Maurer computers for pipelined instruction processing

Maurer Computers for Pipelined Instruction Processing*

J.A. Bergstra[1,2] and C.A. Middelburg[3,1]

[1] Programming Research Group, University of Amsterdam,
P.O. Box 41882, 1009 DB Amsterdam, the Netherlands
[2] Department of Philosophy, Utrecht University,
P.O. Box 80126, 3508 TC Utrecht, the Netherlands
[3] Computing Science Department, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, the Netherlands
janb@science.uva.nl,keesm@win.tue.nl

Abstract. We model micro-architectures with non-pipelined instruction processing and pipelined instruction processing, using Maurer machines, basic thread algebra and program algebra. We show that stored programs are executed as intended with these micro-architectures. We believe that this work provides a new mathematical approach to model micro-architectures and to verify their correctness and anticipated speed-up results.

Keywords: Maurer machines, basic thread algebra, program algebra, instruction set architecture, micro-architecture, pipelining.

1998 CR Categories: C.1.1, F.1.1, F.1.2.

1 Introduction

Pipelined instruction processing is a basic technique used in the design of micro-architectures (see e.g. [13, 19]). In this paper, we investigate the issue of dealing with pipelined instruction processing when modelling micro-architectures in a mathematically precise way. We model micro-architectures with non-pipelined instruction processing and pipelined instruction processing, using Maurer machines, basic thread algebra and program algebra. Moreover, we show that stored programs are executed as intended with these micro-architectures.

Maurer machines are based on a model for computers proposed by Maurer in [17]. Maurer's model for computers is quite different from the well-known models such as register machines, multi-stack machines and Turing machines (see e.g. [15]). The strength of Maurer's model is that it is close to real computers. The operations that can be performed on the state of a computer play a prominent part in the model. Basic thread algebra is a form of process algebra which is introduced in [3] under the name basic polarized process algebra. It is a form of process algebra which is tailored to the description of the behaviour

* This research was carried out as part of the GLANCE-project MICROGRIDS, which is funded by the Netherlands Organisation for Scientific Research (NWO).

of deterministic sequential programs under execution. Basic thread algebra is used in this paper to direct a Maurer machine in performing operations on its state. Program algebra is introduced in [3] as well. In program algebra, not the behaviour of deterministic sequential programs under execution is considered, but the programs themselves. A program is viewed as an instruction sequence. The behaviour of a program is taken for a thread of the kind considered in basic thread algebra. With regard to execution of stored programs on a Maurer machine, we take the line that the programs concerned are programs of the kind considered in program algebra.

To make it possible that threads direct a Maurer machine in performing operations on its state, basic thread algebra must be extended with, for each Maurer machine, an operator for applying a thread to the Maurer machine from a state of the Maurer machine. Applying a thread to a Maurer machine amounts to generating a sequence of state changes according to the operations that the Maurer machine associates with the basic actions performed by the thread. Because a program is viewed as an instruction sequence in the setting of program algebra, the representation of programs in the memory of a Maurer machines becomes trivial.

In [4], we have demonstrated the feasibility of the approach to model micro-architectures taken in this paper. In this paper, we make use of the experience gained in that feasibility study to model more advanced micro-architectures. Maurer's model for computers is quite different from Turing's model. The latter model belongs to the foundations of theoretical computer science, whereas the model used in our approach to model micro-architectures is relatively unknown indeed. For that reason, we have investigated the connections between the two models in [5].

We treat the instruction set architecture for which micro-architectures are modelled as a parameter that must fulfil a simple assumption: each instruction from the instruction set must be of a kind considered in program algebra. For example, program algebra does consider test instructions and unconditional jump instructions, but it does not consider conditional jump instructions. Besides, program algebra does consider forward jump instructions, but it does not consider backward jump instructions. The effect of a conditional jump instruction can be mimicked by a test instruction and an unconditional jump instruction; and the effect of a backward jump instruction can be mimicked by a forward jump instruction because programs may be infinite instruction sequences in program algebra.

Conditional jump instructions need another treatment than unconditional jump instructions in pipelined instruction processing. Backward jump instructions do not need another treatment than forward jump instructions in pipelined instruction processing. In order to demonstrate the generality of our approach, we look in this paper also at the influence of extending program algebra with conditional jump instructions on non-pipelined and pipelined instruction processing. We also pay some attention to backward jump instructions.

We do not make the instruction set architecture for which micro-architectures are modelled explicit. In our modelling of a micro-architecture, we start from an arbitrary Maurer machine and enhance it. That Maurer machine determines the instruction set architecture for which a micro-architecture is modelled. However, there are Maurer machines for which the enhancement is primarily intended. We describe in this paper those Maurer machines as well. Because they approximate the concept of an instruction set architecture, we call them Maurer instruction set architectures.

We regard the work reported upon in this paper as one of the preparatory steps in developing, as part of a project investigating micro-threading [8, 16], a formal approach to design new micro-architectures. That approach should allow for the correctness of new micro-architectures and their anticipated speed-up results to be verified.

The structure of this paper is as follows. First, we review Maurer computers (Section 2) and basic thread algebra (Section 3). Next, we extend basic thread algebra with, for each Maurer machine, the operator for applying a thread to the Maurer machine from a state of the Maurer machine (Section 4). Following this, we review program algebra (Section 5) and describe the way in which programs are represented in the memory of Maurer machines (Section 6). Then, we model a micro-architecture with non-pipelined instruction processing (Section 7). After that, we model a variant of that micro-architecture with pipelined instruction processing (Sections 8 and 9). Following this, we look at the influence of the addition of conditional jump instructions (Section 10) and discuss the addition of backward jump instructions in short (Section 11). Then, we introduce the concept of a Maurer instruction set architecture (Section 12). Finally, we make some concluding remarks (Section 13).

2 Maurer Computers

In this section, we shortly review Maurer computers, i.e. computers as defined by Maurer in [17].

A *Maurer computer* C consists of the following components:

- a set M;
- a set B with $card(B) \geq 2$;
- a set $\mathcal{S}$ of functions $S : M \to B$;
- a set $\mathcal{O}$ of functions $O : \mathcal{S} \to \mathcal{S}$;

and satisfies the following conditions:

- if $S_1, S_2 \in \mathcal{S}$, $M' \subseteq M$ and $S_3 : M \to B$ is such that $S_3(x) = S_1(x)$ if $x \in M'$ and $S_3(x) = S_2(x)$ if $x \notin M'$, then $S_3 \in \mathcal{S}$;
- if $S_1, S_2 \in \mathcal{S}$, then the set $\{x \in M \mid S_1(x) \neq S_2(x)\}$ is finite.

M is called the *memory*, B is called the *base set*, the members of $\mathcal{S}$ are called the *states*, and the members of $\mathcal{O}$ are called the *operations*. It is obvious that the first condition is satisfied if C is *complete*, i.e. if $\mathcal{S}$ is the set of all functions

$S : M \to B$, and that the second condition is satisfied if C is *finite*, i.e. if M and B are finite sets.

In [17], operations are called instructions. In the current paper, the term operation is used because of the confusion that would otherwise arise with the instructions of which program algebra programs are made up.

The memory of a Maurer computer consists of memory elements which have as contents an element from the base set of the Maurer computer. The contents of all memory elements together make up a state of the Maurer computer. The operations of the Maurer computer transform states in certain ways and thus change the contents of certain memory elements. We return to the conditions on the states of a Maurer computer after the introduction of the input region and output region of an operation.

Let $(M, B, \mathcal{S}, \mathcal{O})$ be a Maurer computer, and let $O : \mathcal{S} \to \mathcal{S}$. Then the *input region* of O, written $IR(O)$, and the *output region* of O, written $OR(O)$, are the subsets of M defined as follows:[4]

$$IR(O) = \{x \in M \mid \exists S_1, S_2 \in \mathcal{S} \bullet (\forall z \in M \setminus \{x\} \bullet S_1(z) = S_2(z) \wedge \exists y \in OR(O) \bullet O(S_1)(y) \neq O(S_2)(y))\} ,$$

$$OR(O) = \{x \in M \mid \exists S \in \mathcal{S} \bullet S(x) \neq O(S)(x)\} .$$

$OR(O)$ is the set of all memory elements that are possibly affected by O; and $IR(O)$ is the set of all memory elements that possibly affect elements of $OR(O)$ under O.

Let $(M, B, \mathcal{S}, \mathcal{O})$ be a Maurer computer, let $S_1, S_2 \in \mathcal{S}$, and let $O \in \mathcal{O}$. Then $S_1 \restriction IR(O) = S_2 \restriction IR(O)$ implies $O(S_1) \restriction OR(O) = O(S_2) \restriction OR(O)$. The conditions on the states of a Maurer computer are necessary for this desirable property to hold.

Let $(M, B, \mathcal{S}, \mathcal{O})$ be a Maurer computer, let $O \in \mathcal{O}$, let $M' \subseteq OR(O)$, and let $M'' \subseteq IR(O)$. Then the *region affecting M' under O*, written $RA(M', O)$, and the *region affected by M'' under O*, written $AR(M'', O)$, are the subsets of M defined as follows:

$$RA(M', O) = \{x \in IR(O) \mid AR(\{x\}, O) \cap M' \neq \emptyset\} ,$$

$$AR(M'', O) = \{x \in OR(O) \mid \exists S_1, S_2 \in \mathcal{S} \bullet (\forall z \in IR(O) \setminus M'' \bullet S_1(z) = S_2(z) \wedge O(S_1)(x) \neq O(S_2)(x))\} .$$

$AR(M'', O)$ is the set of all elements of $OR(O)$ that are possibly affected by the elements of M'' under O; and $RA(M', O)$ is the set of all elements of $IR(O)$ that possibly affect elements of M' under O.

[4] The following precedence conventions are used in logical formulas. Operators bind stronger than predicate symbols, and predicate symbols bind stronger than logical connectives and quantifiers. Moreover, $\neg$ binds stronger than $\wedge$ and $\vee$, and $\wedge$ and $\vee$ bind stronger than $\Rightarrow$ and $\Leftrightarrow$. Quantifiers are given the smallest possible scope.

Table 1. Axiom of BTA

$x \trianglelefteq \mathsf{tau} \trianglerighteq y = x \trianglelefteq \mathsf{tau} \trianglerighteq x$	T1

In [17], Maurer gives many results about the relation between the input region and output region of operations, the composition of operations, the decomposition of operations and the existence of operations. In [4], we summarize the main results.

3 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra), a form of process algebra which is tailored to the description of the behaviour of deterministic sequential programs under execution. The behaviours concerned are called *threads*.

In BTA, it is assumed that there is a fixed but arbitrary set of *basic actions* $\mathcal{A}$ with $\mathsf{tau} \notin \mathcal{A}$. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$. BTA has the following constants and operators:

- the *deadlock* constant D;
- the *termination* constant S;
- for each $a \in \mathcal{A}_{\mathsf{tau}}$, a binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_$.

We use infix notation for postconditional composition. We introduce *action prefixing* as an abbreviation: $a \circ p$, where p is a term of BTA, abbreviates $p \trianglelefteq a \trianglerighteq p$.

The intuition is that each basic action performed by a thread is taken as a command to be processed by the execution environment of the thread. The processing of a command may involve a change of state of the execution environment. At completion of the processing of the command, the execution environment produces a reply value. This reply is either T or F and is returned to the thread concerned. Let p and q be closed terms of BTA. Then $p \trianglelefteq a \trianglerighteq q$ will proceed as p if the processing of a leads to the reply T (called a positive reply), and it will proceed as q if the processing of a leads to the reply F (called a negative reply). The action tau plays a special role. Its execution will never change any state and always produces a positive reply.

BTA has only one axiom. This axiom is given in Table 1. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

A *recursive specification* over BTA is a set of equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables and each t_X is a term of BTA that only contains variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation in E. Let t be a term of BTA containing a variable X. Then an occurrence of X in t is *guarded* if t has a subterm of the form $t' \trianglelefteq a \trianglerighteq t''$ containing this occurrence of X. A recursive specification E is *guarded* if all occurrences of variables in the right-hand sides of its equations are guarded or it can be rewritten to such a recursive specification using the equations of E. We are only interested in models of BTA in which guarded

Table 2. Axioms for guarded recursion

$\langle X\|E\rangle = \langle t_X\|E\rangle \quad \mathsf{if}\ X = t_X \in E$	RDP
$E \Rightarrow X = \langle X\|E\rangle \quad \mathsf{if}\ X \in \mathrm{V}(E)$	RSP

Table 3. Approximation induction principle

$\bigwedge_{n\geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$ AIP

recursive specifications have unique solutions, such as the projective limit model of BTA presented in [1, 3]. A thread that is the solution of a finite guarded recursive specification over BTA is called a *finite-state* thread.

We extend BTA with guarded recursion by adding constants for solutions of guarded recursive specifications and axioms concerning these additional constants. For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we add a constant standing for the unique solution of E for X to the constants of BTA. The constant standing for the unique solution of E for X is denoted by $\langle X|E\rangle$. Moreover, we use the following notation. Let t be a term of BTA and E be a guarded recursive specification. Then we write $\langle t|E\rangle$ for t with, for all $X \in \mathrm{V}(E)$, all occurrences of X in t replaced by $\langle X|E\rangle$. We add the axioms for guarded recursion given in Table 2 to the axioms of BTA. In this table, X, t_X and E stand for an arbitrary variable, an arbitrary term of BTA and an arbitrary guarded recursive specification, respectively. Side conditions are added to restrict the variables, terms and guarded recursive specifications for which X, t_X and E stand. The additional axioms for guarded recursion are known as the recursive definition principle (RDP) and the recursive specification principle (RSP). The equations $\langle X|E\rangle = \langle t_X|E\rangle$ for a fixed E express that the constants $\langle X|E\rangle$ make up a solution of E. The conditional equations $E \Rightarrow X = \langle X|E\rangle$ express that this solution is the only one.

We often write X for $\langle X|E\rangle$ if E is clear from the context. It should be borne in mind that, in such cases, we use X as a constant.

The projective limit characterization of process equivalence on threads is based on the notion of a finite approximation of depth n. When for all n these approximations are identical for two given threads, both threads are considered identical. This is expressed by the infinitary conditional equation AIP (Approximation Induction Principle) given in Table 3. Here, following [1, 3], approximation of depth n is phrased in terms of a unary *projection* operator $\pi_n(\_)$. The projection operators are defined inductively by means of the axioms given in Table 4. In this table, a stands for an arbitrary member of $\mathcal{A}_{\mathsf{tau}}$. It happens that RSP follows from AIP.

The structural operational semantics of BTA and its extensions with guarded recursion and projection can be found in [6, 4].

Henceforth, we write $\mathcal{T}_{\mathsf{finrec}}$ for the set of all closed terms of BTA with guarded recursion in which no constants $\langle X|E\rangle$ for infinite E occur. We write $\mathcal{T}_{\mathsf{finrec}}(A)$,

Table 4. Axioms for projection operators

$\pi_0(x) = \mathsf{D}$	P0
$\pi_{n+1}(\mathsf{S}) = \mathsf{S}$	P1
$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	P2
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3

where $A \subseteq \mathcal{A}$, for the set of all closed terms from $\mathcal{T}_{\mathsf{finrec}}$ that only contain basic actions from A.

4 Applying Threads to Maurer Machines

In this section, we introduce Maurer machines and add for each Maurer machine H a binary *apply* operator $\_ \bullet_H \_$ to BTA.

