such\ that\ \lambda(s) = t.$$

The set of all immersion from clock *T* to clock *S* is represented by *Imm (T, S)*.The meaning of $\bar{\lambda}$ is to search the first $s \in S$ such that $\lambda(s) = t$. We can give another λ definition as follows:

$$\lambda(s) = t \in T\ such\ that\ \bar{\lambda}(t) \leq s,$$

We also recognize $\bar{\lambda}$ as an inverse function to λ.

According to definition 2.5, the notion of start is present and defined as follows:

Definition 2.6 Given a retiming $\lambda \in Ret\ (S, T)$ and a time $s \in S$, the function *start*, parameterized by λ and s, returns the first time $s' \in S$ such that λ(s′) = λ(s), is defined as follows:

$$start\ (\lambda, s) = \bar{\lambda}(\lambda(s)).$$

The role of $s = start\ (\lambda\ (b, s))$ is to ensure that the correctness at all ‘start’ clock is hold. When an initialization function $h: B \rightarrow A$ is determined, then the functionality of an implementation G is restricted in a specific way with respect to the initialization function $h\ (s) = start\ (\lambda(b, s))$.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

Now, based on the above definition, the length function, which represents the length of a retimed clock *T* with respect to the numbers of clock *S*, is introduced as follows:

Definition 2.7 Given a retiming $\lambda \in Ret\ (S, T)$ and its immersion function $\bar{\lambda} \in Imm\ (T, S)$, the length function *len*, parameterized by λ and *t*, returns the number of cycles $s' \in S^{+} = S\text{-}\{0\}$, is defined as follows:

$$len\ (\lambda, t) = \bar{\lambda}(t+1) - \bar{\lambda}(t).$$

Definition 2.4 - 2.7 are illustrated in Figure 2.

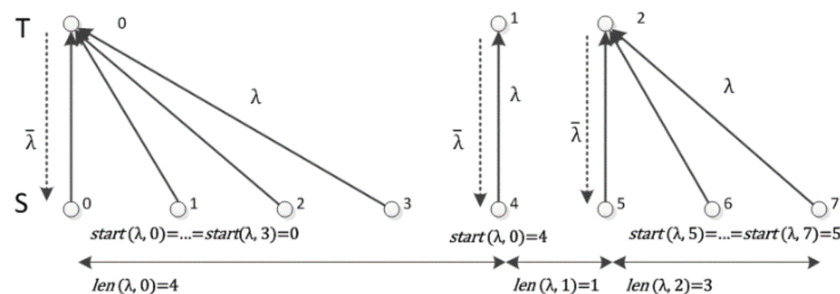

Figure. 2 Retimings, Immersions, start and len

2.2 Data Abstraction and Iterated Maps

Microprocessors can be modeled as evolving systems of states from a set A, generated by the recursive application of a next-state function $f: A \rightarrow A$, starting from some initial state $a \in A$. A state function $F: T \times A \rightarrow A$, for some clock T, computes the state of a microprocessor at time $t \in T$, giving staring state $a \in A$. The implication of *A* depends on the level of data abstraction of microprocessors. Typically *A* will be a Cartesian product of components abstraction representing registers and memories. The clock *T* depends on the level of time abstraction. For example, if each cycle of a clock *T* corresponds with an instruction, the T is suitable for architecture, or a programmer's model **PM**; if each cycle of *T* corresponds with a system clock, T is suitable for an implementation, or an abstract circuit model **AC**.

Definition 2.8 Given clock *T*, non-empty set *A*, and primitive recursive function $f: A \rightarrow A$, an iterated map $F: T \times A \rightarrow A$ is a primitive recursive function defined as follows, for all $t \in T$ and $a \in A$:

$$F\ (a, 0) = a,$$

$$F\ (a, t+1) = f\ (F\ (a, t)).$$

The above definition acquiesces the starting state is a constant state, we also consider the iterated maps generalized by an initialization function $h: A \rightarrow A$.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

Definition 2.9 Given clock *T*, non-empty set *A*, and primitive recursive function $h: A \rightarrow A$ and $f: A \rightarrow A$, an iterated map $F: T \times A \rightarrow A$ is a primitive recursive function defined as follows, for all $t \in T$ and $a \in A$:

$$F(a, 0) = h(a),$$

$$F(a, t+1) = f(F(a, t)).$$

Section 4.1.1 of [1] denotes that, the purpose of initialization functions is to eliminate unwanted starting states, not to describe the initial behavior of a system.

