Verifying Out-of-Order Executions*

W. Damm† A. Pnueli‡

October 3, 1997

Abstract

The paper presents an approach to the specification and verification of out-of-order execution in the design of micro-processors. Ultimately, the appropriate statement of correctness is that the out-of-order execution produces the same final state as a purely sequential machine running the same program.

1 Introduction

Modern processor architectures such as the PowerPC or the DEC Alpha employ aggressive implementation techniques to sustain peak-throughput of instructions. Multiple functional units inside the data-path allow for concurrent execution of multiple instructions and allow to hide latencies stemming from data-dependencies as well as varying pipeline delays. The design of controllers maintaining compatibility with sequential program execution while supporting out-of-order as well as speculative execution of instructions, interrupts, and load/store buffers is both challenging and error-prone (c.f. e.g. [HP96]).

As the complexity of designs has grown, so has the need for advanced validation techniques. The need for formal verification tools to support industrial design processes is now recognized [Keu96] and is apparent by the introduction of commercial verification tools. However, while circuit comparison and - to a lesser extend - property verification based on symbolic model-checking [BCM+92] have found their way into industrial applications (c.f. e.g. [CCH+96, BM97]) , coping with the complexity of industrial designs remains a key challenge, requiring complementary proof-methods to be combined in verification environments. In particular, proof-methods based on decision procedures for first-order logic [BD94], [JDJ95] have gained high attention due to their ability to naturally cope with abstractions from data-computations

*This research was supported in part by a gift from Intel, a grant from the Minerva foundation, and an *Infrastructure* grant from the Israeli Ministry of Science and the Arts.

†OFFIS, Oldenburg, Germany

‡Weizmann Institute of Science, Rehovot, Israel

when analyzing complex control circuitry while allowing full automation of the proof (in contrast to approaches relying on interactive theorem proving such as [GM93, ORSS94, SM96]).

In this paper we propose a proof-method for proving correctness of such processor designs based on *refinement*, and illustrate it by showing correctness of the Tomasulo algorithm [HP96] supporting out-of-order execution of floating point instructions. The key features of our approach are

- The introduction of an intermediate abstraction level formalizing *data-driven* execution of instruction streams
- The ability to cope with *generic* designs, in particular establishing correctness for an arbitrary number of functional units
- The potential for a high degree of automation of the correctness proof
- A formal basis of the method within the established theory of refinement of synchronous transition systems.

We see the introduction of an intermediate abstraction level formalizing the concept of data-driven execution as a key aspect in the verification of modern processor architectures. Super-scaler processors rely on exploiting the parallelism inherent in the instruction stream and schedule their execution as soon as operands are available - independent of program order. The model we develop captures the algorithmic essence of such a data-driven execution of instructions, and thus provides a natural basis for establishing a refinement relation between data-structures such as reservation stations and a reorder buffer appearing in the data-path and reductions of the executed programs according to the abstract model of data-driven execution. In particular, contrary to [BD94], we are able to give natural meaning to *any* state of the hardware within our intermediate model, and thus need not code our refinement relation by relying on the existence of a flushing mechanism. Clearly the complexity of a refinement proof depends on the complexity of the refinement relation; while flushing may be appropriate in static pipelines with an essentially fixed delay, its extension to out-of-order execution [J.B96] is cumbersome due to the loss of information when flushing dynamically scheduled hardware, information which has to be painfully reconstructed when actually carrying out the proof.

We note, that also [HIKB96] work with a logic of uninterpreted functions, but use the techniques of reducing property verification to verifying the property for finite-instantiations of the model, which in general (and in particular for the application to super-scalar processors discussed in this paper) is not complete.

We base this paper on the semantic model of *synchronous transition systems* ([KP96], [PS97]), a variant of the clocked transition systems used in [MP96] in particular providing the concept of *step*. Steps will correspond to clock-cycles at the hardware-level, while a step of our model of data-driven execution will correspond to the reduction of a subset of all “enabled” instructions. Transitions are expressed using first-order transition predicates.