A *Maurer machine* is a tuple $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$, where $(M, B, \mathcal{S}, \mathcal{O})$ is a Maurer computer and:

- $A \subseteq \mathcal{A}$;
- $[\![\_]\!] : A \to (\mathcal{O} \times M)$ is such that for all $S \in \mathcal{S}$ and $a \in A$, $S(\pi_2([\![a]\!])) \in \mathbb{B}$.

The members of A are called the *basic actions* of H, and $[\![\_]\!]$ is called the *basic action interpretation function* of H.

The apply operators associated with Maurer machines are related to the apply operators introduced in [7]. They allow for threads to transform states of the associated Maurer machine by means of its operations. Such state transformations produce either a state of the associated Maurer machine or the *undefined state* $\uparrow$. It is assumed that $\uparrow$ is not a state of any Maurer machine. We extend function restriction to $\uparrow$ by stipulating that $\uparrow \restriction M = \uparrow$ for any set M.[5] The first operand of the apply operator $\_ \bullet_H \_$ associated with Maurer machine $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ must be a term from $\mathcal{T}_{\mathsf{finrec}}(A)$ and its second argument must be a state from $\mathcal{S} \cup \{\uparrow\}$.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine, let $p \in \mathcal{T}_{\mathsf{finrec}}(A)$, and let $S \in \mathcal{S}$. Then $p \bullet_H S$ is the state from $\mathcal{S}$ that results if all basic actions performed by thread p are processed by the Maurer machine H from initial state S. Moreover, let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the processing of a basic action a by H amounts to a state change according to the operation O_a. In the resulting state, the reply produced by H is contained in memory element m_a. If p is S, then there will be no state change. If p is D, then the result is $\uparrow$.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine, and let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the apply operator $\_ \bullet_H \_$ is defined by the equations given in Table 5 (for $a \in A$ and $S \in \mathcal{S}$) and the rule given in Table 6 (for $S \in \mathcal{S}$). We say that $p \bullet_H S$ is *convergent* if $\exists n \in \mathbb{N} \bullet \pi_n(p) \bullet_H S \neq \uparrow$. If $p \bullet_H S$ is convergent, then

[5] In this paper, we use the notation $f \restriction D$, where f is a function and D is a set, for the function g with $\mathrm{dom}(g) = \mathrm{dom}(f) \setminus D$ such that for all $d \in \mathrm{dom}(g)$, $g(d) = f(d)$.

Table 5. Defining equations for apply operator

$x \bullet_H \uparrow = \uparrow$
$\mathsf{S} \bullet_H S = S$
$\mathsf{D} \bullet_H S = \uparrow$
$(\mathsf{tau} \circ x) \bullet_H S = x \bullet_H S$
$(x \trianglelefteq a \trianglerighteq y) \bullet_H S = x \bullet_H O_a(S)$ if $O_a(S)(m_a) = \mathsf{T}$
$(x \trianglelefteq a \trianglerighteq y) \bullet_H S = y \bullet_H O_a(S)$ if $O_a(S)(m_a) = \mathsf{F}$

Table 6. Rule for divergence

$\bigwedge_{n>0} \pi_n(x) \bullet_H S = \uparrow \Rightarrow x \bullet_H S = \uparrow$

the *length of the computation* of $p \bullet_H S$, written $|p \bullet_H S|$, is the smallest $n \in \mathbb{N}$ such that $\pi_n(p) \bullet_H S \neq \uparrow$. If $p \bullet_H S$ is not convergent, then $|p \bullet_H S|$ is undefined. We say that $p \bullet_H S$ is *divergent* if $p \bullet_H S$ is not convergent. Note that the rule from Table 6 can be read as follows: if $x \bullet_H S$ is divergent, then it equals $\uparrow$.

We introduce some auxiliary notions, which are useful in proofs. Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine, and let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the *step* relation $\_ \vdash_H \_ \subseteq (\mathcal{T}_{\mathsf{finrec}}(A) \times \mathcal{S}) \times (\mathcal{T}_{\mathsf{finrec}}(A) \times \mathcal{S})$ is inductively defined as follows:

- if $p = \mathsf{tau} \circ p'$, then $(p, S) \vdash_H (p', S)$;
- if $O_a(S)(m_a) = \mathsf{T}$ and $p = p' \trianglelefteq a \trianglerighteq p''$, then $(p, S) \vdash_H (p', O_a(S))$;
- if $O_a(S)(m_a) = \mathsf{F}$ and $p = p' \trianglelefteq a \trianglerighteq p''$, then $(p, S) \vdash_H (p'', O_a(S))$.

If $(p, S) \vdash_H (p', S')$, then $p \bullet_H S = p' \bullet_H S'$. Moreover, let $p \in \mathcal{T}_{\mathsf{finrec}}(A)$, and let $S \in \mathcal{S}$. Then the *full path* of $p \bullet_H S$ is the unique full path in $\_ \vdash_H \_$ from (p, S). A *full* path in $\_ \vdash_H \_$ is one of the following:

- a finite path $\langle (p_0, S_0), \ldots, (p_n, S_n) \rangle$ in $\_ \vdash_H \_$ such that there exists no $(p_{n+1}, S_{n+1}) \in \mathcal{T}_{\mathsf{finrec}}(A) \times \mathcal{S}$ with $(p_n, S_n) \vdash_H (p_{n+1}, S_{n+1})$;
- an infinite path $\langle (p_0, S_0), (p_1, S_1), \ldots \rangle$ in $\_ \vdash_H \_$.

If $p \bullet_H S$ is convergent, then its full path is a path of length $|p \bullet_H S|$ from (p, S) to (S, S'), where $S' = p \bullet_H S$. Such a full path is also called a *computation*.

Henceforth, we write $\_ \vdash_H^* \_$ for the reflexive and transitive closure of $\_ \vdash_H \_$.

In the definition of a Maurer machine, we could have taken a function $[\![\_]\!]$ that associates with each $a \in A$ a triple $(n_a, O_a, m_a) \in M \times \mathcal{O} \times M$ such that $S(n_a), S(m_a) \in \mathbb{B}$ for all $S \in \mathcal{S}$. In that case, $S(n_a)$ would indicate whether basic action a is enabled in state S, i.e. whether the processing of a is not blocked in state S. In this paper, we consider only threads that are behaviours of deterministic sequential programs under execution. For such behaviours, it is not at all interesting to take into account the possibility that some basic actions are not always enabled. Therefore, it is assumed that all basic actions of a Maurer machine are enabled in all states. Under this assumption, it is sufficient that the

function $[\![\_]\!]$ associates with each $a \in A$ a pair $(O_a, m_a) \in \mathcal{O} \times M$ as in the definition given at the beginning of this section.

5 Program Algebra

In this section, we review PGA (ProGram Algebra), an algebra of sequential programs based on the idea that sequential programs are in essence sequences of instructions. PGA provides a program notation for finite-state threads. A hierarchy of program notations that provide more and more sophisticated programming features are rooted in PGA (see [3]).

In PGA, it is assumed that there is a fixed but arbitrary set $\mathfrak{A}$ of *basic instructions*. PGA has the following *primitive instructions*:

- for each $a \in \mathfrak{A}$, a *void basic instruction* a;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;
- for each $k \in \mathbb{N}$, a *forward jump instruction* $\#k$;
- a *termination instruction* !.

We write $\mathfrak{I}$ for the set of all primitive instructions.

The intuition is that the execution of a basic instruction a may modify a state and produces T or F at its completion. In the case of a positive test instruction $+a$, basic instruction a is executed and execution proceeds with the next primitive instruction if T is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one. In the case where T is produced and there is not at least one subsequent primitive instruction and in the case where F is produced and there are not at least two subsequent primitive instructions, deadlock occurs. In the case of a negative test instruction $-a$, the role of the value produced is reversed. In the case of a void basic instruction a, the value produced is disregarded: execution always proceeds as if T is produced. The effect of a forward jump instruction $\#k$ is that execution proceeds with the k-th next instruction of the program concerned. If k equals 0 or the k-th next instruction does not exist, then $\#k$ results in deadlock. The effect of the termination instruction ! is that execution terminates.

The thread extraction operator introduced below, together with the apply operator introduced in Section 4 make it possible to associate operations of Maurer machines with basic instructions, and consequently with primitive instructions of PGA.

PGA has the following constants and operators:

- for each $u \in \mathfrak{I}$, an *instruction* constant u ;
- the binary *concatenation* operator $\_ \,;\, \_$;
- the unary *repetition* operator $\_^\omega$.

Closed terms of PGA are considered to denote programs. The intuition is that a program is in essence a non-empty, finite or infinite sequence of primitive instructions. These sequences are called *single pass instruction sequences*

Table 7. Axioms of PGA

$(X\,;Y)\,;Z = X\,;(Y\,;Z)$	PGA1
$(X^n)^\omega = X^\omega$	PGA2
$X^\omega\,;Y = X^\omega$	PGA3
$(X\,;Y)^\omega = X\,;(Y\,;X)^\omega$	PGA4

Table 8. Defining equations for thread extraction operator

$\lvert a\rvert = a \circ \mathsf{D}$	$\lvert \#k\rvert = \mathsf{D}$
$\lvert a\,;X\rvert = a \circ \lvert X\rvert$	$\lvert \#0\,;X\rvert = \mathsf{D}$
$\lvert +a\rvert = a \circ \mathsf{D}$	$\lvert \#1\,;X\rvert = \lvert X\rvert$
$\lvert +a\,;X\rvert = \lvert X\rvert \trianglelefteq a \trianglerighteq \lvert \#2\,;X\rvert$	$\lvert \#k+2\,;u\rvert = \mathsf{D}$
$\lvert -a\rvert = a \circ \mathsf{D}$	$\lvert \#k+2\,;u\,;X\rvert = \lvert \#k+1\,;X\rvert$
$\lvert -a\,;X\rvert = \lvert \#2\,;X\rvert \trianglelefteq a \trianglerighteq \lvert X\rvert$	$\lvert !\rvert = \mathsf{S}$
	$\lvert !\,;X\rvert = \mathsf{S}$

Table 9. Rule for cyclic jump chains

$X \cong \#0\,;Y \;\Rightarrow\; \lvert X\rvert = \mathsf{D}$

because PGA has been designed to enable single pass execution of instruction sequences: each instruction can be dropped after it has been executed. Programs are considered to be equal if they represent the same single pass instruction sequence. The axioms for instruction sequence equivalence are given in Table 7. In this table, n stands for an arbitrary natural number greater than 0. For each $n > 0$, the term X^n is defined by induction on n as follows: $X^1 = X$ and $X^{n+1} = X\,;X^n$. The *unfolding* equation $X^\omega = X\,;X^\omega$ is derivable. Each closed term of PGA is derivably equal to a term in *canonical form*, i.e. a term of the form P or $P\,;Q^\omega$, where P and Q are closed terms of PGA that do not contain the repetition operator.

Each closed term of PGA is considered to denote a program of which the behaviour is a finite-state thread, taking the set $\mathfrak{A}$ of basic instructions for the set $\mathcal{A}$ of actions. The *thread extraction* operator $\lvert\_\rvert$ assigns a thread to each program. The thread extraction operator is defined by the equations given in Table 8 (for $a \in \mathfrak{A}$, $k \in \mathbb{N}$ and $u \in \mathfrak{I}$) and the rule given in Table 9. This rule is expressed in terms of the *structural congruence* predicate $\_ \cong \_$, which is defined by the formulas given in Table 10 (for $n, m, k \in \mathbb{N}$ and $u_1, \ldots, u_n, v_1, \ldots, v_{m+1} \in \mathfrak{I}$).

The equations given in Table 8 do not cover the case where there is a cyclic chain of forward jumps. Programs are structural congruent if they are the same after removing all chains of forward jumps in favour of direct jumps. Because a cyclic chain of forward jumps corresponds to #0, the rule from Table 9 can be read as follows: if X starts with a cyclic chain of forward jumps, then $\lvert X\rvert$

Table 10. Defining formulas for structural congruence predicate

$\#n+1\,;u_1\,;\ldots;u_n\,;\#0 \cong \#0\,;u_1\,;\ldots;u_n\,;\#0$
$\#n+1\,;u_1\,;\ldots;u_n\,;\#m \cong \#m+n+1\,;u_1\,;\ldots;u_n\,;\#m$
$(\#n+k+1\,;u_1\,;\ldots;u_n)^\omega \cong (\#k\,;u_1\,;\ldots;u_n)^\omega$
$\#m+n+k+2\,;u_1\,;\ldots;u_n\,;(v_1\,;\ldots;v_{m+1})^\omega \cong$
$\#n+k+1\,;u_1\,;\ldots;u_n\,;(v_1\,;\ldots;v_{m+1})^\omega$
$X \cong X$
$X_1 \cong Y_1 \wedge X_2 \cong Y_2 \Rightarrow X_1\,;X_2 \cong Y_1\,;Y_2 \wedge {X_1}^\omega \cong {Y_1}^\omega$

equals D. It is easy to see that the thread extraction operator assigns the same thread to structurally congruent programs. Therefore, the rule from Table 9 can be replaced by the following generalization: $X \cong Y \Rightarrow |X| = |Y|$.

Let E be a finite guarded recursive specification over BTA with $\mathrm{V}(E) = \{X_1, \ldots, X_n\}$, and let P_1, …, P_n be closed terms of PGA. Let E' be the set of equations that results from replacing in E all occurrences of X_1 by $|P_1|$ and … and all occurrences of X_n by $|P_n|$. If E' can be obtained by applications of axioms PGA1–PGA4, the defining equations for the thread extraction operator and the rule for cyclic jump chains, then $|P_1|$ is the solution of E for X_1. Such a finite guarded recursive specification can always be found. Thus, the behaviour of each closed PGA term, is a thread that is definable by a finite guarded recursive specification over BTA. Moreover, each finite guarded recursive specification over BTA can be translated to a PGA program of which the behaviour is the solution of the finite guarded recursive specification concerned.

Closed terms of PGA are loosely called PGA *programs*. PGA programs in which the repetition operator do not occur are called *finite* PGA programs. Henceforth, we write $\mathcal{P}_{\mathsf{fin}}$ for the set of all finite PGA programs. We write $\mathcal{P}_{\mathsf{fin}}(A)$, where $A \subseteq \mathfrak{A}$, for the set of all closed terms from $\mathcal{P}_{\mathsf{fin}}$ that only contain basic instructions from A.

In the remainder of this paper, with the exception of Section 11, we consider only finite PGA programs.

6 Stored Programs

In this short section, we make precise how to represent PGA programs in the memory of a Maurer machine.

It is assumed that a fixed but arbitrary finite set $\mathsf{M}_{\mathsf{prog}}$ and a fixed but arbitrary bijection $\mathsf{m}_{\mathsf{prog}} : [0, \mathit{card}(\mathsf{M}_{\mathsf{prog}}) - 1] \to \mathsf{M}_{\mathsf{prog}}$ have been given. $\mathsf{M}_{\mathsf{prog}}$ is called the *program memory*. We write $\mathit{size}(\mathsf{M}_{\mathsf{prog}})$ for $\mathit{card}(\mathsf{M}_{\mathsf{prog}})$. Let $n, n' \in [0, \mathit{size}(\mathsf{M}_{\mathsf{prog}}) - 1]$ be such that $n \leq n'$. Then, we write $\mathsf{M}_{\mathsf{prog}}[n]$ for $\mathsf{m}_{\mathsf{prog}}(n)$, and $\mathsf{M}_{\mathsf{prog}}[n, n']$ for $\{\mathsf{m}_{\mathsf{prog}}(k) \mid n \leq k \leq n'\}$.