2.3 Data Abstraction and Iterated Maps

System states can be 'abstracted' or 'specified' by an abstraction mapping ψ. For example, if a state b represents a state of a microprocessor's micro-architecture, the state $\psi(a)$ can represent a state of the processor's architecture. Through the state transition and abstraction, a notion of temporal abstraction is induced. For example, if the mapping

$$\psi(b_0)=\psi(b_5)=a_0 \text{ and } \psi(b_1)=\psi(b_2)=\psi(b_3)=\psi(b_4)=a_1$$

Is applied to the state sequence

$$b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4 \rightarrow b_5 \rightarrow \cdots,$$

Then the state sequence will be

$$a_0 \rightarrow a_1 \rightarrow a_1 \rightarrow a_1 \rightarrow a_1 \rightarrow a_0 \rightarrow \cdots,$$

Which has only two abstracted state changes and can be recognized as the following sequence

$$a_0 \rightarrow a_1 \rightarrow a_0 \rightarrow \cdots.$$

We consider that, time is determined by considering the transition of distinct states; if no states transition occurs, or sates cease to change, time is redundant indeed.

By this example, we know that temporal abstraction may occur when there are some data abstraction and state transition. We consider time to be determined by events which can be the occurrence of something significant at the level of abstraction under consideration. For example, we may only consider the start/end of machine instructions to be events at the level of a microprocessor abstraction, and register or memory transfer operations to be events at a lower level.

3. State Iterated Maps and Time Abstraction

Section 3.1, 3.2, and 3.3 present the concepts and conditions required to simplify the verification process of an implementation.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

3.1 Time-Consistent Iterated Maps

Section 3.2.1 of [13] specifies the details of theory of time-consistency. The property of time-consistent is that it is not possible from the state of F to distinguish any time $t \in T$ from any other time $t' \in T$. The definition of time-consistent is defined as follows.

Definition 3.1 An iterated function $F: T \times A \rightarrow A$ is time-consistent if, and only if,

$$F(a, t_1 + t_2) = F(F(a, t_2), t_1),$$

For all $a \in A$ and $t_1, t_2 \in T$.

There is no initialization function in Definition 3.1. Now we consider the situation of existing initialization function, see also Section 2.3.1 of [13].

Corollary 3.2 $F: T \times A \rightarrow A$ is an iterated map with next-state function $f: A \rightarrow A$ and initialization function $h: A \rightarrow A$. Map F is time-consistent if and only if for all $a \in A$ and $t \in T$

$$F(a, t) = h(F(a, t)).$$

3.2 State-Dependent and Uniform Retimings

For each state of an implementation there will be an associated state-dependent retiming according to the theory of Section 2.3.

Retimings, and associated immersions, should be determined relative to states transition. The sequence of states is generated by recursive function F and the initialization state $G(0, b) \in B$, so the retiming λ: $S \rightarrow T$ with respect to $F: S \times B \rightarrow B$ and $G: T \times A \rightarrow A$ (clock S is faster than clock T) should also be determined by the initialization state $G(0, b) \in B$ (Details of Definition 3.2 and 3.3 are in [1, 10]).

Definition 3.2 A state-dependent retiming $\lambda: B \rightarrow Ret(S, T)$ is a map from states to retimings. The set of all state-dependent retimings is denoted as $Ret(B, S, T)$.

Definition 3.3 An immersion $\bar{\lambda}$ with respect to a state-dependent retiming λ is defined as follows:

$$\bar{\lambda}(a, t) = least\ s \in S \text{ such that } \lambda(a, s) = t.$$

The set of all immersions relative to retimings in the set $Ret(B, S, T)$ is denote d by $Imm(A, S, T)$.

Refer to definition 2.6 and 2.7, we will get the state-dependent function start and length function len as follows:

Definition 3.4 Given a time $s \in S$, a starting state $b \in B$ and the associated state-depended retiming $\lambda \in Ret(B, S, T)$ and its associated immersion function $\bar{\lambda} \in Imm(A, T, S)$, the function start, parameterized by b, λ and s, returns the first time $s' \in S$ such that $\lambda(b, s') = \lambda(b, s)$, is defined as follows:

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

$$start\ (b,\lambda,s) = \bar{\lambda}(b, \lambda(b, s)).$$

Now, based on the above definition, the length function, which represents the length of a *retimed* clock T with respect to clock S, is defined as follows:

Definition 3.5 Given a time $s \in S$, a starting state $a \in A$, a retiming $\lambda \in Ret\ (B, S, T)$ and its associated immersion function $\bar{\lambda} \in Imm\ (A, T, S)$, the length function *len*, parameterized by a, λ and t, returns the number of cycles $s' \in S^{+} = S\text{-}\{0\}$, is defined as follows:

$$len\ (a, \lambda, t) = \bar{\lambda}\ (a, t+1) - \bar{\lambda}\ (a, t).$$

We can get another one definition and two lemmas as follows.