Work outside the scope of this paper provides the tool-basis integrating the proof-method described into a verification environment for hardware-verification [DJS94, DHP95].

This paper is structured as follows. The next section presents a short summary of the underlying mathematical model of synchronous transition systems and their refinement theory. Ultimately, the implementation has to be compatible with the sequential reference model developed in Section 3. Section 4 introduces the formal model of data-driven execution of programs. We give a formal definition of the Tomasulo Algorithm in Section 6. The final section contains the refinement relation and the correctness proof.

2 Synchronous Transition Systems and their Refinement

2.1 Synchronous Transition Systems

As our computational model we take synchronous transition systems ([KP96], [PS97]. A *synchronous transition system* (STS) $S = \langle V, \Theta, \rho \rangle$, consists of the following components:

- V : A finite set of typed *system variables.* We define a *state* s to be a type-consistent interpretation of V, assigning to each variable $u \in V$ a value $s[u]$ over its domain. We denote by Σ the set of all states.

- Θ : The *initial condition.* A satisfiable assertion characterizing the initial states.

- ρ : A *transition relation.* This is an assertion $\rho(V, V')$, which relates a state $s \in \Sigma$ to its possible successors $s' \in \Sigma$ by referring to both unprimed and primed versions of the system variables. An unprimed version of a system variable refers to its value in s, while a primed version of the same variable refers to its value in s'. For example, the assertion $x' = x + 1$ states that the value of x in s' is greater by 1 than its value in s. If $\rho(s[V], s'[V]) = true$, we say that state s' is an *S-successor* of state s.

Let $S = \langle V, \Theta, \rho \rangle$ be an STS. A *computation* of S is an infinite sequence of states σ: $s_0, s_1, s_2, \ldots$, satisfying the following requirements:

- *Initiation*: s_0 is initial, i.e., $s_0 \models \Theta$.

- *Consecution*: State s_{j+1} is a S-successor of s_j, for each $j = 0, 1, \ldots$.

2.2 Refinement between Systems

Our method for proving correctness of various proposed algorithms for out-of-order execution is based on the notion of *refinement*. This concept is based on the comparison between two systems, consisting of an *abstract system* $S_A = \langle V_A, \Theta_A, \rho_A \rangle$ and a *concrete system*

$S_C = \langle V_C, \Theta_C, \rho_C \rangle$. The abstract system serves as a specification and is expected to capture all the acceptable correct executions of a program for the considered micro-processor. The concrete system will typically represent one of the proposed out-of-order execution algorithms, whose correctness we wish to establish. Correctness is established by proving that S_C *refines* (or *implements*) the specification S_A. Refinement essentially means that every computation of S_C corresponds to some computation of S_A. Note that S_C is not required to implement *all* computations of S_A but only a subset thereof.

Existing variants of the refinement relation differ by the way a computation of S_C is matched against its corresponding S_A- computation. We proceed to explain the variant we intend to use.

Let Ω denote a set of elements, to which we refer as the *domain of observations*. Let Σ_A and Σ_C denote the sets of the abstract and concrete states, respectively. Let $\mathcal{O}_A: \Sigma_A \mapsto \Omega$ and $\mathcal{O}_C: \Sigma_C \mapsto \Omega$ be two functions to which we refer as the *abstract* and *concrete observation functions*, respectively. These functions capture the part of the state we wish to observe in the two systems, and it is only these parts which will be compared by the refinement relations.

Given a general system S and an observation function $\mathcal{O}: \Sigma \mapsto \Omega$, we define an *observation of* S to be an infinite sequence of Ω-elements obtained by applying $\mathcal{O}$ to a computation of S. We denote by $Obs_{\mathcal{O}}(S)$ the set of all observations of S according to the observation function $\mathcal{O}$.