The program memory is a memory of which the elements can be addressed by means of members of $[0, \mathit{size}(\mathsf{M}_{\mathsf{prog}}) - 1]$. We write $\mathsf{MA}_{\mathsf{prog}}$ for $[0, \mathit{size}(\mathsf{M}_{\mathsf{prog}}) - 1]$ and $\mathsf{MA}'_{\mathsf{prog}}$ for $[0, \mathit{size}(\mathsf{M}_{\mathsf{prog}})]$.

The program memory elements are meant for containing the primitive instructions that form part of a finite PGA program.

We write $\mathfrak{I}_{\mathsf{prog}}$ for $\mathfrak{I} \setminus \{\#k \mid k > \mathit{size}(\mathsf{M}_{\mathsf{prog}})-1\}$. $\mathfrak{I}_{\mathsf{prog}}$ is the *program memory base set*. We write $\mathsf{S}_{\mathsf{prog}}$ for the set of all functions $S_{\mathsf{prog}} : \mathsf{M}_{\mathsf{prog}} \to \mathfrak{I}_{\mathsf{prog}}$.

Let $P = u_1;\ldots;u_n \in \mathcal{P}_{\mathsf{fin}}$ with $n \leq \mathit{size}(\mathsf{M}_{\mathsf{prog}})$. Then the *stored representation* of P, written $\mathsf{s}_{\mathsf{prog}}(P)$, is the unique function $s_{\mathsf{prog}} : \mathsf{M}_{\mathsf{prog}}[0, n-1] \to \mathfrak{I}_{\mathsf{prog}}$ such that for all $i \in [0, n-1]$, $s_{\mathsf{prog}}(\mathsf{M}_{\mathsf{prog}}[i]) = u_{i+1}$. We call $\mathsf{s}_{\mathsf{prog}}(P)$ a *stored program*.

Note that $\mathsf{s}_{\mathsf{prog}}(u_1 ; \ldots ; u_n)$ is not defined if $n > \mathit{size}(\mathsf{M}_{\mathsf{prog}})$. The size of the program memory restricts the programs that can be stored.

7 Non-Pipelined Instruction Processing

In this section, we model a micro-architecture with non-pipelined instruction processing. We do not make the instruction set architecture for which this micro-architecture is modelled explicit. We start from an arbitrary Maurer machine and enhance it. That Maurer machine determines the instruction set architecture for which a micro-architecture is modelled. However, there are Maurer machines for which the enhancement is primarily intended. Those Maurer machines will be introduced in Section 12. Henceforth, we write PGA instruction for primitive instruction of PGA.

We enhance Maurer machines by extending the memory with a *program memory* ($\mathsf{M}_{\mathsf{prog}}$), a *program counter upper bound register* (pcbr), a *program counter* (pc), an *instruction register* (ir), a *decoded instruction type register* (ditr), a *basic action register* (bar), a *displacement register* (dr), an *executed instruction type register* (eitr), an *instruction reply register* (irr), a *fetch reply register* ($\mathsf{rr}_{\mathsf{fetch}}$), a *pre-process reply register* ($\mathsf{rr}_{\mathsf{prep}}$), an *execute reply register* ($\mathsf{rr}_{\mathsf{exec}}$) and a *post-process reply register* ($\mathsf{rr}_{\mathsf{postp}}$), and the operation set with a *fetch* operation (O_{fetch}), a *pre-process* operation (O_{prep}), an *execute* operation (O_{exec}) and a *post-process* operation (O_{postp}). Moreover, we replace the basic actions of the original Maurer machine by basic actions fetch, prep, exec and postp, with which the operations O_{fetch}, O_{prep}, O_{exec} and O_{postp} are associated. The resulting Maurer machines are called SP-NPL-enhancements. SP stands for stored program and NPL stands for non-pipelined instruction processing. In SP-NPL-enhancements of Maurer machines, the five *instruction types* bsc, ptst, ntst, fjmp and term are distinguished. These types correspond to the five kinds of PGA instructions introduced in Section 5. Henceforth, we write IT for the set $\{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}, \mathsf{fjmp}, \mathsf{term}\}$. The registers $\mathsf{rr}_{\mathsf{fetch}}$, $\mathsf{rr}_{\mathsf{prep}}$, $\mathsf{rr}_{\mathsf{exec}}$ and $\mathsf{rr}_{\mathsf{postp}}$ are the reply registers of the execution handling operations O_{fetch}, O_{prep}, O_{exec} and O_{postp}, respectively. Henceforth, we write M'_{rr} for $\{\mathsf{rr}_{\mathsf{fetch}}, \mathsf{rr}_{\mathsf{prep}}, \mathsf{rr}_{\mathsf{exec}}, \mathsf{rr}_{\mathsf{postp}}\}$.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine such that $M \cap \mathsf{M}_{\mathsf{prog}} = \emptyset$, $\mathsf{pcbr}, \mathsf{pc}, \mathsf{ir}, \mathsf{ditr}, \mathsf{bar}, \mathsf{dr}, \mathsf{eitr}, \mathsf{irr} \notin M$, $M \cap M'_{\mathrm{rr}} = \emptyset$ and $\mathsf{fetch}, \mathsf{prep}, \mathsf{exec}, \mathsf{postp} \notin A$, and let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the *SP-NPL-enhancement* of H is the Maurer machine $H' = (M', B', \mathcal{S}', \mathcal{O}', A', [\![\_]\!]')$ such that

$$M' = M \cup \mathsf{M}_{\mathsf{prog}} \cup \{\mathsf{pcbr}, \mathsf{pc}, \mathsf{ir}, \mathsf{ditr}, \mathsf{bar}, \mathsf{dr}, \mathsf{eitr}, \mathsf{irr}\} \cup M'_{\mathrm{rr}} \,,$$
$$B' = B \cup \mathsf{MA}'_{\mathsf{prog}} \cup \mathfrak{I}_{\mathsf{prog}} \cup IT \cup A \cup \mathbb{B} \,,$$

$$
\begin{aligned}
\mathcal{S}' &= \{S' : M' \to B' \mid \\
&\qquad S' \restriction M \in \mathcal{S} \wedge S' \restriction \mathsf{M}_{\mathsf{prog}} \in \mathsf{S}_{\mathsf{prog}} \wedge S'(\mathsf{pcbr}) \in \mathsf{MA}_{\mathsf{prog}} \wedge \\
&\qquad S'(\mathsf{pc}) \in \mathsf{MA}'_{\mathsf{prog}} \wedge S'(\mathsf{ir}) \in \mathfrak{I}_{\mathsf{prog}} \wedge \\
&\qquad S'(\mathsf{ditr}) \in IT \wedge S'(\mathsf{bar}) \in A \wedge S'(\mathsf{dr}) \in \mathsf{MA}_{\mathsf{prog}} \wedge \\
&\qquad S'(\mathsf{eitr}) \in IT \wedge S'(\mathsf{irr}) \in \mathbb{B} \wedge \\
&\qquad S'(\mathsf{rr}_{\mathsf{fetch}}) \in \mathbb{B} \wedge S'(\mathsf{rr}_{\mathsf{prep}}) \in \mathbb{B} \wedge S'(\mathsf{rr}_{\mathsf{exec}}) \in \mathbb{B} \wedge S'(\mathsf{rr}_{\mathsf{postp}}) \in \mathbb{B}\} \;, \\
\mathcal{O}' &= \{O' : \mathcal{S}' \to \mathcal{S}' \mid \\
&\qquad \exists O \in \mathcal{O} \bullet \forall S' \in \mathcal{S}' \bullet \\
&\qquad\quad (O'(S') \restriction M = O(S' \restriction M) \wedge O'(S') \restriction (M' \setminus M) = S' \restriction (M' \setminus M))\} \\
&\qquad \cup \{O_{\mathsf{fetch}}, O_{\mathsf{prep}}, O_{\mathsf{exec}}, O_{\mathsf{postp}}\} \;, \\
A' &= \{\mathsf{fetch}, \mathsf{prep}, \mathsf{exec}, \mathsf{postp}\} \;, \\
[\![a]\!]' &= (O_a, \mathsf{rr}_a) \quad \text{for all } a \in A' \;.
\end{aligned}
$$

O_{fetch} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
O_{\mathsf{fetch}}(S') \restriction M &= S' \restriction M \;, \\
O_{\mathsf{fetch}}(S') \restriction \mathsf{M}_{\mathsf{prog}} &= S' \restriction \mathsf{M}_{\mathsf{prog}} \;, \\
O_{\mathsf{fetch}}(S')(\mathsf{pcbr}) &= S'(\mathsf{pcbr}) \;, \\
O_{\mathsf{fetch}}(S')(\mathsf{pc}) &= S'(\mathsf{pc}) + 1 && \text{if } S'(\mathsf{pc}) + 1 \leq S'(\mathsf{pcbr}) \;, \\
O_{\mathsf{fetch}}(S')(\mathsf{pc}) &= S'(\mathsf{pc}) && \text{if } S'(\mathsf{pc}) + 1 > S'(\mathsf{pcbr}) \;, \\
O_{\mathsf{fetch}}(S')(\mathsf{ir}) &= S'(\mathsf{M}_{\mathsf{prog}}[S'(\mathsf{pc})]) && \text{if } S'(\mathsf{pc}) \leq S'(\mathsf{pcbr}) \;, \\
O_{\mathsf{fetch}}(S')(\mathsf{ir}) &= \#0 && \text{if } S'(\mathsf{pc}) > S'(\mathsf{pcbr}) \;, \\
O_{\mathsf{fetch}}(S') \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} &= S' \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} \;, \\
O_{\mathsf{fetch}}(S') \restriction \{\mathsf{eitr}, \mathsf{irr}\} &= S' \restriction \{\mathsf{eitr}, \mathsf{irr}\} \;, \\
O_{\mathsf{fetch}}(S')(\mathsf{rr}_{\mathsf{fetch}}) &= \mathsf{T} && \text{if } S'(\mathsf{pc}) \leq S'(\mathsf{pcbr}) \;, \\
O_{\mathsf{fetch}}(S')(\mathsf{rr}_{\mathsf{fetch}}) &= \mathsf{F} && \text{if } S'(\mathsf{pc}) > S'(\mathsf{pcbr}) \;, \\
O_{\mathsf{fetch}}(S') \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{fetch}}\}) &= S' \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{fetch}}\}) \;.
\end{aligned}
$$

O_{prep} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
O_{\mathsf{prep}}(S') \restriction M &= S' \restriction M \;, \\
O_{\mathsf{prep}}(S') \restriction \mathsf{M}_{\mathsf{prog}} &= S' \restriction \mathsf{M}_{\mathsf{prog}} \;, \\
O_{\mathsf{prep}}(S')(\mathsf{pcbr}) &= S'(\mathsf{pcbr}) \;, \\
O_{\mathsf{prep}}(S') \restriction \{\mathsf{pc}, \mathsf{ir}\} &= S' \restriction \{\mathsf{pc}, \mathsf{ir}\} \;, \\
O_{\mathsf{prep}}(S')(\mathsf{ditr}) &= \pi_1(dec(S')) \;, \\
O_{\mathsf{prep}}(S')(\mathsf{bar}) &= \pi_2(dec(S')) \;, \\
O_{\mathsf{prep}}(S')(\mathsf{dr}) &= \pi_3(dec(S')) \;, \\
O_{\mathsf{prep}}(S') \restriction \{\mathsf{eitr}, \mathsf{irr}\} &= S' \restriction \{\mathsf{eitr}, \mathsf{irr}\} \;, \\
O_{\mathsf{prep}}(S')(\mathsf{rr}_{\mathsf{prep}}) &= \mathsf{T} \;, \\
O_{\mathsf{prep}}(S') \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{prep}}\}) &= S' \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{prep}}\}) \;,
\end{aligned}
$$

where $dec : \mathcal{S}' \to IT \times A \times \mathsf{MA}_{\mathsf{prog}}$ is defined as follows:

$$
\begin{aligned}
dec(S') &= (\mathsf{bsc}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = a \ , \\
dec(S') &= (\mathsf{ptst}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = +a \ , \\
dec(S') &= (\mathsf{ntst}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = -a \ , \\
dec(S') &= (\mathsf{fjmp}, S'(\mathsf{bar}), k) && \text{if } S'(\mathsf{ir}) = \#k \ , \\
dec(S') &= (\mathsf{term}, S'(\mathsf{bar}), S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = ! \ .
\end{aligned}
$$

O_{exec} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
O_{\mathsf{exec}}(S') \restriction M &= O_{S'(\mathsf{bar})}(S' \restriction M) && \text{if } opc(S') \ , \\
O_{\mathsf{exec}}(S') \restriction M &= S' \restriction M && \text{if } \neg\ opc(S') \ , \\
O_{\mathsf{exec}}(S') \restriction \mathsf{M}_{\mathsf{prog}} &= S' \restriction \mathsf{M}_{\mathsf{prog}} \ , \\
O_{\mathsf{exec}}(S')(\mathsf{pcbr}) &= S'(\mathsf{pcbr}) \ , \\
O_{\mathsf{exec}}(S') \restriction \{\mathsf{pc}, \mathsf{ir}\} &= S' \restriction \{\mathsf{pc}, \mathsf{ir}\} \ , \\
O_{\mathsf{exec}}(S') \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} &= S' \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} \ , \\
O_{\mathsf{exec}}(S')(\mathsf{eitr}) &= S'(\mathsf{ditr}) \ , \\
O_{\mathsf{exec}}(S')(\mathsf{irr}) &= O_{S'(\mathsf{bar})}(S' \restriction M)(m_{S'(\mathsf{bar})}) && \text{if } opc(S') \ , \\
O_{\mathsf{exec}}(S')(\mathsf{irr}) &= \mathsf{T} && \text{if } \neg\ opc(S') \ , \\
O_{\mathsf{exec}}(S')(\mathsf{rr}_{\mathsf{exec}}) &= \mathsf{T} \ , \\
O_{\mathsf{exec}}(S') \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{exec}}\}) &= S' \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{exec}}\}) \ ,
\end{aligned}
$$

where $opc : \mathcal{S}' \to \mathbb{B}$ is defined as follows:

$$opc(S') = \mathsf{T} \text{ iff } S'(\mathsf{ditr}) \in \{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}\} \ .$$