According to Definition 3.1 and Corollary 3.1, Definition 3.6 is given.

Definition 3.6 An iterated map G: S×B→B is time-consistent with respect to a retiming λ∈ Ret (B, S, T) if, and only if,

$$G(b, \bar{\lambda}(p_2, t_1) + \bar{\lambda}(b, t_2)) = G(p_2, \bar{\lambda}(p_2, t_1))$$

Where $p_2 = G\ (b, \bar{\lambda}\ (b, t_2))$, for all $b \in B$ and $t_1, t_2 \in T$.

Lemma 3.1 $F: T \times A \rightarrow A$ be a iterated map with next-state function f: A→A and initialization function $h: A \rightarrow A$. The map F is time-consistent with respect to $\lambda \in Ret\ (B, S, T)$ if and only if , for all a $\in A$ and $t \in T$

$$F\ (a, \bar{\lambda}\ (a, t)) = h\ (F\ (a, \bar{\lambda}(a, t)))$$

Lemma 3.2 All iterated maps that do not have initialization functions are time-consistent.

Now, we introduce the concept of uniformity. Uniform shows the relation between the *length len (a, λ, t)* at some clock t ∈ T and the initial state a ∈ A. According to this concept, given a uniform retiming $\lambda \in Ret\ (B, S, T)$, in which the length *len (a, λ, t)* should be a function of the state $a \in A$ and its retiming λ, independent of time $t \in T$. We call this property of a retiming is uniformity (see Section 4.4 of [1]).

We define uniform retiming in terms of its immersion using duration function $dur: A \rightarrow S^{+}$.

Definition 3.7 Let T and S be clocks with clock S faster than clock T, $G: S \times B \rightarrow B$ and $F: T \times A \rightarrow A$ be any time-consistent functions, data abstraction map $\psi: B \rightarrow A$ and $dur: A \rightarrow S^{+}$ be a function mapping states to a positive number of cycles of clock S, $b \in B$ is an initial state of F and G. A state-dependent retiming $\lambda \in Ret\ (B, S, T)$, with its immersion $\bar{\lambda}$ is said to be uniform with respect to F and dur if, and only if , $\bar{\lambda}$ is of the form

$$\bar{\lambda}\ (b, 0) = 0,$$

$$\bar{\lambda}\ (b, t+1) = dur\ (F\ (b, t)) + \bar{\lambda}\ (b, t)$$

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

$$= dur\ (\psi\ (G\ (b, \bar{\lambda}\ (b, t))) + \bar{\lambda}\ (b, t).$$

According to the definition, the nature of dur is same as len with respect to a retiming λ and its associated immersion $\bar{\lambda}$.

Suppose that *G* represents the implementation of some systems over a clock *S*, *F* represents the specification of these systems over clock *T*, where *S* is faster than *T*. Then specification clock *t* lasts *dur (x)* cycles of clock, where $x = F\ (b, t) = \psi\ (G\ (b, \bar{\lambda}\ (b, t))$ is the state of F at clock cycle $t \in T$.

Note that, *dur* is a function only of states, because data abstraction ψ and immersion $\bar{\lambda}$ is dependent of state, and consequently the number of cycles corresponding with any states is independent of numerical value of $t \in T$.

In practice, the meaning of uniform is to denote the number of states $b \in B$ with respect to an state of $a \in A$, because the clock s denotes the state transition of *B*. With the statically definition of *dur*, we can make concrete statements about how many cycles of the implementation clock S correspond with one cycle of the specification clock T for a possible initial state $b \in B$. And because clock *S* corresponds the state transition on micro-programmed level, clock T corresponds the state transition on programmer level, we can use the meaning of uniform and *dur* to get the number of states of micro-programmed level with respect to its programmer level.

3.3 One-Step Theory for Simplifying Verification

[1] particularly specifies the concept of one-step theory for non-superscalar microprocessors, we briefly describe it here.

The role of time-consistent iterated maps and uniform retimings is to construct a theorem of one-step for simplifying formal verification. The method of simplifying formal verification is to eliminate induction over time. The fundamental notion is that, in real hardware, future state evolution is not dependent on time, but only on the current state. That is to say that state transition does depend only on the current state (and inputs at the current time if any). Briefly, given two time-consistent iterated maps $F: T\times A\rightarrow A$ and $G: S\times B\rightarrow B$, related by surjective data abstraction map $\psi: B\rightarrow A$ and uniform retiming $\lambda \in Ret\ (B, S, T)$, we can simplify the verification of G with respect to F by just considering correctness at specification times $t = 0$ and $t = 1$: that is times $s=0$ and $s = start\ (\lambda, 1)$.