Given systems S_A, S_C, and their observation functions $\mathcal{O}_A$ and $\mathcal{O}_C$, we say that S_C *refines* S_A *according to the observation pair* $(\mathcal{O}_C, \mathcal{O}_A)$, denoted $S_C \sqsubseteq_{(\mathcal{O}_C, \mathcal{O}_A)} S_A$, if $Obs_{\mathcal{O}_C}(S_C) \subseteq Obs_{\mathcal{O}_A}(S_A)$. That is, every observation of S_C is also an observation of S_A. When the identity of the observation functions is obvious from the context we simply write $S_C \sqsubseteq S_A$ to indicate that S_C refines S_A.

2.3 Verifying Refinement

Here we present the proof method we intend to use in our verification. The proof method is based on following the different states in a given S_C-computation and showing how we can incrementally construct a corresponding S_A-computation which yields the same observation. This incremental construction is based on the identification of a *refinement relation* $\alpha \subseteq \Sigma_C \times \Sigma_A$ holding between S_C-states and S_A-states, and establishing the following three conditions:

1. For every S_C-initial state $s_C \in \Sigma_C$ (i.e., $s_C \models \Theta_C$), there exists an S_A-initial state $s_A \in \Sigma_A$ such that $(s_C, s_A) \in \alpha$.

2. For every states $s_C, s'_C \in \Sigma_C$ and $s_A \in \Sigma_A$, such that $(s_C, s_A) \in \alpha$ and s'_C is an S_C-successor of s_C, there exists a state $s'_A \in \Sigma_A$ such that $(s'_C, s'_A) \in \alpha$ and s'_A is an S_A-successor of s_A.

3. For every states $s_C \in \Sigma_C$ and $s_A \in \Sigma_A$ such that $(s_C, s_A) \in \alpha$, $\mathcal{O}_C(s_C) = \mathcal{O}_A(s_A)$. That is, every two α-related states agree on their observed parts.

Obviously, conditions 1 and 2 ensure that, for every concrete computation $\sigma^c\colon s_0^c, s_1^c, \ldots$, we can construct an abstract computation $\sigma^a\colon s_0^a, s_1^a, \ldots$, such that $(s_j^c, s_j^a) \in \alpha$, for every $j = 0, 1, \ldots$. By condition 3, the two computations yield identical observations.

To make these semantic conditions into a syntactic rule, we represent the refinement relation α as an assertion (first-order formula) $\alpha(V_C, V_A)$. In addition, we request the user to identify an initialization function $I\colon \Sigma_C \mapsto \Sigma_A$ which selects the abstract state required in condition 1, and a continuation function $\delta\colon \Sigma_C \times \Sigma_C \times \Sigma_A \mapsto \Sigma_A$ which selects the abstract state required in condition 2. This leads to the following proof rule:

For assertion $\alpha(V_C, V_A)$,
function $I(V_C)$ mapping Σ_C into Σ_A,
and function $\delta(V_C, V_C', V_A)$ mapping $\Sigma_C \times \Sigma_C \times \Sigma_A$ into Σ_A

$$\begin{array}{lrcl}
\textbf{R1.} & \Theta_C(V_C) & \rightarrow & \Theta_A(I) \wedge \alpha(V_C, I) \\
\textbf{R2.} & \rho_C(V_C, V_C') \wedge \alpha(V_C, V_A) & \rightarrow & \rho_A(V_A, \delta) \wedge \alpha(V_C', \delta) \\
\textbf{R3.} & \alpha(V_C, V_A) & \rightarrow & \mathcal{O}_C(V_C) = \mathcal{O}_A(V_A) \\
\hline
 & & C \sqsubseteq A &
\end{array}$$

Rule REF: Proving Refinement

2.4 Presentation of STS's

To make our presentation of an STSmore readable, we use standard programming constructs such as assignments, if- and case-statements, as well as sequential and parallel composition to denote the effect of a *single* step. The translation of this notation to the formal model presented in subsection 2.1 is straightforward. We assume in this programming notation a rich set of types, including range-types, records with named components, and type constructors such as sets and arrays. We freely use VHDL's package concept to collect type-, variable-, and function-definitions. A *design* consists of a set of (possibly parameterized) *systems*, which define in their *behavior* section the step- relation of an STS, possibly as the product of the step-relation of instances of other systems. Declarations can either be imported from packages or explicitly given within a system. The defining expression of a definition is evaluated each step. See e.g. chapter 3 for examples.