O_{postp} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
O_{\mathsf{postp}}(S') \restriction M &= S' \restriction M \ , \\
O_{\mathsf{postp}}(S') \restriction \mathsf{M}_{\mathsf{prog}} &= S' \restriction \mathsf{M}_{\mathsf{prog}} \ , \\
O_{\mathsf{postp}}(S')(\mathsf{pcbr}) &= S'(\mathsf{pcbr}) \ , \\
O_{\mathsf{postp}}(S')(\mathsf{pc}) &= pcu(S') \ , \\
O_{\mathsf{postp}}(S')(\mathsf{ir}) &= S'(\mathsf{ir}) \ , \\
O_{\mathsf{postp}}(S') \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} &= S' \restriction \{\mathsf{ditr}, \mathsf{bar}, \mathsf{dr}\} \ , \\
O_{\mathsf{postp}}(S') \restriction \{\mathsf{eitr}, \mathsf{irr}\} &= S' \restriction \{\mathsf{eitr}, \mathsf{irr}\} \ , \\
O_{\mathsf{postp}}(S')(\mathsf{rr}_{\mathsf{postp}}) &= \mathsf{T} && \text{if } S'(\mathsf{eitr}) \neq \mathsf{term} \ , \\
O_{\mathsf{postp}}(S')(\mathsf{rr}_{\mathsf{postp}}) &= \mathsf{F} && \text{if } S'(\mathsf{eitr}) = \mathsf{term} \ , \\
O_{\mathsf{postp}}(S') \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{postp}}\}) &= S' \restriction (M'_{\mathrm{rr}} \setminus \{\mathsf{rr}_{\mathsf{postp}}\}) \ ,
\end{aligned}
$$

where $pcu : \mathcal{S}' \to \mathsf{MA}'_{\mathsf{prog}}$ is defined as follows:

$$
\begin{aligned}
pcu(S') = S'(\mathsf{pc}) \quad & \text{if } S'(\mathsf{eitr}) = \mathsf{bsc} \vee \\
& \quad S'(\mathsf{eitr}) = \mathsf{ptst} \wedge S'(\mathsf{irr}) = \mathsf{T} \vee \\
& \quad S'(\mathsf{eitr}) = \mathsf{ntst} \wedge S'(\mathsf{irr}) = \mathsf{F} \vee \\
& \quad S'(\mathsf{eitr}) = \mathsf{term} \ , \\
pcu(S') = S'(\mathsf{pc}) + 1 \quad & \text{if } (S'(\mathsf{eitr}) = \mathsf{ptst} \wedge S'(\mathsf{irr}) = \mathsf{F} \vee \\
& \quad S'(\mathsf{eitr}) = \mathsf{ntst} \wedge S'(\mathsf{irr}) = \mathsf{T}) \wedge \\
& \quad S'(\mathsf{pc}) + 1 \leq S'(\mathsf{pcbr}) \ , \\
pcu(S') = S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) \quad & \text{if } S'(\mathsf{eitr}) = \mathsf{fjmp} \wedge S'(\mathsf{dr}) \neq 0 \wedge \\
& \quad S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) \leq S'(\mathsf{pcbr}) \ , \\
pcu(S') = S'(\mathsf{pcbr}) + 1 \quad & \text{if } (S'(\mathsf{eitr}) = \mathsf{ptst} \wedge S'(\mathsf{irr}) = \mathsf{F} \vee \\
& \quad S'(\mathsf{eitr}) = \mathsf{ntst} \wedge S'(\mathsf{irr}) = \mathsf{T}) \wedge \\
& \quad S'(\mathsf{pc}) + 1 > S'(\mathsf{pcbr}) \vee \\
& \quad S'(\mathsf{eitr}) = \mathsf{fjmp} \wedge \\
& \quad (S'(\mathsf{dr}) = 0 \vee \\
& \quad S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) > S'(\mathsf{pcbr})) \ .
\end{aligned}
$$

Figure 7 shows the structure of an SP-NPL-enhancement. The program counter pc contains the address of the program memory element from which a PGA instruction is fetched next, unless its contents is greater than the highest program address (contained in pcbr). Fetched PGA instructions are stored in ir. The program counter is incremented at every fetch. Pre-processing amounts to decoding the PGA instruction stored in ir: the type of that PGA instruction is stored in ditr, the basic action involved is stored in bar if it is not a jump

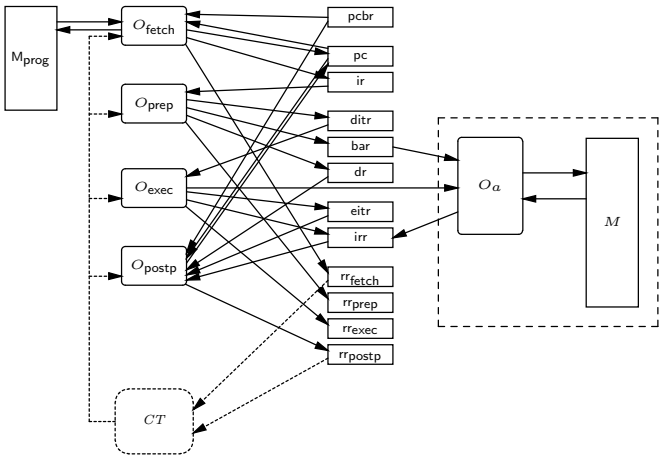

Fig. 1. Structure of an SP-NPL-enhancement

or termination instruction and the displacement is stored in dr if it is a jump instruction. Execution does not deal with jump and termination instructions. They are dealt with by post-processing. Post-processing amounts to adjusting the program counter and recognizing termination. The program counter is adjusted on a positive test instruction that has given a negative reply, a negative test instruction that has given a positive reply and a jump instruction.

Essential information about the last fetched PGA instruction is forwarded from one execution handling operation to the next: from O_{fetch} to O_{prep} via ir, from O_{prep} to O_{exec} via ditr and either bar or dr, from O_{exec} to O_{postp} via eitr. Moreover, each execution handling operation has a reply register by itself. All this fits in well with the pipelined variant of SP-NPL-enhancements that will be introduced in Section 8.

Because the memory is extended with only finitely many memory elements, it is easy to check, using Proposition IV from [17], that the SP-NPL-enhancement of a Maurer machine is a Maurer machine indeed. The same remark applies to the SP-PL-enhancement of a Maurer machine introduced in Section 8 as well.

Consider the guarded recursive specification over BTA that consists of the following equation:

$$CT = (\mathsf{prep} \circ \mathsf{exec} \circ (CT \trianglelefteq \mathsf{postp} \trianglerighteq \mathsf{S})) \trianglelefteq \mathsf{fetch} \trianglerighteq \mathsf{D} \;.$$

Let P be a finite PGA program. Then applying thread $|P|$ to a state of Maurer machine H has the same effect as applying the execution handling thread CT to the corresponding state of the SP-NPL-enhancement of H in which the program memory contains the stored representation of P.

Theorem 1 (SP-NPL-enhancement). *Let $H' = (M', B', \mathcal{S}', \mathcal{O}', A', [\![\_]\!]')$ be the SP-NPL-enhancement of $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$, let $P = u_1 \,;\ldots;\, u_n \in \mathcal{P}_{\mathsf{fin}}(A)$ be such that $n \leq \mathit{size}(\mathsf{M}_{\mathsf{prog}})$, let $S'_0 \in \mathcal{S}'$ be such that $S'_0 \restriction \mathsf{M}_{\mathsf{prog}}[0, n-1] = \mathsf{s}_{\mathsf{prog}}(P)$, $S'_0(\mathsf{pcbr}) = n-1$ and $S'_0(\mathsf{pc}) = 0$. Then $|P| \bullet_H (S'_0 \restriction M) = (CT \bullet_{H'} S'_0) \restriction M$.*

Proof. Let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$, and let $(O_a, \mathsf{rr}_a) = [\![a]\!]'$ for all $a \in A'$. Then it is easy to see that for all $S' \in \mathcal{S}'$ and $a \in A$ such that $S'(\mathsf{pc}) \leq S'(\mathsf{pcbr})$ and $S'(\mathsf{M}_{\mathsf{prog}}[S'(\mathsf{pc})]) \in \{a, +a, -a\}$:

$$O_{\mathsf{postp}}(O_{\mathsf{exec}}(O_{\mathsf{prep}}(O_{\mathsf{fetch}}(S')))) \restriction M = O_a(S' \restriction M) \;, \tag{1}$$

$$O_{\mathsf{postp}}(O_{\mathsf{exec}}(O_{\mathsf{prep}}(O_{\mathsf{fetch}}(S'))))(\mathsf{irr}) \;= O_a(S' \restriction M)(m_a) \;; \tag{2}$$

and it is easy to see that for all $S' \in \mathcal{S}'$ and $a \in A$ such that $S'(\mathsf{pc}) \leq S'(\mathsf{pcbr})$ and $S'(\mathsf{M}_{\mathsf{prog}}[S'(\mathsf{pc})]) \notin \{a, +a, -a\}$:

$$O_{\mathsf{postp}}(O_{\mathsf{exec}}(O_{\mathsf{prep}}(O_{\mathsf{fetch}}(S')))) \restriction M = S' \restriction M \;. \tag{3}$$

Let (p'_i, S'_i) be the $i{+}1$-th element in the full path of $CT \bullet_{H'} S'_0$. Then it is easy to prove by induction on i that

$$\begin{aligned} p'_{4i+4} &= CT \quad \text{if } S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \wedge S'_{4i+4}(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{T} \;, \\ p'_{4i+4} &= \mathsf{S} \quad \text{if } S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \wedge S'_{4i+4}(\mathsf{rr}_{\mathsf{postp}}) = \mathsf{F} \;, \\ p'_{4i+1} &= \mathsf{D} \quad \text{if } S'_{4i+1}(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{F} \;. \end{aligned} \tag{4}$$

Let (p_i, S_i) be the i+1-th element in the full path of $|P| \bullet_H (S'_0 \restriction M)$, and let (p'_i, S'_i) be the i+1-th element in the full path of $CT \bullet_{H'} S'_0$ of which the first component equals CT, S or D and the second component, say S', satisfies $S'(\mathsf{M}_{\mathsf{prog}}[S'(\mathsf{pc})]) \neq \#k$ for all $k \in \mathsf{MA}_{\mathsf{prog}}$. Then, using (1), (2), (3) and (4), it is straightforward to prove by induction on i and case distinction on the structure of finite PGA programs that:

- $p_i = |\mathsf{s}_{\mathsf{prog}}(P)(\mathsf{M}_{\mathsf{prog}}[S'_{4i}(\mathsf{pc})]) \,;\, \ldots \,;\, \mathsf{s}_{\mathsf{prog}}(P)(\mathsf{M}_{\mathsf{prog}}[n-1])|$;
- $S_i = S'_{4i} \restriction M$

From this, the theorem follows immediately. □

Henceforth, execution handling threads, like CT, are called *power threads*.

8 Pipelined Instruction Processing

In this section, we model a micro-architecture with pipelined instruction processing which is a variant of the micro-architecture with non-pipelined instruction processing modelled in Section 8. In the latter micro-architecture, PGA instructions are processed after one another, whereas, in the micro-architecture modelled here, four PGA instructions can be simultaneously overlapped in processing. We start again from an arbitrary Maurer machine and enhance it.

We enhance Maurer machines by extending the memory as in the case of SP-NPL-enhancements and additionally with an *instruction skip flag* (isf), a *jump decoded flag* (jdf), a *jump processed flag* (jpf), a *pipeline status register* (plsr) and a *reply register* (rr), and the operation set with a *step* operation (O_{step}), a *pipeline control* operation (O_{plctr}) and a *halt* operation (O_{halt}). Moreover, we replace the basic actions of the original Maurer machine by basic actions step, plctr and halt with which the extra operations O_{step}, O_{plctr} and O_{halt} are associated. The resulting Maurer machines are called SP-PL-enhancements. SP stands again for stored program and PL stands for pipelined instruction processing. In SP-PL-enhancements of Maurer machines, the four *pipeline stages* $\mathsf{fetchst}$, prepst, execst and $\mathsf{postpst}$ are distinguished. Henceforth, we write PS for $\{\mathsf{fetchst}, \mathsf{prepst}, \mathsf{execst}, \mathsf{postpst}\}$. The memory elements isf, jdf, jpf and plsr are used to control the pipelined processing of PGA instructions and to produce a reply in rr at the completion of each step of the pipelined instruction processing. Henceforth, we write M'_{plc} for $\{\mathsf{isf}, \mathsf{jdf}, \mathsf{jpf}, \mathsf{plsr}, \mathsf{rr}\}$.

Let $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ be a Maurer machine such that $M \cap \mathsf{M}_{\mathsf{prog}} = \emptyset$, $\mathsf{pcbr}, \mathsf{pc}, \mathsf{ir}, \mathsf{ditr}, \mathsf{bar}, \mathsf{dr}, \mathsf{eitr}, \mathsf{irr} \notin M$, $M \cap M'_{\mathrm{rr}} = \emptyset$, $M \cap M'_{\mathrm{plc}} = \emptyset$ and $\mathsf{step}, \mathsf{plctr}, \mathsf{halt} \notin A$, and let $(O_a, m_a) = [\![a]\!]$ for all $a \in A$. Then the *SP-PL-enhancement* of H is the Maurer machine $H' = (M', B', \mathcal{S}', \mathcal{O}', A', [\![\_]\!]')$ such that

$$M' = M \cup \mathsf{M}_{\mathsf{prog}} \cup \{\mathsf{pcbr}, \mathsf{pc}, \mathsf{ir}, \mathsf{ditr}, \mathsf{bar}, \mathsf{dr}, \mathsf{eitr}, \mathsf{irr}\} \cup M'_{\mathrm{rr}} \cup M'_{\mathrm{plc}} \,,$$
$$B' = B \cup \mathsf{MA}'_{\mathsf{prog}} \cup \mathfrak{I}_{\mathsf{prog}} \cup IT \cup A \cup \mathbb{B} \cup \mathcal{P}(PS) \,,$$

$$
\begin{aligned}
\mathcal{S}' &= \{S' : M' \to B' \mid \\
&\qquad S' \restriction M \in \mathcal{S} \wedge S' \restriction \mathsf{M}_{\mathsf{prog}} \in \mathsf{S}_{\mathsf{prog}} \wedge S'(\mathsf{pcbr}) \in \mathsf{MA}_{\mathsf{prog}} \wedge \\
&\qquad S'(\mathsf{pc}) \in \mathsf{MA}'_{\mathsf{prog}} \wedge S'(\mathsf{ir}) \in \mathfrak{I}_{\mathsf{prog}} \wedge \\
&\qquad S'(\mathsf{ditr}) \in IT \wedge S'(\mathsf{bar}) \in A \wedge S'(\mathsf{dr}) \in \mathsf{MA}_{\mathsf{prog}} \wedge \\
&\qquad S'(\mathsf{eitr}) \in IT \wedge S'(\mathsf{irr}) \in \mathbb{B} \wedge \\
&\qquad S'(\mathsf{rr}_{\mathsf{fetch}}) \in \mathbb{B} \wedge S'(\mathsf{rr}_{\mathsf{prep}}) \in \mathbb{B} \wedge S'(\mathsf{rr}_{\mathsf{exec}}) \in \mathbb{B} \wedge S'(\mathsf{rr}_{\mathsf{postp}}) \in \mathbb{B} \wedge \\
&\qquad S'(\mathsf{jdf}) \in \mathbb{B} \wedge S'(\mathsf{isf}) \in \mathbb{B} \wedge S'(\mathsf{jpf}) \in \mathbb{B} \wedge S'(\mathsf{plsr}) \in \mathcal{P}(PS) \wedge \\
&\qquad S'(\mathsf{rr}) \in \mathbb{B}\} \;, \\
\mathcal{O}' &= \{O' : \mathcal{S}' \to \mathcal{S}' \mid \\
&\qquad \exists O \in \mathcal{O} \bullet \forall S' \in \mathcal{S}' \bullet \\
&\qquad\quad (O'(S') \restriction M = O(S' \restriction M) \wedge O'(S') \restriction (M' \setminus M) = S' \restriction (M' \setminus M))\} \\
&\qquad \cup \{O_{\mathsf{step}}, O_{\mathsf{plctr}}, O_{\mathsf{halt}}\} \;, \\
A' &= \{\mathsf{step}, \mathsf{plctr}, \mathsf{halt}\} \;, \\
[\![a]\!]' &= (O_a, \mathsf{rr}) \quad \text{for all } a \in A' \;.
\end{aligned}
$$