Definition 3.8 Let $F: T\times A\rightarrow A$ and $G: S\times B\rightarrow B$ be iterated maps, $\lambda \in Ret\ (B, S, T)$ be a uniform retiming with respect to *G*, $\psi: B\rightarrow A$ be a surjective data abstraction map. If

(1) F is time-consistent; and

(2) G is time-consistent with respect to λ,

Then for all $b \in B$ and $s = start\ (\lambda, b, s)$

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

$$F(\psi(b), \lambda(b, s)) = \psi(G(b, s))$$

If and only if

$$F(\psi(b), 0) = \psi(G(b, 0)) \text{ and}$$

$$F(\psi(b), 1) = \psi(G(b, \bar{\lambda}(b, 1))).$$

Now, when we formally verify an abstract circuit **AC** with respect to a design **PM** in programmer level, we need only to verify **AC** at times $s=\bar{\lambda}(b, 0)=0$ and $s=\bar{\lambda}(b, 1)$.

4. An Abstract Pipeline Example

[1, 3, 15] apply the above concepts to an abstract pipeline case study, and the author has verified the abstract pipeline using HOL or Maude. Now we exploit the concept of [1] to a more universal of abstract non-superscalar pipeline.

We introduce an abstract pipeline with four stages to sufficiently demonstrate the functionality of pipelined designs. The abstract pipeline is illustrated as follows.

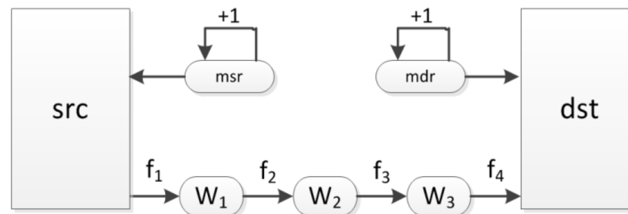

Figure. 3 An Abstract Pipeline

The function of the abstract pipeline is to transfer the data of memory source *src* to memory destination *dst*. The memory source register *msr* and the memory destination *dst* address the memories.

4.1 Functionality Specification

From the perspective of programmers, the abstract pipeline system has two memories and two memory-address registers. The system transfers the data of *src* at address *msr* to *dst* at *mdr*. The memory state-space is $M=[MAR \rightarrow W]$ where W is any non-empty set, and the memory-address register state-space is MAR. The state-space of the functionality specification is

$$State_{FS} = M \times MAR \times MAR \times M.$$

A state transition function $FS: T \times State_{FS} \rightarrow State_{FS}$ and next-state function $fs: State_{FS} \rightarrow State_{FS}$ is defined as follows:

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

$$FS\ (0,\ src,\ msr,\ mdr,\ dst) = (src,\ msr,\ mdr,\ dst),$$

$$FS\ (t+1,\ src,\ msr,\ mdr,\ dst) = fs\ (FS\ (t,\ src,\ msr,\ mdr,\ dst))$$

Where $src \in M$, $msr \in MAR$, $mdr \in MAR$ and $dst \in M$. The next-state function *fs* updates the destination memory *dst* at location *mdr* with *f (src (mdr))*,

$$Fs\ (src,\ msr,\ mdr,\ dst)=(src,\ msr+1,\ mdr+1,\ dst\ [f\ (src\ (msr))\ /\ mdr]).$$

The expression *dst [f* (src*(msr))/mdr]* is derived from the next memory substitution function:

$$Memory\ [data/a_adress]\ (b_adress)=\begin{cases} memory(b_adress), & if\ b_adress \neq a_adress, \\ data, & if\ b_adress = a_adress, \end{cases}$$

Where the data at *a*_adress $\in MAR$ is denoted as memory (a_adress); and if the *data* $\in M$ is stored at address *a_adress*, the resultant memory is denoted *memory [data/a_adress]*. Then the memory substitution function is determined as the above equation. (see Section 5.1 of [1])

4.2 Implementation Specification without Dynamic Stalling

We can divide the recursive function fs to four computations f_1, f_2, f_3 and f_4:

$$f = (f_1 \circ f_2 \circ f_3 \circ f_4)$$

Where, $f_1 : W \rightarrow W_1$, $f_2 : W_1 \rightarrow W_2$, $f_3 : W_2 \rightarrow W_3$ and $f_4 : W_3 \rightarrow W$, which functionality is to complete the functionality of fs using four steps. W_1, W_2 and W_3 store intermediate computations of operations. For brevity, $f_2 \circ f_1 : W \rightarrow W_2$ is denoted as f_{12}, and $f_3 \circ f_{12}: W \rightarrow W_3$ is denoted as f_{123}.