3 The Reference Machine: System SEQ

As a first step, we present a system SEQ, which serves as the reference model. We restrict ourselves to the interface between the I-Fetch and the Execution- Unit of the processor and defer treatment of load-store instructions and speculation to subsequent papers, hence SEQrepresents a strictly sequential execution of a given non-branching register-to-register program. Obviously, all proposed designs for out-of-order execution should be compatible with this reference model. Indeed, the ultimate result of this paper is to show that the TOMASULO algorithm is a refinement (and hence implementation) of the reference system SEQ.

3.1 Data Types and Declarations

In Fig. 1, we present a package which declares the main types and variables, reused throughout all three refinement levels discussed in this paper. Package R/R_PROGRAM accepts as param-

package R/R_PROGRAM(R, N: $\mathbb{N}^+$) **is**
types

$$
\begin{array}{lcl}
\mathit{REG\_ID} & = & [0..R]; \\
\mathit{TARGET} & = & \mathit{REG\_ID}\ ; \\
\mathit{VALUE} & = & \textbf{floating-point-number}\ ; \\
\mathit{OP\_TYPE} & = & \{\textbf{fpadd}, \textbf{fpsub}, \textbf{fpmlt}, \textbf{fpdiv}\}; \\
\mathit{SRC} & = & \textbf{array}[1..2]\ \textbf{of}\ \mathit{REG\_ID}\ ; \\
\mathit{INST} & = & [\mathit{op}: \mathit{OP\_TYPE},\ t: \mathit{TARGET},\ \mathit{src}: \mathit{SRC}]
\end{array}
$$

variables

$prog$: **array**$[0..N]$ **of** INST

definitions

$$
lw(i: [0..N], r: [0..R]): [0..N] = \left(\begin{array}{lll}
\textbf{if} & i = 0 & \textbf{then}\ 0 \\
\textbf{else if} & prog[i].t = r & \textbf{then}\ i \\
\textbf{else} & lw(i-1, r) &
\end{array}\right)
$$

end package

Figure 1: Package R/R_PROGRAM

eters R which is a positive integer specifying the number of registers (starting at register $reg[0]$, there are $R+1$ registers). We assume, that processing of the program to be executed by the I-Fetch Unit results in a stream of floating-point instructions *prog* of length N, thus representing the content of the instruction-queue of Figure 5. The declarations include a declaration for type *INST* which represents a single instruction of the program. In this presentation, *OP_TYPE* is the type of the operation code, ranging over the set of floating-point arithmetical operations (i.e., $+$, $-$, $\times$, etc.). The field *TARGET* specifies the number of the register in which the instruction stores its computed value. For $j = 1, 2$, $SRC[j]$ identifies one of the operands to which the arithmetical operation is applied, and is given as a register index $r \in [0..R]$.

The function $lw(i, r)$ (*last writer*), provides for each instruction index $i \in [0..N]$ and each register number $r \in [0..R]$ the identity of the last instruction writing on r preceding instruction

i in a sequential execution of the program. It is defined to be i if $prog[i]$ writes on register r (i.e., $prog[i].t = r$), and is equal to $lw(i-1, r)$ in all other cases, except for $i = 0$ in which it is defined to equal 0.