O_{step} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
O_{\mathsf{step}}(S') = O'_{\mathsf{fetch}}(O'_{\mathsf{prep}}(O'_{\mathsf{exec}}(O'_{\mathsf{postp}}(S')))) \;,
$$

where O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} are suboperations defined as follows:

O'_{fetch} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
O'_{\mathsf{fetch}}(S') &= S' && \text{if } \mathsf{fetchst} \notin S'(\mathsf{plsr}) \;, \\
O'_{\mathsf{fetch}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O_{\mathsf{fetch}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{fetchst} \in S'(\mathsf{plsr}) \;, \\
O'_{\mathsf{fetch}}(S') \restriction M'_{\mathrm{plc}} &= S' \restriction M'_{\mathrm{plc}} && \text{if } \mathsf{fetchst} \in S'(\mathsf{plsr}) \;;
\end{aligned}
$$

O'_{prep} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
O'_{\mathsf{prep}}(S') &= S' && \text{if } \mathsf{prepst} \notin S'(\mathsf{plsr}) \;, \\
O'_{\mathsf{prep}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O_{\mathsf{prep}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{prepst} \in S'(\mathsf{plsr}) \;, \\
O'_{\mathsf{prep}}(S')(\mathsf{jdf}) &= jdc(S') && \text{if } \mathsf{prepst} \in S'(\mathsf{plsr}) \;, \\
O'_{\mathsf{prep}}(S') \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{jdf}\}) &= S' \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{jdf}\}) && \text{if } \mathsf{prepst} \in S'(\mathsf{plsr}) \;,
\end{aligned}
$$

where $jdc : \mathcal{S}' \to \mathbb{B}$ is the unique function from $\mathcal{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathcal{S}'$:

$$
jdc(S') = \mathsf{T} \text{ iff } O_{\mathsf{prep}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{ditr}) \in \{\mathsf{fjmp}, \mathsf{term}\} \;;
$$

O'_{exec} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
O'_{\mathsf{exec}}(S') &= S' && \text{if } \mathsf{execst} \notin S'(\mathsf{plsr}) \;, \\
O'_{\mathsf{exec}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{execst} \in S'(\mathsf{plsr}) \;, \\
O'_{\mathsf{exec}}(S')(\mathsf{isf}) &= isc(S') && \text{if } \mathsf{execst} \in S'(\mathsf{plsr}) \;, \\
O'_{\mathsf{exec}}(S') \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{isf}\}) &= S' \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{isf}\}) && \text{if } \mathsf{execst} \in S'(\mathsf{plsr}) \;,
\end{aligned}
$$

where $isc : \mathcal{S}' \to \mathbb{B}$ is the unique function from $\mathcal{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathcal{S}'$:

$$\begin{aligned}
&isc(S') = \mathsf{T} \text{ iff} \\
&\quad S'(\mathsf{ditr}) = \mathsf{ptst} \wedge O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{irr}) = \mathsf{F} \vee \\
&\quad S'(\mathsf{ditr}) = \mathsf{ntst} \wedge O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{irr}) = \mathsf{T} \ ;
\end{aligned}$$

O'_{postp} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$\begin{aligned}
O'_{\mathsf{postp}}(S') &= S' && \text{if } \mathsf{postpst} \notin S'(\mathsf{plsr}) \ , \\
O'_{\mathsf{postp}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O''_{\mathsf{postp}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{postpst} \in S'(\mathsf{plsr}) \ , \\
O'_{\mathsf{postp}}(S')(\mathsf{jpf}) &= jpc(S') && \text{if } \mathsf{postpst} \in S'(\mathsf{plsr}) \ , \\
O'_{\mathsf{postp}}(S') \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{jpf}\}) &= S' \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{jpf}\}) && \text{if } \mathsf{postpst} \in S'(\mathsf{plsr}) \ ,
\end{aligned}$$

where $jpc : \mathcal{S}' \to \mathbb{B}$ is the unique function from $\mathcal{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathcal{S}'$:

$$jpc(S') = \mathsf{T} \text{ iff } S'(\mathsf{eitr}) = \mathsf{fjmp} \ ,$$

and O''_{postp} is defined as O_{postp} in the case of the SP-NPL-enhancement, except for the replacement of the auxiliary program counter update function pcu by the function pcu' defined as follows:

$$\begin{aligned}
pcu'(S') &= S'(\mathsf{pc}) && \text{if } S'(\mathsf{eitr}) \neq \mathsf{fjmp} \ , \\
pcu'(S') &= S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) && \text{if } S'(\mathsf{eitr}) = \mathsf{fjmp} \wedge S'(\mathsf{dr}) \neq 0 \wedge \\
&&& \quad S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) \leq S'(\mathsf{pcbr}) \ , \\
pcu'(S') &= S'(\mathsf{pcbr}) + 1 && \text{if } S'(\mathsf{eitr}) = \mathsf{fjmp} \wedge \\
&&& \quad (S'(\mathsf{dr}) = 0 \vee \\
&&& \quad\ S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) > S'(\mathsf{pcbr})) \ .
\end{aligned}$$

O_{plctr} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$\begin{aligned}
O_{\mathsf{plctr}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= S' \restriction (M' \setminus M'_{\mathrm{plc}}) \ , \\
O_{\mathsf{plctr}}(S')(\mathsf{jdf}) &= \mathsf{F} \ , \\
O_{\mathsf{plctr}}(S')(\mathsf{isf}) &= \mathsf{F} \ , \\
O_{\mathsf{plctr}}(S')(\mathsf{jpf}) &= \mathsf{F} \ , \\
O_{\mathsf{plctr}}(S')(\mathsf{plsr}) &= plsu(S') \ , \\
O_{\mathsf{plctr}}(S')(\mathsf{rr}) &= ru(S') \ ,
\end{aligned}$$

where $plsu : \mathcal{S}' \to \mathcal{P}(PS)$ is the unique function from $\mathcal{S}'$ to $\mathcal{P}(PS)$ such that for all $S' \in \mathcal{S}'$:

$$\begin{aligned}
\mathsf{fetchst} \in plsu(S') \ \text{ iff } \ & S'(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \wedge \\
& (\mathsf{fetchst} \in S'(\mathsf{plsr}) \wedge S'(\mathsf{jdf}) = \mathsf{F} \vee \\
& \ S'(\mathsf{isf}) = \mathsf{T} \vee S'(\mathsf{jpf}) = \mathsf{T}) \ ,
\end{aligned}$$

$$
\begin{aligned}
&\mathsf{prepst} \in \mathit{plsu}(S') \quad \text{iff} \quad S'(\mathsf{rr_{fetch}}) = \mathsf{T} \wedge \\
&\qquad\qquad\qquad\qquad (\mathsf{fetchst} \in S'(\mathsf{plsr}) \wedge S'(\mathsf{jdf}) = \mathsf{F} \vee \\
&\qquad\qquad\qquad\qquad S'(\mathsf{isf}) = \mathsf{T})\ , \\
&\mathsf{execst} \in \mathit{plsu}(S') \quad \text{iff} \quad \mathsf{prepst} \in S'(\mathsf{plsr}) \wedge S'(\mathsf{isf}) = \mathsf{F}\ , \\
&\mathsf{postpst} \in \mathit{plsu}(S') \quad \text{iff} \quad \mathsf{execst} \in S'(\mathsf{plsr})\ ,
\end{aligned}
$$

and $ru : \mathcal{S}' \to \mathbb{B}$ is the unique function from $\mathcal{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathcal{S}'$:

$$
ru(S') = \mathsf{T} \ \text{ iff } \ \mathit{plsu}(S') \neq \emptyset \wedge S'(\mathsf{rr_{postp}}) = \mathsf{T}\ .
$$

O_{halt} is the unique function from $\mathcal{S}'$ to $\mathcal{S}'$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
&O_{\mathsf{halt}}(S') \restriction (M' \setminus \{\mathsf{rr}\}) = S' \restriction (M' \setminus \{\mathsf{rr}\})\ , \\
&O_{\mathsf{halt}}(S')(\mathsf{rr}) \quad = \mathsf{T} \quad \text{if } S'(\mathsf{rr_{postp}}) = \mathsf{F}\ , \\
&O_{\mathsf{halt}}(S')(\mathsf{rr}) \quad = \mathsf{F} \quad \text{if } S'(\mathsf{rr_{postp}}) = \mathsf{T}\ .
\end{aligned}
$$

Figure 8 shows the structure of an SP-PL-enhancement. The suboperations O'_{fetch}, O'_{prep} and O'_{exec} of O_{step} either do not affect the memory elements of $M' \setminus M'_{\mathrm{plc}}$ or do affect the memory elements of $M' \setminus M'_{\mathrm{plc}}$ exactly in the way in

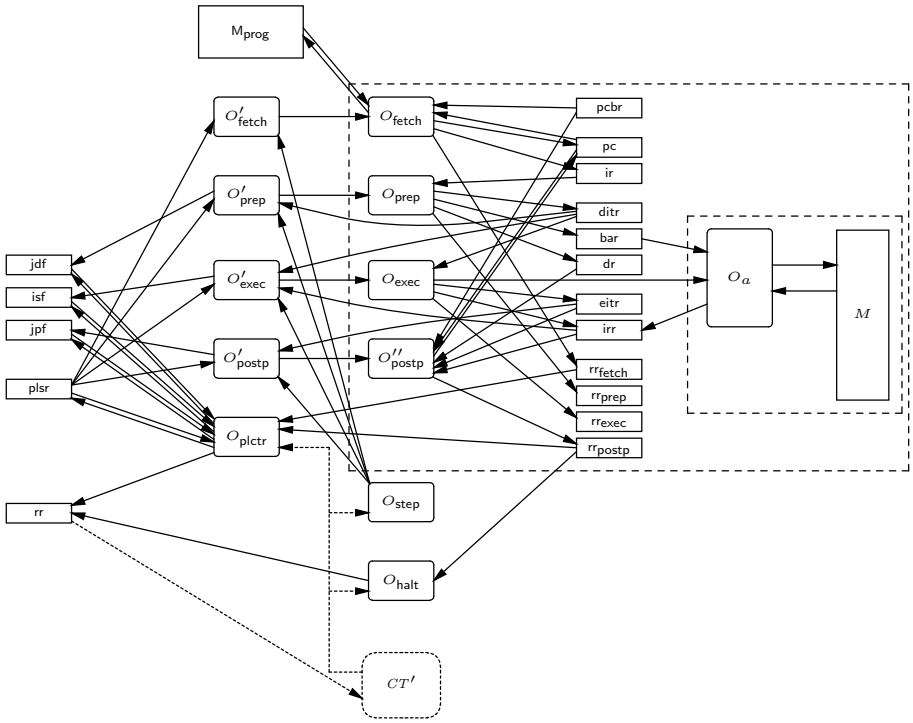

Fig. 2. Structure of an SP-PL-enhancement

which the operations O_{fetch}, O_{prep} and O_{exec} of the SP-NPL-enhancement of H would affect them. The suboperation O'_{postp} of O_{step} either does not affect the memory elements of $M' \setminus M'_{\mathrm{plc}}$ or does affect the memory elements of $M' \setminus M'_{\mathrm{plc}}$ in a way that is similar to the way in which the operation O_{postp} of the SP-NPL-enhancement of H would affect them. The difference with O_{postp} is due to the different way in which skipping of a PGA instruction is accomplished in pipelined instruction processing.

The suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} of O_{step} correspond to the pipeline stages that a PGA instruction being processed passes through successively. When the suboperation corresponding to a stage other than the last one has handled a PGA instruction, the suboperation corresponding to the next stage is enabled to handle that PGA instruction in the next step, subject to the exceptions mentioned below. O'_{fetch}, the suboperation corresponding to the first stage, is always enabled to fetch a PGA instruction in the next step, subject to the exceptions mentioned below. The exceptions are the following:

- when O'_{prep} has decoded a jump or termination instruction, pipelined instruction processing is stalled beginning with the PGA instruction fetched in the same step;
- when O'_{exec} has executed either a positive test instruction with a negative reply as result or a negative test instruction with a positive reply as result, the PGA instruction fetched immediately after the test instruction is further discarded and pipelined instruction processing is started again with the next step if the latter instruction is a jump or termination instruction;
- when O'_{postp} has adjusted the program counter on a jump instruction, the last fetched PGA instruction is discarded and pipelined instruction processing is started again with the next step.

Thus, the suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} are not all enabled to handle a PGA instruction in every step of the pipelined instruction processing. The contents of the pipeline status register indicates which of the suboperations are enabled. Enabledness is controlled by the pipeline control operation O_{plctr}. This operation is intended to be performed immediately after O_{step}. It takes output of the suboperations of O_{step} to fix up the enabledness of these suboperations for the next step.

The idea is that in each step the suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} are performed in parallel. To justify the use of the term pipeline here, we have to show that the suboperations can actually be performed in parallel. We come back on this issue in Section 9.

Consider the guarded recursive specification over BTA that consists of the following equation:

$$CT' = \mathsf{step} \circ (CT' \trianglelefteq \mathsf{plctr} \trianglerighteq (\mathsf{S} \trianglelefteq \mathsf{halt} \trianglerighteq \mathsf{D})) \,.$$

Let P be a finite PGA program. Then applying thread $|P|$ to a state of Maurer machine H has the same effect as applying power thread CT' to the corresponding state of the SP-PL-enhancement of H in which the program memory contains the stored representation of P.

Theorem 2 (SP-PL-enhancement). *Let* $H' = (M', B', \mathcal{S}', \mathcal{O}', A', [\![\_]\!]')$ *be the SP-PL-enhancement of* $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$*, let* $P = u_1; \ldots; u_n \in \mathcal{P}_{\mathsf{fin}}(A)$ *be such that* $n \leq size(\mathsf{M_{prog}})$*, let* $S'_0 \in \mathcal{S}'$ *be such that* $S'_0 \restriction \mathsf{M_{prog}}[0, n-1] = \mathsf{s_{prog}}(P)$*,* $S'_0(\mathsf{pcbr}) = n-1$*,* $S'_0(\mathsf{pc}) = 0$*,* $S'_0(\mathsf{rr_{fetch}}) = \mathsf{T}$*,* $S'_0(\mathsf{jdf}) = S'_0(\mathsf{isf}) = S'_0(\mathsf{jpf}) = \mathsf{F}$ *and* $S'_0(\mathsf{plsr}) = \{\mathsf{fetchst}\}$*. Then* $|P| \bullet_H (S'_0 \restriction M) = (CT' \bullet_{H'} S'_0) \restriction M$*.*

Proof. We prove that $(CT \bullet_{H''} (S'_0 \restriction (M' \setminus M'_{\mathrm{plc}}))) \restriction M = (CT' \bullet_{H'} S'_0) \restriction M$, where H'' is the SP-NPL-enhancement of H. From this and Theorem 1, the theorem follows immediately.

We use the following notation in the proof. For each $S' \in \mathcal{S}'$ and each $n > 0$, $cycle^n(S')$ is defined by induction on n as follows: $cycle^1(S') = O_{\mathsf{plctr}}(O_{\mathsf{step}}(S'))$ and $cycle^{n+1}(S') = O_{\mathsf{plctr}}(O_{\mathsf{step}}(cycle^n(S')))$. For each $S' \in \mathcal{S}'$, $tip(S')$ is defined as follows: $tip(S') \Leftrightarrow \mathsf{fetchst} \in S'(\mathsf{plsr}) \wedge \mathsf{prepst} \in cycle^1(S')(\mathsf{plsr}) \wedge \mathsf{execst} \in cycle^2(S')(\mathsf{plsr}) \wedge \mathsf{postpst} \in cycle^3(S')(\mathsf{plsr})$. Thus, $tip(S')$ indicates that some instruction will be totally processed from state S'.