In previous articles [1, 12, 18] have modeled several kinds of microprocessors, such as pipelined microprocessors, superscalar-pipeline, SMT/CMT processors.

Now, what we interest is to extend the basic notion of above methods to build a more universal algebraic model for formal verification of non-superscalar pipelines. First, we will model an abstract pipelined implementation P_1 without dynamic stalling. We use a counter *ctr* $\in \{1, 2, 3, 4\}$. If *ctr*=1, it means only f_4 is idle and w_1, w_2, w_3 store valid data; *ctr*=2 denotes f_3 and f_4 are idle and only w_3 stores junk data; *ctr*=3 denotes f_2, f_3 and f_4 are idle and w_2, w_3 sotre junk data; *ctr*=4 denotes f_1 f_2, f_3 and f_4 are all idle and w_1, w_2, w_3 all store junk data [3].

The state-space of $State_{P_1}$ is

$$State_{P1} = M \times MAR \times W_1 \times W_2 \times W_3 \times MAR \times M.$$

The iterated state evolution function P_1: $S \times State_{p1} \rightarrow State_{p1}$ is determined as follows

$$P_1(0,\ \sigma) = p_1^0(\sigma),$$

$$P_1(s+1,\ \sigma) = p_1(P_1(s,\ \sigma)),$$

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

Where σ = *(ctr, src, msr,* w_1, w_2, w_3, *mdr, dst),*

$$p_1^0(\sigma)=\begin{cases} (1, src, msr, f_1(src(msr-1)), f_{12}(src(msr-2)), \\ \quad f_{123}(src(msr-3)), mdr, dst) & , ① \; if \; ctr(0)=1, \\ (2, src, msr, f_1(src(msr-1)), f_{12}(src(msr-2)), \\ \quad JD, mdr, dst) & , ② \; if \; ctr(0)=1, \\ (3, src, msr, f_1(src(msr-1)), JD, JD, mdr, dst), & ③ \; if \; ctr(0)=3, \\ (4, src, msr, JD, JD, JD, mdr, dst), & ④ \; if \; ctr(0)=4, \end{cases}$$

$$P_1\ (s+1,\ \sigma) = p_1\ (P_1\ (s,\ \sigma))=\begin{cases} ⑤\ ,\ if\ ctr(s)=1, \\ ⑥\ ,\ if\ ctr(s)=2, \\ ⑦\ ,\ if\ ctr(s)=3, \\ ⑧,\ if\ ctr(s)=4, \end{cases}$$

Where

⑤=(1,src,msr+1,f_1(src(msr)),f_{12}(src(msr-1)),

f_{123}(src(msr-2)),mdr+1,dst(f_{123}(src(msr-3)))),

⑥=(1,src,msr+1,f_1(src(msr)),f_{12}(src(msr-1)),

f_{123}(src(msr-2)),mdr,dst),

⑦=(2,src,msr+1,f_1(src(msr)),f_{12}(src(msr-1)),JD,mdr,dst),

⑧=(3,src,msr+1,f1(src(msr)),JD,JD,mdr,dst).

It can also be denotes as follows

$$P_1(s+1,\ \sigma) = p_1(P_1(s,\ \sigma))=\begin{cases} ⑤, & if\ ctr(s)=1, \\ (ctr-1, src, msr+1, f_1(src(msr)), \\ \quad f_2(w_1), f_3(w_2), mdr, dst) & , ⑨ if\ ctr(s)>1, \end{cases}$$

Expression ⑨ denotes that if *ctr(s)>1*, the pipeline will fetch instructions to fill the pipeline. The notion of $p_1(\sigma)$ is to forward the data computed in the pipeline, and when ctr>1 at a clock time, the ctr will decrement 1 in the next stage.

There is no dynamic stalling, so P_1 will fetch one instruction in one cycle of abstract circuit clock while the *msr* will plus 1 to fetch the next instruction in the next clock cycle. If *ctr=m<4*, the state of w_i ($1 \le i \le 4-m$) corresponds with source data from memory address *(msr-i)*, after the appropriate operations of f_j, for all $j \le i$; and the component(s) after w_i stores junk data. For example, if the pipeline is empty or partly empty, w_3 stores junk data; if the pipeline is full, $w_3 = f_{123}$ (src (msr-3))= $f_3 \circ f_2 \circ f_1$ (src (msr-3)). The next-state function $p_1 : State_{p1} \rightarrow State_{p1}$ implements a pipeline by forwarding intermediate computed results w_i to the next operation f_{i+1} along the pipeline. The last operation with respect to w_3 stores the result in the *dst* at address *mdr=msr*-3. For any fixed time, in our view, *msr* in the specification is the value of *msr* in the implementation from three clock cycles earlier.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

Definition 4.1 The map P_1 is a correct implementation of *FS* with respect to data abstraction map *ψ*: $State_{P_1} \rightarrow State_{FS}$

ψ(ctr, src, msr, w_1, w_2, w_3 *mdr,*dst*)* =

$$\begin{cases} (src, msr - 3, mdr, dst), & if\ ctr = 1; \\ (src, msr, mdr, dst), & if\ ctr > 1, \end{cases}$$

and uniform retiming $\lambda \in Ret_{P_1}$ *(*$State_{P_1}$*, S, T)* where duration *dur:* $State_{P_S} \rightarrow S^+$ is defined by the equation

dur(ctr, src, msr, w_1, w_2, w_3*, mdr,dst) = ctr.*

The concept and proof of this definition refers to [1]. The next section, we will extend this model to a non-superscalar pipeline with dynamic stalling.

4.3 A Model of Non-Superscalar Pipelines with Dynamic Stalling

Now, we introduce our new model of pipelined microprocessors with dynamic stalling. The method of modelling is same as [1], but we increase some improvements in this article. We do not discuss the details of stalling in pipelines here, readers can research it in many other documents. The notion of a pipeline with stalling is illustrated as Figure 4.

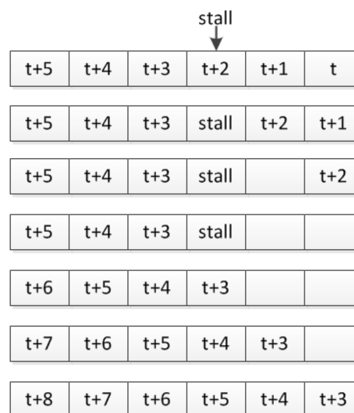

Figure. 4 A Pipeline with Stalling

In this paper, our principle concern is mathematical models, and not the practical verification. However, our method and model would be advantageous to reduce practical work of formal verification. The one-step theorem is practical-benefic for simplifying workloads in practical verification.

According to the concept of pipelines with dynamic stalling, the stalling component and its previous will detain their states evolution until an event cancels

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

the stalling, nevertheless, the stages of back components will evolves normally. For example, if a stalling source will affect the computation f_3 and there is a valid data in w_3, the correct value of w_3 will store in dst in the next clock cycles, although f_1 and f_2 are stalling and the states of w_1 and w_2 will not change until the source of stalling is cancelled. In many actual pipelines, the stalling is considered to be a nop(no-operations) in corresponding components. We now consider the circumstances that a source of stalling will affect the computation f_3.To model the time of stalling source, we introduce a map from S to a Booleans streams using the next set

$$Dstl=\{stl \in [S\rightarrow B] \mid \forall s \in S, \exists s' \in S \text{ such that } s'>s \text{ and } stl(s') = ff\}.$$

If $stl(s')=ff$, the pipeline will compute normally; and if $stl(s')=tt$, there will be a source of stalling in corresponding components.

The pipelines with dynamic stalling are named as P_S. The iterated state transition map P_S: $S\times State_{P_1}\times Dstl\rightarrow State_{P_S}$ is defined as follows

$$P_S(0, \sigma, stl) = p_S^0(\sigma),$$

$$P_S(s+1, \sigma) = p_S(P_S(s, \sigma, stl), stl(s)),$$

Where $\sigma = (ctr,src,msr, w_1, w_2, w_3,mdr,dst)$, $p_S^0(\sigma) = p_1^0(\sigma)$,

and $P_S(s+1, \sigma) = p_S(P_S(s, \sigma, stl), stl(s)) =$

$$\begin{cases} ⑤, & \text{if } ctr(s) = 1 \text{ and } stl(s) = ff, \\ ⑨, & \text{if } ctr(s) = 1 \text{ and } stl(s) = tt, \\ ⑩, & \text{if } ctr(s) = 2 \text{ and } stl(s) = tt, \\ ⑥, & \text{if } ctr(s) = 2 \text{ and } stl(s) = ff, \\ ⑦, & \text{if } ctr(s) = 3 \text{ and } stl(s) = ff \text{ or } tt, \\ ⑧, & \text{if } ctr(s) = 4 \text{ and } stl(s) = ff \text{ or } tt. \end{cases}$$

Where

⑨=*(2, src, msr, f1(src(msr)), f12(src (msr-1)), JD, mdr+1,dst (f123 (src (msr-2))))*

⑩=*(2, src, msr, f1 (src (*msr*)), f12 (src (msr-1)), JD, mdr, dst).*

We can conclude that, when existing a source of stalling, the pipeline will stop to fetch instructions until the stalling is canceled. The above equation can be reduced from the notion of pipeline and stalling. We now define the correct implementation equation.