Analysis of input and output regions yields three potential sources of interference between the suboperations of O_{step}: $OR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{fetch}}) = \{\mathsf{pc}\}$, $OR(O'_{\mathsf{postp}}) \cap IR(O'_{\mathsf{fetch}}) = \{\mathsf{pc}\}$ and $IR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{fetch}}) = \{\mathsf{pc}\}$. It is easy to see that, by stalling pipelined instruction processing when O'_{prep} has decoded a jump instruction, interference does not really happen: O'_{fetch} does not change any memory element if O'_{postp} has changed pc in the same step, and O'_{postp} does not change any memory element if O'_{fetch} has changed pc in the previous step. Because of this, it is not difficult to see that for all $S' \in \mathcal{S}'$:

$$\begin{array}{l} tip(S') \Rightarrow \\ \quad cycle^4(S') \restriction M = O_{\mathsf{postp}}(O_{\mathsf{exec}}(O_{\mathsf{prep}}(O_{\mathsf{fetch}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))))) \restriction M\ . \end{array} \tag{5}$$

We have that $tip(S'_0)$ holds. Moreover, tip is preserved by the total processing of an instruction if there is a next instruction to be processed:

- if $S'(\mathsf{M_{prog}}[S'(\mathsf{pc})]) = a$ and $S'(\mathsf{pc}) + 1 \leq S'(\mathsf{pcbr})$, then $tip(S') \Rightarrow tip(cycle^1(S'))$;
- if $S'(\mathsf{M_{prog}}[S'(\mathsf{pc})]) \in \{+a, -a\}$, $cycle^3(S')(\mathsf{isf}) = \mathsf{F}$ and $S'(\mathsf{pc})+1 \leq S'(\mathsf{pcbr})$, then $tip(S') \Rightarrow tip(cycle^1(S'))$;
- if $S'(\mathsf{M_{prog}}[S'(\mathsf{pc})]) \in \{+a, -a\}$, $cycle^3(S')(\mathsf{isf}) = \mathsf{T}$ and $S'(\mathsf{pc})+2 \leq S'(\mathsf{pcbr})$, then $tip(S') \Rightarrow tip(cycle^2(S'))$;
- if $S'(\mathsf{M_{prog}}[S'(\mathsf{pc})]) = \#k$ and $S'(\mathsf{pc}) + k \leq S'(\mathsf{pcbr})$, then $tip(S') \Rightarrow tip(cycle^4(S'))$.

Let (p_i, S_i) be the i+1-th element in the full path of $CT \bullet_{H''} (S'_0 \restriction (M' \setminus M'_{\mathrm{plc}}))$. Then it is easy to prove by induction on i that

$$\begin{array}{ll} p_{4i+4} = CT & \text{if } S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{T} \wedge S'_{4i+4}(\mathsf{rr_{postp}}) = \mathsf{T}\ , \\ p_{4i+4} = \mathsf{S} & \text{if } S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{T} \wedge S'_{4i+4}(\mathsf{rr_{postp}}) = \mathsf{F}\ , \\ p_{4i+1} = \mathsf{D} & \text{if } S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{F}\ . \end{array} \tag{6}$$

Let (p'_i, S'_i) be the i+1-th element in the full path of $CT' \bullet_{H'} S'_0$. Then it is easy to prove by induction on i that

$$\begin{array}{ll} p'_{4i+4} = CT' & \text{if } tip(S_{4i}) \wedge S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{T} \wedge S'_{4i+4}(\mathsf{rr_{postp}}) = \mathsf{T}\ , \\ p'_{4i+4} = \mathsf{S} & \text{if } tip(S_{4i}) \wedge S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{T} \wedge S'_{4i+4}(\mathsf{rr_{postp}}) = \mathsf{F}\ , \\ p'_{4i+1} = \mathsf{D} & \text{if } tip(S_{4i}) \wedge S'_{4i+1}(\mathsf{rr_{fetch}}) = \mathsf{F}\ . \end{array} \tag{7}$$

Table 11. Pipelined instruction processing of $a\,;+b\,;\#3\,;c\,;\#2\,;d\,;!$

	1	2	3	4	5	6	7	8	9	10	11	12
a	fetch	prep	exec	postp								
$+b$		fetch	prep	exec	postp							
$\#3$			fetch	prep								
c				fetch	prep	exec	postp					
$\#2$					fetch	prep	exec	postp				
d						fetch						
!									fetch	prep	exec	postp
										fetch		

Let (p_i, S_i) be the $i{+}1$-th element in the full path of $CT \bullet_{H''} (S'_0 \restriction (M' \setminus M'_{\mathrm{plc}}))$ of which the first component equals CT, S or D, and let (p'_i, S'_i) be the $i{+}1$-th element in the full path of $CT' \bullet_{H'} S'_0$ of which the first component equals CT', S or D and the second component, say S', satisfies $tip(S')$ if the first component equals CT'. Then, using (5), (6), (7) and the preservation properties of tip, it is straightforward to prove by induction on i and case distinction on the kinds of primitive instructions of PGA:

- $(p_i = CT \Leftrightarrow p'_i = CT') \wedge (p_i = \mathsf{S} \Leftrightarrow p'_i = \mathsf{S}) \wedge (p_i = \mathsf{D} \Leftrightarrow p'_i = \mathsf{D})$;
- $S_i \restriction (M' \setminus M'_{\mathrm{plc}}) = S'_i \restriction (M' \setminus M'_{\mathrm{plc}})$.

From this, the theorem follows immediately. □

Example (Pipelined instruction processing). Table 11 shows the pipelined instruction processing of the PGA program $a\,;+b\,;\#3\,;c\,;\#2\,;d\,;!$. It is assumed that the execution of $+b$ results in a negative reply. We see that the pipelined instruction processing of this PGA program is stalled three times: after the jump instruction #3 has been decoded in step 4, after the jump instruction #2 has been decoded in step 6 and after the termination instruction ! has been decoded in step 10. Because the execution of the positive test instruction $+b$ has produced a negative reply in step 4, the next instruction in the pipeline, i.e. the jump instruction #3, is not executed and post-processed in later steps. Pipelined instruction processing is started again from step 5, because there is no longer a jump instruction in the pipeline. The jump instruction #2 passes all four pipeline stages before pipelined instruction processing is started again from step 9. Moreover, because the jump is actually taken, the prematurely fetched instruction d is discarded when pipelined instruction processing is started again. The attempt to fetch another instruction prematurely in step 10 does not succeed because the last instruction of the PGA program was fetched in step 9. Instruction processing stops after step 12, because in that step the termination instruction was recognized.

Table 12 shows the pipelined instruction processing of the program $a\,;+b\,;c\,;\#3\,;d\,;e$. It is assumed that the execution of $+b$ results in a negative reply.

Table 12. Pipelined instruction processing of $a\,;+b\,;c\,;\#3\,;d\,;e$

	1	2	3	4	5	6	7	8
a	fetch	prep	exec	postp				
$+b$		fetch	prep	exec	postp			
c			fetch	prep				
$\#3$				fetch	prep	exec	postp	
d					fetch			
e								
								fetch

We see that the pipelined instruction processing of this PGA program is stalled once: after the jump instruction #3 has been decoded in step 5. Because the execution of the positive test instruction $+b$ has produced a negative reply in step 4, the next instruction in the pipeline, i.e. the void basic instruction c, is not executed and post-processed in later steps. The jump instruction #3 passes all four pipeline stages before pipelined instruction processing is started again from step 8. Moreover, because the jump is actually taken, the prematurely fetched instruction d is discarded when pipelined instruction processing is started again. The attempt to fetch another instruction in step 8 does not succeed because the jump instruction #3 has brought the program counter beyond the last instruction of the PGA program. Instruction processing stops after step 8, because in that step fetching fails while there is no other instruction in the pipeline. This situation corresponds to a programming error, such as a jump out of the program, as a result of which further instruction processing is blocked.

With pipelined instruction processing, execution of the first example program takes 12 steps and execution of the second example program takes 8 steps. With non-pipelined instruction processing, it would take 20 steps and 13 steps, respectively. However, there will be no real gain unless O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} can be performed in parallel.

9 Parallel Composability

In this section, we justify the use of the term pipeline in Section 8 by showing that the suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} of O_{step} can actually be performed in parallel.

Let $(M, B, \mathcal{S}, \mathcal{O})$ be a Maurer computer, let $O \in \mathcal{O}$, and let $O_1, O_2 : \mathcal{S} \to \mathcal{S}$ be such that $O_2(O_1(S)) = O(S)$ for all $S \in \mathcal{S}$. Then O is parallel composable of O_1 and O_2 if the following conditions are fulfilled:

- O_1 is *consistent* with O_2: if O_1 and O_2 affect the same memory element, then they affect that memory element the same;
- O_1 is *transparent* to O_2: if O_1 affects a memory element, then that memory element does not affect any memory element under O_2.

More precisely, O is *parallel composable* of O_1 and O_2 iff $O_1 \mathsf{\ con\ } O_2 \wedge O_1 \mathsf{\ tra\ } O_2$, where con and tra are defined as follows:

$$
\begin{aligned}
&O_1 \mathsf{\ con\ } O_2 \mathsf{\ iff}\\
&\quad \forall m \in OR(O_1) \cap OR(O_2), S \in \mathcal{S} \bullet\\
&\qquad (O_1(S)(m) \neq S(m) \wedge O_2(S)(m) \neq S(m) \Rightarrow O_1(S)(m) = O_2(S)(m)) \ ,
\end{aligned}
$$

$$
\begin{aligned}
&O_1 \mathsf{\ tra\ } O_2 \mathsf{\ iff}\\
&\quad \forall m \in OR(O_1) \cap IR(O_2), S \in \mathcal{S} \bullet\\
&\qquad (O_1(S)(m) \neq S(m) \Rightarrow\\
&\qquad\quad \neg\, (\exists S' \in \mathcal{S} \bullet (\forall m' \in M \setminus \{m\} \bullet O_1(S)(m') = S'(m') \wedge\\
&\qquad\qquad\qquad \exists m'' \in OR(O_2) \bullet O_2(O_1(S))(m'') \neq O_2(S')(m'')))) \ .
\end{aligned}
$$

Sufficient conditions for $O_1 \mathsf{\ con\ } O_2$ and $O_1 \mathsf{\ tra\ } O_2$ to hold are $OR(O_1) \cap OR(O_2) = \emptyset$ and $OR(O_1) \cap IR(O_2) = \emptyset$, respectively.

Parallel composability generalizes easily to n operations (for $n \geq 2$).

Let $(M, B, \mathcal{S}, \mathcal{O})$ be a Maurer computer, let $O \in \mathcal{O}$, and let $O_1, \ldots, O_n : \mathcal{S} \to \mathcal{S}$ be such that $O_n(\ldots O_1(S) \ldots) = O(S)$ for all $S \in \mathcal{S}$. Then O is parallel composable of $O_1, \ldots, O_n$ iff $\bigwedge_{1 \leq i < n} \bigwedge_{i < j \leq n} (O_i \mathsf{\ con\ } O_j \wedge O_i \mathsf{\ tra\ } O_j)$.

The suboperations O'_{fetch}, O'_{prep}, O'_{exec} and O'_{postp} of O_{step} from Section 8 can be performed in parallel.

Theorem 3 (Parallel composability). *Take the SP-PL-enhancement of a Maurer machine H as in Section 8. Then O_{step} is parallel composable of O'_{postp}, O'_{exec}, O'_{prep} and O'_{fetch}.*

Proof. The following follows immediately from the definitions:

$$
\begin{aligned}
&OR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{exec}}) = \emptyset, && OR(O'_{\mathsf{postp}}) \cap IR(O'_{\mathsf{exec}}) = \emptyset,\\
&OR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{prep}}) = \emptyset, && OR(O'_{\mathsf{postp}}) \cap IR(O'_{\mathsf{prep}}) = \emptyset,\\
&OR(O'_{\mathsf{postp}}) \cap OR(O'_{\mathsf{fetch}}) = \{\mathsf{pc}\}, && OR(O'_{\mathsf{postp}}) \cap IR(O'_{\mathsf{fetch}}) = \{\mathsf{pc}\},\\
&OR(O'_{\mathsf{exec}}) \cap OR(O'_{\mathsf{prep}}) = \emptyset, && OR(O'_{\mathsf{exec}}) \cap IR(O'_{\mathsf{prep}}) = \emptyset,\\
&OR(O'_{\mathsf{exec}}) \cap OR(O'_{\mathsf{fetch}}) = \emptyset, && OR(O'_{\mathsf{exec}}) \cap IR(O'_{\mathsf{fetch}}) = \emptyset,\\
&OR(O'_{\mathsf{prep}}) \cap OR(O'_{\mathsf{fetch}}) = \emptyset, && OR(O'_{\mathsf{prep}}) \cap IR(O'_{\mathsf{fetch}}) = \emptyset \ .
\end{aligned}
$$

Hence, we need to have a closer look only on the conditions $O'_{\mathsf{postp}} \mathsf{\ con\ } O'_{\mathsf{fetch}}$ and $O'_{\mathsf{postp}} \mathsf{\ tra\ } O'_{\mathsf{fetch}}$; and we have to consider only the memory element pc. Now, take an arbitrary state S'. It is easy to see that, if O'_{postp} changes pc in state S', then O'_{exec} must not have set isf one step back and O'_{prep} must have set jdf two steps back. It is also easy to see that, as a consequence, O'_{fetch} does not change any memory element in states S' and $O'_{\mathsf{prep}}(S')$. Hence, both the consistency condition and the transparency condition are trivially met. □

The proof of Theorem 3 shows that stalling pipelined instruction processing when O'_{prep} has decoded a jump instruction is crucial for parallel composability. It is easy to see that O_{step} is not parallel composable of O'_{postp}, O'_{exec}, O'_{prep}, O'_{fetch} and O_{plctr}. This is to be expected. For example, the flags jdf, isf and jpf are set by O'_{prep}, O'_{exec} and O'_{postp} to influence how plsr is updated by O_{plctr}.

10 Conditional Jump Instructions

In this section, we extend PGA with conditional jump instructions and look at the effect of this on non-pipelined and pipelined instruction processing.

We add to PGA the following primitive instructions:

- for each $a \in \mathfrak{A}$ and $k \in \mathbb{N}$, a *positive conditional jump instruction* $+a\#k$;
- for each $a \in \mathfrak{A}$ and $k \in \mathbb{N}$, a *negative conditional jump instruction* $-a\#k$.

A positive conditional jump instruction $+a\#k$ has the same effect as $+a \,;\, \#k$, but counts for one instruction; and a negative conditional jump instruction $-a\#k$ has the same effect as $-a \,;\, \#k$, but counts for one instruction. In [3], PGA is extended with a *unit instruction* operator $\mathbf{u}$ which turns PGA programs into single instructions. In that extension of PGA, called $\mathrm{PGA_u}$, $+a\#k$ and $-a\#k$ can be taken as abbreviations for $\mathbf{u}(+a;\#k)$ and $\mathbf{u}(-a;\#k)$, respectively. In [18], thread extraction for $\mathrm{PGA_u}$ programs is described by means of a mapping from $\mathrm{PGA_u}$ programs to PGA programs.