Definition 4.2 The map P_S is a correct implementation of *FS* with respect to data abstraction map ψ: $State_{P_S}\rightarrow State_{FS}$

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

ψ (ctr, src, msr, w_1, w_2, w_3, *mdr, dst, stl) =*

$$\begin{cases}(src, msr-3, mdr, dst), & if\ ctr(0)=1\ and\ stl(0)=ff,\\ (src, msr-2, mdr, dst), & if\ ctr(0)=1\ and\ stl(0)=tt.\\ (src, msr, mdr, dst), & if\ ctr(0)>1,\end{cases}$$

and uniform retiming $\lambda \in Ret_{P_S}$ *(* $State_{P_S}$ *, S, T)* where duration *dur:* $State_{P_S}$→S^+ is defined by the equation

dur (ctr, src, msr, w_1, w_2, w_3, *mdr, dst, stl) =*

$$\begin{cases}1, & if\ ctr(0)=1,\\ NextFalse(stl)+2, & if\ ctr(0)=2,\\ 4, & if\ ctr(0)>2,\end{cases}$$

Where *NextFalse (stl): Dstl→N* is defined by

NextFalse (*stl*) = least *s* ∈*S* such that *stl(s) = ff.*

Following the property of our time abstraction (Section 2.3), we should discard clock cycles where P_S does not change stages. In our example, this is the circumstance when *ctr=2* and *stl = tt*. We can combine all clock cycles *s* ∈*S* where *stl(s)=tt* into a single cycle of some new abstract clock S'. The clock S' disallows contiguous sequences of more than one tt element:

$Dstl'$*={stl* ∈*[S→B]* | *∀s* ∈ *S, ∃*s' ∈ *S such that* s'*>s and stl(*s'*) = ff* and

∀s'' ∈ *S, if stl(*s''*) = tt,* then *stl(*s''*-1) = stl(*s''*+1) = ff}.*

The retiming is necessary to be uniform to apply the one-step theorems in practical verification. Therefore the function *dur* is revised to dur' as follows:

dur' *(ctr, src, msr,* w_1, w_2, w_3, *mdr, dst, stl) =*

$$\begin{cases}1, if\ ctr(0)=1,\\ 2, if\ ctr(0)=2,\\ ctr, if\ ctr(0)>2,\end{cases}$$

and can be reduced to

dur'*(ctr,src,msr,* w_1, w_2, w_3*,mdr,dst,stl) = ctr.*

From the different of Definition 4.1 and 4.2, we can conclude that, if we ignore the clock cycles where the stages of systems do not change, in this example, the length of state in the functionality specification is not changed.

In our example of P_S which may be stalled in computation f_3, we analysis its property and try to model a correct implementation to a specification to simplify the practical verification.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

5. Further Considerations

The next work we are concerning is modeling some more complex microprocessors, such as multicore or many-core, superscalar pipelines and other parallel microprocessors; or some complex components, such as cache and memory in many-core, which must be of cache-coherency and memory-consistence. The complex relation between time and data is the most challenge for us to overcome.

References

[1] Fox A C J, Harman N. A. 2003.Algebraic models of correctness for abstract pipelines[J]. *The Journal of Logic and Algebraic Programming*, 2003, 57 (1): 71-107.

[2] Harman, N.A.1989. Formal specifications for digital systems. Ph.D. Thesis, School of Computer Studies, University of Leeds, 1989.

[3] Harman, N.A., Tucker, J.V.1988.Clocks, retimings, and the formal specification of a UART. In: Milne, G.J. (ed.), *The Fusion of Hardware Design and Verification*, pp. 375-396. Amsterdam: North-Holland, 1988.

[4] Harman, N.A., Tucker, J.V.1988. Formal specification and the design of verifiable computers. In: *Proceedings of the 1988 UK IT Conference*, pp. 500-503, University College Swansea, lEE, 1988.

[5] Harman, N.A., Tucker, J.V.1990. The formal specification of a digital correlator I: Abstract user specification. *Theoretical Foundations for VLSI Design*, In: McEvoy, K., Tucker, J.V. (eds.), Cambridge University Press Tracts in *Theoretical Computer Science* 10, pp. 161-262 (1990).

[6] Harman, N.A., Tucker, J.V.1992. Consistent refinements of specifications for digital systems. In: Prinetto, P., Camurati, P. (eds.), *Correct Hardware Design Methodologies*, pp.273-295. Amsterdam: North-Holland 1992.

[7] John O'Donnell. 2002.Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2002.

[8] Harman N. A, Tucker J V. 1996.Algebraic models of microprocessors architecture and organization [J] .*Acta Informatica*, 1996, 33 (5): 421-456.

[9] Harman, N. A. and Tucker, J. V.1997. Algebraic models of microprocessors: the verification of a simple computer. In V.Stavridou, ed., *Proceedings of the 1995 IMA Conference on Mathematics for Dependable Systems*. Oxford University Press, Oxford, 1997.

[10] Harman N. A, Tucker J V. 1996.Algebraic models and the correctness of microprocessors [M]. *Correct Hardware Design and Verification Methods*. Springer Berlin Heidelberg, 1993: 92-108.

[11] Fox A C J, Harman N. A. 1998.Algebraic models of superscalar microprocessor implementations: A case study. [M] *Prospects for Hardware Foundations*. Springer Berlin Heidelberg, 1998: 138-183.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

[12] Fox A C J, Harman N. A. 2003.Algebraic models of correctness for abstract pipelines [J]. *The Journal of Logic and Algebraic Programming*, 2003, 57 (1): 71-107.

[13] Fox A C J. An algebraic framework for modelling and verifying microprocessors using HOL [M]. University of Cambridge, *Computer Laboratory*, 2001.

[14] Fox A C J. 2003. Formal specification and verification of ARM6 [M]. *Theorem proving in higher order logics*. Springer Berlin Heidelberg, 2003: 25-40.

[15] Harman N. A.2001. Verifying a simple pipelined microprocessor using Maude. In M Cerioli and G Reggio, editors, *Recent trends in algebraic development techniques: 15th International Workshop*, WADT 2001, Genova, Italy, April2001. Lecture Notes in Computer Science 2267, pages 128-151, SpringerV erlag.

[16] Fox A C J. 2005. An algebraic framework for verifying the correctness of hardware with input and output: a formalization in HOL [M]. *Algebra and Coalgebra in Computer Science*. Springer Berlin Heidelberg, 2005: 157-174. }

[17] Harman N. A. 2007.Algebraic models of behaviour and correctness of SMT and CMT processors [J]. *The Journal of Logic and Algebraic Programming*, 2007, 74 (1): 32-56.

[18] Harman N. A. 2007.Algebraic models of simultaneous multithreaded and multi-core processors [M] // Algebra and Coalgebra in Computer Science. Springer Berlin Heidelberg, 2007: 294-311.

[19] S Miller and M Srivas. 1995. Formal verification of an avionics microprocessor. Technical report, *SRI International Computer Science Laboratory* CSL-95-04, 1995.

[20] S Miller and M Srivas. 1995.Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods. In *Proceedings of WIFT 95*, Boca Raton.

[21] M Bickford and M Srivas. 1990.Verification of a pipelined processor using Clio. In M Leeser and G Brown, editors, *Proceedings of the Mathematical Sciences Institute workshop on hardware specification, Verification and Synthesis*: Mathematical Aspe cts, pages 307 {332. Lecture Notes in Computer Science 408, Springer-V erlag.

[22] M Srivas and M Bickford. 1991. Formal verification of a pipelined microprocessor [J]. *IEEE Software*, 7 (5): 52 -64, 1991.

[23] Fox A C J.2012. Directions in ISA specification [M]. *Interactive Theorem Proving*. Springer Berlin Heidelberg, 2012: 338-344.

[24] Rutten JJMM. 2000. Universal coalgebra: a theory of systems [J]. *Theoretical Computer Science*, 2000, 249 (1): 3-80.

[25] Meinke, K. and Tucker, J. V.1992. Universal algebra. In T. S. E. Maibaum, S. Abramsky and D. Gabbay, eds, *Handbook of Logic in Computer Science.* Oxford University Press, Oxford, 1992, 189-411.

[26] M Wirsing. 1990. Algebraic specification. In Jvan Leeuwen, editor, *Handbook of theoretical computer science*, volume B: formal models and semantics, pages 675-788. Elsevier, 1990.

[27] Gratzer G A. *Universal algebra* [M], pages: 223-269 Springer, 2008.

ISSN 2616-5775 Vol. 1, Issue 1: 53-70, DOI: 10.25236/AJCIS.010006

[28] Ehrig H, Mahr B. 2011. *Fundamentals of algebraic specification I: Equations and initial semantics* [M]. Springer Publishing Company, Incorporated, 2011.

[29] Fox A C J, Harman N A. 1998. Algebraic models of temporal abstraction for initialised iterated state systems: An abstract pipelined case study [R]. *Technical Report CSR 21-98 (submitted to Acta Informatica)*, University of Wales Swansea, 1998.