The SP-NPL-enhancement of Maurer machines change only slightly when conditional jump instructions are added. Just the set IT and the auxiliary functions *dec*, *opc* and *pcu* used in the definition of the SP-NPL-enhancement of a Maurer machine from Section 7 have to be re-defined. The set IT is re-defined because the two kinds of conditional jump instructions give rise to two additional instruction types: pcfjmp and ncfjmp. The function *dec* is re-defined in order to deal with the decoding of conditional jump instructions. The function *opc* is re-defined because conditional jump instructions cause an operation to be performed. The function *pcu* is re-defined in order to deal with the adjustment of the program counter in the case of conditional jump instructions.

IT is re-defined to be the set $\{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}, \mathsf{fjmp}, \mathsf{pcfjmp}, \mathsf{ncfjmp}, \mathsf{term}\}$.

The function $dec : \mathcal{S}' \to IT \times A \times \mathsf{MA}_{\mathsf{prog}}$ is re-defined as follows:

$$
\begin{aligned}
dec(S') &= (\mathsf{bsc}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = a \;, \\
dec(S') &= (\mathsf{ptst}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = +a \;, \\
dec(S') &= (\mathsf{ntst}, a, S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = -a \;, \\
dec(S') &= (\mathsf{fjmp}, S'(\mathsf{bar}), k) && \text{if } S'(\mathsf{ir}) = \#k \;, \\
dec(S') &= (\mathsf{pcfjmp}, a, k) && \text{if } S'(\mathsf{ir}) = +a\#k \;, \\
dec(S') &= (\mathsf{ncfjmp}, a, k) && \text{if } S'(\mathsf{ir}) = -a\#k \;, \\
dec(S') &= (\mathsf{term}, S'(\mathsf{bar}), S'(\mathsf{dr})) && \text{if } S'(\mathsf{ir}) = ! \;.
\end{aligned}
$$

The function $opc : \mathcal{S}' \to \mathbb{B}$ is re-defined as follows:

$$opc(S') = \mathsf{T} \text{ iff } S'(\mathsf{ditr}) \in \{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}, \mathsf{pcfjmp}, \mathsf{ncfjmp}\} \;.$$

The function $pcu : \mathcal{S}' \to \mathsf{MA}'_{\mathsf{prog}}$ is re-defined as follows:

$$
\begin{array}{ll}
pcu(S') = S'(\mathsf{pc}) & \text{if } S'(\mathsf{eitr}) = \mathsf{bsc} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{ptst} \wedge S'(\mathsf{irr}) = \mathsf{T} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{ntst} \wedge S'(\mathsf{irr}) = \mathsf{F} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{pcfjmp} \wedge S'(\mathsf{irr}) = \mathsf{F} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \wedge S'(\mathsf{irr}) = \mathsf{T} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{term}\ , \\
pcu(S') = S'(\mathsf{pc}) + 1 & \text{if } (S'(\mathsf{eitr}) = \mathsf{ptst} \wedge S'(\mathsf{irr}) = \mathsf{F} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{ntst} \wedge S'(\mathsf{irr}) = \mathsf{T}) \wedge \\
 & \quad S'(\mathsf{pc}) + 1 \leq S'(\mathsf{pcbr})\ , \\
pcu(S') = S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) & \text{if } (S'(\mathsf{eitr}) = \mathsf{fjmp} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{pcfjmp} \wedge S'(\mathsf{irr}) = \mathsf{T} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \wedge S'(\mathsf{irr}) = \mathsf{F}) \wedge \\
 & \quad S'(\mathsf{dr}) \neq 0 \wedge \\
 & \quad S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) \leq S'(\mathsf{pcbr})\ , \\
pcu(S') = S'(\mathsf{pcbr}) + 1 & \text{if } (S'(\mathsf{eitr}) = \mathsf{ptst} \wedge S'(\mathsf{irr}) = \mathsf{F} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{ntst} \wedge S'(\mathsf{irr}) = \mathsf{T}) \wedge \\
 & \quad S'(\mathsf{pc}) + 1 > S'(\mathsf{pcbr}) \vee \\
 & \quad (S'(\mathsf{eitr}) = \mathsf{fjmp} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{pcfjmp} \wedge S'(\mathsf{irr}) = \mathsf{T} \vee \\
 & \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \wedge S'(\mathsf{irr}) = \mathsf{F}) \wedge \\
 & \quad (S'(\mathsf{dr}) = 0 \vee \\
 & \quad S'(\mathsf{pc}) - 1 + S'(\mathsf{dr}) > S'(\mathsf{pcbr}))\ .
\end{array}
$$

Like the SP-NPL-enhancement of Maurer machines, the SP-PL-enhancement of Maurer machines change only slightly when conditional jump instructions are added. The memory has to be extended with a *conditional jump flag* (cjf) which, like the other flags, contains a Boolean value. The set M'_{plc}, the auxiliary functions *jpc* and *pcu′*, the suboperation O'_{exec} and the operation O_{plctr} used in the definition of the SP-PL-enhancement of a Maurer machine from Section 8 have to be re-defined. The flag cjf is needed in order to control the pipelined processing of instructions in the presence of conditional jump instructions. The set M'_{plc} is re-defined because of the addition of the flag cjf. The function *jpc* is re-defined because, after adjustment of the program counter on conditional jump instructions, pipelined instruction processing must be restarted as in the case of unconditional jump instructions. Just like *pcu* before, the function *pcu′* is re-defined in order to deal with the adjustment of the program counter in the case of conditional jump instructions. The suboperation O'_{exec} is re-defined in order to set the additional flag cjf when, in the case of conditional jump instructions, the reply value is produced on which the jump concerned must actually take place.

The operation O_{plctr} is re-defined in order to control the pipelined processing of instructions in the presence of conditional jump instructions.

M'_{plc} is re-defined to be the set $\{\mathsf{isf}, \mathsf{jdf}, \mathsf{jpf}, \mathsf{cjf}, \mathsf{plsr}, \mathsf{rr}\}$.

The function $jpc : \mathcal{S}' \to \mathbb{B}$ is re-defined as follows:

$$
\begin{aligned}
&jpc(S') = \mathsf{T} \text{ iff} \\
&\quad S'(\mathsf{eitr}) = \mathsf{fjmp} \vee \\
&\quad S'(\mathsf{eitr}) = \mathsf{pcfjmp} \wedge S'(\mathsf{irr}) = \mathsf{T} \vee S'(\mathsf{eitr}) = \mathsf{ncfjmp} \wedge S'(\mathsf{irr}) = \mathsf{F}\ .
\end{aligned}
$$

The function $pcu' : \mathcal{S}' \to \mathsf{MA}'_{\mathsf{prog}}$ is re-defined as follows:

$$
\begin{aligned}
pcu'(S') &= S'(\mathsf{pc}) && \text{if } S'(\mathsf{eitr}) \in \{\mathsf{bsc}, \mathsf{ptst}, \mathsf{ntst}, \mathsf{term}\} \vee \\
&&& \quad S'(\mathsf{eitr}) = \mathsf{pcfjmp} \wedge S'(\mathsf{irr}) = \mathsf{F} \vee \\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \wedge S'(\mathsf{irr}) = \mathsf{T}\ , \\
pcu'(S') &= S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) && \text{if } S'(\mathsf{eitr}) = \mathsf{fjmp} \wedge S'(\mathsf{dr}) \neq 0 \wedge \\
&&& \quad S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) \leq S'(\mathsf{pcbr})\ , \\
pcu'(S') &= S'(\mathsf{pc}) - 3 + S'(\mathsf{dr}) && \text{if } (S'(\mathsf{eitr}) = \mathsf{pcfjmp} \wedge S'(\mathsf{irr}) = \mathsf{T} \vee \\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \wedge S'(\mathsf{irr}) = \mathsf{F}) \wedge \\
&&& \quad S'(\mathsf{dr}) \neq 0 \wedge \\
&&& \quad S'(\mathsf{pc}) - 3 + S'(\mathsf{dr}) \leq S'(\mathsf{pcbr})\ , \\
pcu'(S') &= S'(\mathsf{pcbr}) + 1 && \text{if } S'(\mathsf{eitr}) = \mathsf{fjmp} \wedge \\
&&& \quad (S'(\mathsf{dr}) = 0 \vee \\
&&& \quad S'(\mathsf{pc}) - 2 + S'(\mathsf{dr}) > S'(\mathsf{pcbr})) \vee \\
&&& \quad (S'(\mathsf{eitr}) = \mathsf{pcfjmp} \wedge S'(\mathsf{irr}) = \mathsf{T} \vee \\
&&& \quad S'(\mathsf{eitr}) = \mathsf{ncfjmp} \wedge S'(\mathsf{irr}) = \mathsf{F}) \wedge \\
&&& \quad (S'(\mathsf{dr}) = 0 \vee \\
&&& \quad S'(\mathsf{pc}) - 3 + S'(\mathsf{dr}) > S'(\mathsf{pcbr}))\ .
\end{aligned}
$$

The suboperation O'_{exec} is re-defined as follows:

$$
\begin{aligned}
O'_{\mathsf{exec}}(S') &= S' && \text{if } \mathsf{execst} \notin S'(\mathsf{plsr})\ , \\
O'_{\mathsf{exec}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) &= O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}})) && \text{if } \mathsf{execst} \in S'(\mathsf{plsr})\ , \\
O'_{\mathsf{exec}}(S')(\mathsf{isf}) &= isc(S') && \text{if } \mathsf{execst} \in S'(\mathsf{plsr})\ , \\
O'_{\mathsf{exec}}(S')(\mathsf{cjf}) &= cjc(S') && \text{if } \mathsf{execst} \in S'(\mathsf{plsr})\ , \\
O'_{\mathsf{exec}}(S') \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{isf}, \mathsf{cjf}\}) &= S' \restriction (M'_{\mathrm{plc}} \setminus \{\mathsf{isf}, \mathsf{cjf}\}) && \text{if } \mathsf{execst} \in S'(\mathsf{plsr})\ ,
\end{aligned}
$$

where $isc : \mathcal{S}' \to \mathbb{B}$ is defined as in the case without conditional jump instructions and $cjc : \mathcal{S}' \to \mathbb{B}$ is the unique function from $\mathcal{S}'$ to $\mathbb{B}$ such that for all $S' \in \mathcal{S}'$:

$$
\begin{aligned}
&cjc(S') = \mathsf{T} \text{ iff} \\
&\quad S'(\mathsf{ditr}) = \mathsf{pcfjmp} \wedge O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{irr}) = \mathsf{T} \vee \\
&\quad S'(\mathsf{ditr}) = \mathsf{ncfjmp} \wedge O_{\mathsf{exec}}(S' \restriction (M' \setminus M'_{\mathrm{plc}}))(\mathsf{irr}) = \mathsf{F}\ .
\end{aligned}
$$

O_{plctr} is re-defined as follows:

$$\begin{aligned}
&O_{\mathsf{plctr}}(S') \restriction (M' \setminus M'_{\mathrm{plc}}) = S' \restriction (M' \setminus M'_{\mathrm{plc}}) \;,\\
&O_{\mathsf{plctr}}(S')(\mathsf{jdf}) = \mathsf{F} \;,\\
&O_{\mathsf{plctr}}(S')(\mathsf{isf}) = \mathsf{F} \;,\\
&O_{\mathsf{plctr}}(S')(\mathsf{jpf}) = \mathsf{F} \;,\\
&O_{\mathsf{plctr}}(S')(\mathsf{cjf}) = \mathsf{F} \;,\\
&O_{\mathsf{plctr}}(S')(\mathsf{plsr}) = \mathit{plsu}(S') \;,\\
&O_{\mathsf{plctr}}(S')(\mathsf{rr}) = \mathit{ru}(S') \;,
\end{aligned}$$

where $\mathit{plsu} : \mathcal{S}' \to \mathcal{P}(PS)$ is the unique function from $\mathcal{S}'$ to $\mathcal{P}(PS)$ such that for all $S' \in \mathcal{S}'$:

$$\begin{aligned}
&\mathsf{fetchst} \in \mathit{plsu}(S') \;\; \mathsf{iff} \;\; S'(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \wedge\\
&\qquad (\mathsf{fetchst} \in S'(\mathsf{plsr}) \wedge S'(\mathsf{jdf}) = \mathsf{F} \wedge S'(\mathsf{cjf}) = \mathsf{F} \vee\\
&\qquad S'(\mathsf{isf}) = \mathsf{T} \vee S'(\mathsf{jpf}) = \mathsf{T}) \;,\\
&\mathsf{prepst} \in \mathit{plsu}(S') \;\; \mathsf{iff} \;\; S'(\mathsf{rr}_{\mathsf{fetch}}) = \mathsf{T} \wedge\\
&\qquad (\mathsf{fetchst} \in S'(\mathsf{plsr}) \wedge S'(\mathsf{jdf}) = \mathsf{F} \wedge S'(\mathsf{cjf}) = \mathsf{F} \vee\\
&\qquad S'(\mathsf{isf}) = \mathsf{T}) \;,\\
&\mathsf{execst} \in \mathit{plsu}(S') \;\; \mathsf{iff} \;\; \mathsf{prepst} \in S'(\mathsf{plsr}) \wedge S'(\mathsf{isf}) = \mathsf{F} \wedge S'(\mathsf{cjf}) = \mathsf{F} \;,\\
&\mathsf{postpst} \in \mathit{plsu}(S') \;\; \mathsf{iff} \;\; \mathsf{execst} \in S'(\mathsf{plsr}) \;.
\end{aligned}$$

11 Backward Jump Instructions

In this short section, we discuss backward jump instructions and sketch the effect of their inclusion on non-pipelined and pipelined instruction processing.

In the preceding sections, we have considered only finite PGA programs, i.e. closed terms of PGA in which the repetition operator does not occur. This means that programs that are infinite sequences of primitive instructions are excluded. In other words, programs of which the execution goes on indefinitely are not covered. However, in a setting with backward jump instructions, there exists for each such program a behaviourally equivalent program that is a finite sequence of primitive instructions.

In a setting with backward jump instructions, there are, in addition to the primitive instructions of PGA introduced earlier, the following primitive instructions:

- for each $k \in \mathbb{N}$, a *backward jump instruction* $\backslash\#k$.

We write $\mathfrak{I}'$ for the set that consists of all primitive instructions of PGA and all backward jump instructions. A PGLB *program* is a closed term that can be built from:

- for each $u \in \mathfrak{I}'$, an instruction constant u;

– the concatenation operator _ ; _ .

In [3], the meaning of PGLB programs is described by means of a mapping from PGLB programs to PGA programs. For each PGA program, there exists a PGLB program that is mapped to a PGA program with the same behaviour. In other words, the expressiveness is not decreased by replacing the repetition operator by backward jump instructions.

The addition of backward jump instructions gives rise to trivial changes of the SP-NPL-enhancement and SP-PL-enhancement of Maurer machines: forward jump instructions and backward jump instructions can be treated in the same way.

Just the set IT and the auxiliary functions dec and pcu used in the definition of the SP-NPL-enhancement of a Maurer machine from Section 7 and the auxiliary function pcu' used in the definition of the SP-PL-enhancement of a Maurer machine from Section 8 have to be re-defined. The set IT must be re-defined because the backward jump instructions give rise to an additional instruction type: bjmp. The function dec must be re-defined in order to deal with the decoding of backward jump instructions. The function pcu and pcu' must be re-defined in order to deal with the adjustment of the program counter in the case of backward jump instructions.

It is easy to see that with the correct re-definitions, Theorems 1 and 2 go through after the addition of backward jump instructions. Conditional backward jump instructions can be added in the same way as conditional forward jump instructions have been added in Section 10.

12 Instruction Set Architectures

In this section, we introduce the concept of a Maurer instruction set architecture. The concept of a Maurer instruction set architecture, or shortly a Maurer ISA, is an approximation of the concept of an instruction set architecture. It is focussed on instructions for data manipulation and data transfer. Instructions for transfer of program control are treated in a uniform way over different Maurer ISAs. Instances of the concept of a Maurer ISA are those Maurer machines for which SP-NPL-enhancement and SP-PL-enhancement are primarily intended. The SP-NPL-enhancement and SP-PL-enhancement of a Maurer ISA can be viewed as implementations of that ISA.

Each Maurer machine has a number of basic actions with which an operation is associated. In this section, when speaking about Maurer machines that are Maurer ISAs, such basic actions are loosely called basic instructions. The term basic action is uncommon where we are concerned with ISAs, and moreover basic instructions and basic actions are identified in the semantics of PGA.

The basic idea underlying the concept of a Maurer ISA is that there is a main memory of which the elements contain data, an operating unit with a small internal memory by which data can be manipulated, and an interface between the main memory and the operating unit for data transfer between them. For

the sake of simplicity, data is restricted to the natural numbers between 0 and some upper bound. Other types of data that could be supported can always be represented by the natural numbers provided. Moreover, the data manipulation instructions offered by a Maurer ISA are not restricted and may include ones that are tailored to manipulation of representations of other types of data. Therefore, we believe that nothing essential is lost by the restriction to natural numbers.

The concept of a Maurer ISA is parametrized by:

- an address width k;
- a word length l;
- a bit size m of the operating unit;
- a number u of pairs of address and data registers for load instructions;
- a number v of pairs of address and data registers for store instructions;
- a set A' of basic instructions for data manipulation.

It is assumed that a fixed but arbitrary set $\mathsf{M}_{\mathsf{data}}$ of cardinality 2^k and a fixed but arbitrary bijection $\mathsf{m}_{\mathsf{data}} : [0, 2^k - 1] \to \mathsf{M}_{\mathsf{data}}$ have been given. $\mathsf{M}_{\mathsf{data}}$ is called the *data memory*. The data memory is a memory of which the elements can be addressed by means of natural numbers in the interval $[0, 2^k - 1]$. The address width k can be regarded as the number of bits used for the binary representation of addresses of data memory elements. We write $\mathsf{B}_{\mathsf{addr}}$ for $[0, 2^k - 1]$. The data memory elements are meant for containing data. They can contain natural numbers in the interval $[0, 2^l - 1]$. The word length l can be regarded as the number of bits used to represent data in data memory elements. We write $\mathsf{B}_{\mathsf{data}}$ for $[0, 2^l - 1]$.

It is assumed that a fixed but arbitrary set M_{ou} of cardinality m, called the *operating unit memory*, has been given. The operating unit memory is a memory of which the elements can contain natural numbers in the set $\{0, 1\}$, i.e. bits. The bit size m can be regarded as the number of bits that the operating unit can store internally. We write Bit for $\{0, 1\}$.

It is assumed that fixed but arbitrary sets M_{ld} and M_{la} of cardinality u and fixed but arbitrary bijections $\mathsf{m}_{\mathsf{ld}} : [0, u - 1] \to \mathsf{M}_{\mathsf{ld}}$ and $\mathsf{m}_{\mathsf{la}} : [0, u - 1] \to \mathsf{M}_{\mathsf{la}}$ have been given. It is also assumed that fixed but arbitrary sets M_{sd} and M_{sa} of cardinality v and fixed but arbitrary bijections $\mathsf{m}_{\mathsf{sd}} : [0, v - 1] \to \mathsf{M}_{\mathsf{sd}}$ and $\mathsf{m}_{\mathsf{sa}} : [0, v - 1] \to \mathsf{M}_{\mathsf{sa}}$ have been given. The members of M_{la} and M_{ld} are called *load address registers* and *load data registers*, respectively. The members of M_{sa} and M_{sd} are called *store address registers* and *store data registers*, respectively. The load and store registers are special memory elements meant for transferring data between the data memory and the operating unit memory. The members of M_{la} and M_{sa} can contain addresses, i.e. members of $\mathsf{B}_{\mathsf{addr}}$. The members of M_{ld} and M_{sd} can contain data, i.e. members of $\mathsf{B}_{\mathsf{data}}$.

Let $n \in [0, 2^k - 1]$, $n' \in [0, u - 1]$ and $n'' \in [0, v - 1]$. Then, we write $\mathsf{M}_{\mathsf{data}}[n]$ for $\mathsf{m}_{\mathsf{data}}(n)$, $\mathsf{M}_{\mathsf{ld}}[n']$ for $\mathsf{m}_{\mathsf{ld}}(n')$, $\mathsf{M}_{\mathsf{la}}[n']$ for $\mathsf{m}_{\mathsf{la}}(n')$, $\mathsf{M}_{\mathsf{sd}}[n'']$ for $\mathsf{m}_{\mathsf{sd}}(n'')$ and $\mathsf{M}_{\mathsf{sa}}[n'']$ for $\mathsf{m}_{\mathsf{sa}}(n'')$.

A *Maurer instruction set architecture* with parameters k, l, m, u, v and A' is a Maurer machine $H = (M, B, \mathcal{S}, \mathcal{O}, A, [\![\_]\!])$ with

$$\begin{aligned}
M &= \mathsf{M}_{\mathsf{data}} \cup \mathsf{M}_{\mathsf{ou}} \cup \mathsf{M}_{\mathsf{ld}} \cup \mathsf{M}_{\mathsf{sd}} \cup \mathsf{M}_{\mathsf{la}} \cup \mathsf{M}_{\mathsf{sa}} \cup \{\mathsf{rr}_a \mid a \in A\} \;, \\
B &= \mathsf{B}_{\mathsf{data}} \cup \mathsf{B}_{\mathsf{addr}} \cup \mathbb{B} \;, \\
\mathcal{S} &= \{S : M \to B \mid \\
&\qquad \forall m \in \mathsf{M}_{\mathsf{data}} \cup \mathsf{M}_{\mathsf{ld}} \cup \mathsf{M}_{\mathsf{sd}} \bullet S(m) \in \mathsf{B}_{\mathsf{data}} \wedge \\
&\qquad \forall m \in \mathsf{M}_{\mathsf{la}} \cup \mathsf{M}_{\mathsf{sa}} \bullet S(m) \in \mathsf{B}_{\mathsf{addr}} \wedge \\
&\qquad \forall m \in \mathsf{M}_{\mathsf{ou}} \bullet S(m) \in \mathsf{Bit} \wedge \forall a \in A \bullet S(\mathsf{rr}_a) \in \mathbb{B}\} \;, \\
\mathcal{O} &= \{O_a \mid a \in A\} \;, \\
A &= \{\mathsf{load}{:}n \mid n \in [0, u-1]\} \cup \{\mathsf{store}{:}n \mid n \in [0, v-1]\} \cup A' \;, \\
[\![a]\!] &= (O_a, \mathsf{rr}_a) \quad \text{for all } a \in A \;,
\end{aligned}$$

where, for all $n \in [0, u-1]$, $O_{\mathsf{load}:n}$ is the unique function from $\mathcal{S}$ to $\mathcal{S}$ such that for all $S \in \mathcal{S}$:

$$\begin{aligned}
O_{\mathsf{load}:n}(S) \upharpoonright (M \setminus \{\mathsf{M}_{\mathsf{ld}}[n]\}) &= S \upharpoonright (M \setminus \{\mathsf{M}_{\mathsf{ld}}[n]\}) \;, \\
O_{\mathsf{load}:n}(S)(\mathsf{M}_{\mathsf{ld}}[n]) &= S(\mathsf{M}_{\mathsf{data}}[S(\mathsf{M}_{\mathsf{la}}[n])]) \;,
\end{aligned}$$

and, for all $n \in [0, v-1]$, $O_{\mathsf{store}:n}$ is the unique function from $\mathcal{S}$ to $\mathcal{S}$ such that for all $S \in \mathcal{S}$:

$$\begin{aligned}
O_{\mathsf{store}:n}(S) \upharpoonright (M \setminus \{\mathsf{M}_{\mathsf{data}}[S(\mathsf{M}_{\mathsf{sa}}[n])]\}) &= S \upharpoonright (M \setminus \{\mathsf{M}_{\mathsf{data}}[S(\mathsf{M}_{\mathsf{sa}}[n])]\}) \;, \\
O_{\mathsf{store}:n}(S)(\mathsf{M}_{\mathsf{data}}[S(\mathsf{M}_{\mathsf{sa}}[n])]) &= S(\mathsf{M}_{\mathsf{sd}}[n]) \;,
\end{aligned}$$

and, for all $a \in A'$, O_a is a function from $\mathcal{S}$ to $\mathcal{S}$ such that:

$$\begin{aligned}
IR(O_a) &\subseteq \mathsf{M}_{\mathsf{ou}} \cup \mathsf{M}_{\mathsf{ld}} \;, \\
OR(O_a) &\subseteq \mathsf{M}_{\mathsf{ou}} \cup \mathsf{M}_{\mathsf{sd}} \cup \mathsf{M}_{\mathsf{la}} \cup \mathsf{M}_{\mathsf{sa}} \;.
\end{aligned}$$

On purpose, Maurer instruction set architectures have a bias towards load/store architectures. We believe that load/store architectures give rise to the most conveniently arranged interface between the data memory and the operating unit. For example, with an architecture other than a load/store architecture, it is more difficult to establish statically, when it concerns instructions for data manipulation and/or dat transfer, the cases in which the operations associated with instructions that follow each other can be safely performed in a different order or in parallel.

13 Conclusions

We have modelled micro-architectures with non-pipelined instruction processing and pipelined instruction processing, using Maurer machines, basic thread algebra and program algebra. Because our descriptions of micro-architectures are more precise than those usually given, we have been able to verify that stored

programs are executed as intended with these micro-architectures. A thorough understanding of the issues relevant to pipelined instruction processing can be acquired by modelling micro-architectures based on different pipeline organizations as well.

In this paper, pipelined instruction processing does deal with control conflicts, but does not deal with data conflicts. Because memory access is not made explicit, data conflicts simply do not occur in the model presented in this paper. Models in which memory access is made explicit may have it placed in a separate pipeline stage, as a result of which data conflicts may occur. In those models, additional assumptions are needed about the instruction set architecture. The additional assumptions for load/store instruction set architectures are incorporated in the concept of a Maurer instruction set architecture introduced in this paper.

Several techniques for speeding up instruction processing involve multi-threading, a form of concurrency where some interleaving strategy determines how threads that exist concurrently are interleaved (see also [6]). When modelling micro-architectures for those techniques, the enabledness of basic actions discussed in Section 4 is likely to be relevant. It certainly is relevant in the case of micro-threading [8, 16].

There are many options for future work. We mention only the modelling of micro-architectures for different combinations of instruction set architecture and technique for speeding up instruction processing. By that, the work presented in this paper may grow out to a theoretical basis for micro-architecture design.

The work presented in this paper, as well as the preceding work presented in [4], has convinced us that a special notation for the description of micro-architectures is desirable. For example, it is annoying that, for each memory element that is not affected by an operation, this must be described explicitly. However, we found that fixing an appropriate notation still requires some significant design decisions. We aim at a notation of which the semantics can simply be given by a translation to logical formulas, much in the spirit of predicative methodology [12]. The following alternative description of the operation O_{fetch} from Section 7 shows how an appropriate notation could look like:

$$\begin{aligned} O_{\mathsf{fetch}} : \; & \mathbf{if}\ \mathsf{pc}+1 \leq \mathsf{pcbr}\ \mathbf{then}\ \mathsf{pc} := \mathsf{pc}+1\ ; \\ & \mathbf{if}\ \mathsf{pc} \leq \mathsf{pcbr}\ \mathbf{then}\ (\mathsf{ir} := \mathsf{M}_{\mathsf{prog}}[\mathsf{pc}]\ ;\ \mathsf{rr} := \mathsf{T})\ \mathbf{else}\ (\mathsf{ir} := \#0\ ;\ \mathsf{rr} := \mathsf{F})\ . \end{aligned}$$

The work presented in [4] and this paper has also convinced us that modularity is material to this work: it is about combining and extending models and about renaming and hiding names used in those models. All this is done informally until now, but in the future there may arise a need to formalize it. We believe that module algebra [2] is a suitable formalism to base that formalization on.

Parallel composibility in connection with pipelined instruction processing is studied in a different setting in [14]. Using algebraic techniques from [11], three simple pipelined systems and a pipelined implementation of a micro-processor are both modelled and verified in [10] and [9], respectively. The simple pipelined

systems as well as the pipelined implementation of a micro-processor are modelled as iterated maps. By modelling a pipelined micro-processor as an iterated map, it is modelled at a level of abstraction that is higher than the level of abstraction at which micro-architecture design takes place. We focus our attention on modelling at the latter level of abstraction. A very extensive and up-to-date overview of interesting work on modelling and verifying pipelined micro-processors can also be found in [10].

Acknowledgements

We thank Bob Diertens from the University of Amsterdam, Programming Research Group, for carefully reading a draft of this paper, for contributing the pictures included in this paper, and for implementing a simulator for the micro-architectures modelled in this paper.

References

1. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.
2. J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.
3. J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
4. J. A. Bergstra and C. A. Middelburg. Maurer computers with single-thread control. Computer Science Report 05-17, Department of Mathematics and Computer Science, Eindhoven University of Technology, June 2005.
5. J. A. Bergstra and C. A. Middelburg. Simulating Turing machines on Maurer machines. Computer Science Report 05-28, Department of Mathematics and Computer Science, Eindhoven University of Technology, November 2005.
6. J. A. Bergstra and C. A. Middelburg. A thread algebra with multi-level strategic interleaving. In S. B. Cooper, B. Löwe, and L. Torenvliet, editors, *CiE 2005*, volume 3526 of *Lecture Notes in Computer Science*, pages 35–48. Springer-Verlag, 2005.
7. J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
8. A. Bolychevsky, C. R. Jesshope, and V. Muchnick. Dynamic scheduling in RISC architectures. *IEE Proceedings Computers and Digital Techniques*, 143(5):309–317, 1996.
9. A. J. C. Fox. *Algebraic Representation of Advanced Microprocessors*. PhD thesis, Department of Computer Science, University of Wales Swansea, 1998.
10. A. J. C. Fox and N. A. Harman. Algebraic models of correctness for abstract pipelines. *Journal of Logic and Algebraic Programming*, 57(1/2):71–107, 2003.
11. N. A. Harman and J. V. Tucker. Algebraic models of microprocessors: Architecture and organisation. *Acta Informatica*, 33:421–456, 1996.
12. E. C. R. Hehner, L. E. Gupta, and A. J. Malton. Predicative methodology. *Acta Informatica*, 23:487–505, 1986.

13. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Fransisco, third edition, 2003.
14. J. C. Hoe and Arvind. Operation-centric harware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(9):1277–1288, 2004.
15. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, second edition, 2001.
16. C. R. Jesshope and B. Luo. Micro-threading: A new approach to future RISC. In *ACAC 2000*, pages 34–41. IEEE Computer Society Press, 2000.
17. W. D. Maurer. A theory of computer instructions. *Journal of the ACM*, 13(2):226–235, 1966.
18. A. Ponse. Program algebra with unit instruction operators. *Journal of Logic and Algebraic Programming*, 51(2):157–174, 2002.
19. D. Sima. Decisive aspects in the evolution of microprocessors. *Proceedings of the IEEE*, 92(12):1896–1926, 2004.