3.2 System SEQ

In Fig. 2, we present system SEQ, which models sequential execution of a given input program.

system SEQ(R, N: $\mathbb{N}^+$) **is**
import R/R_PROGRAM(R, N)
variables
  reg: **array**[0..R] **of** *VALUE* **init** 0;
  top: [1..N+1] **init** 1
behavior
  if $top \leq N$ **then**

$$\begin{array}{lcl} reg[prog[top].t] & := & do\_op\begin{pmatrix} prog[top].op \\ reg[prog[top].src[1]] \\ reg[prog[top].src[2]] \end{pmatrix}; \\ top & := & top + 1 \end{array}$$

  end if
end system

Figure 2: System SEQ

System SEQ processes a stream of instructions presented as array *prog* in a strictly sequential order, incrementing the top-pointer of the instruction-queue, and operating on a floating-point register-file *reg*. The execution of a single instruction is represented by the generic function *do_op* which accepts as arguments one of the four floating operations and two operands, assumed to be floating-point numbers.

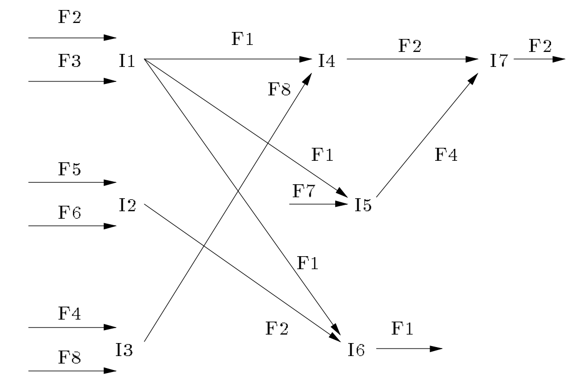
4 The Specification Model: Data-driven Execution of Instruction Streams

4.1 Motivation

Modern processor architectures exploit the parallelism inside the instruction stream by executing *independent* instructions concurrently using multiple functional units. This independence relation can be computed from the program; in the situation of pure floating-point arithmetic instructions considered in this paper, it can be inferred from the program text, and there is a trade-off between compiler- and hardware-measures to exploit it. As soon as data-dependencies across load-store instructions are considered, data- dependencies can only be computed at run-time. In a later paper, we show how to extend the simple model presented here to also cope with such dynamic dependencies, as well as speculative execution of instructions as resulting from branch-prediction.

For the sake of this paper, a *program* is simply a finite array of *instructions*, which are restricted to pure register-to-register floating point instructions. Intuitively, two instruction with indices i and j are *data-dependent* iff $i < j$ and i is the *last* writer on a source register of j. We refer to this situation by saying that instruction i is the *producer* of a value *consumed* by instruction j. Package R/R_PROGRAM of Fig. 1 contains a declaration of the function lw, such that $lw(j, r)$ identifies for instruction j the *last writer* on register r. Note the special case, in which *no* previous instruction has written on the considered register, and then $lw(j, r)$ returns the value 0.

Fig. 3 shows a sample program as well as its *dependency graph*, whose arcs are additionally labeled with the register shared between producer and consumer. The dependency graph of a program P can be viewed as a skeleton of a partial order $<_P$ giving the coarsest restriction on execution order of Ps instructions: if $i < j$, then the execution of j cannot be initiated before the execution of i has completed. It also makes explicit the parallelism inherent in P: two instructions can (up to structural hazards) be executed concurrently iff they are unordered w.r.t. $<_P$. The *minimal elements* of this partial order are the instructions j such that there exists no instruction $i < j$.



I1	FADD F1,F2,F3
I2	FMUL F2,F5,F6
I3	FSUB F8,F4,F8
I4	FADD F2,F1,F8
I5	FDIV F4,F1,F7
I6	FADD F1,F1,F2
I7	FSUB F2,F2,F4

Figure 3: Dependency Graph

Observe that the dependency graph pays no attention to the identity of the registers in which instructions store their results, except for their use in identifying the producer-consumer relations between instructions. For example, the graph allows the execution of I_6 after I_1 and before I_4, even though both I_1 and I_6 write to F_1 and I_4 needs the value produced by I_1. In fact, the dependency graph can be interpreted as denoting a set of *processes* - the instruction instances -, which communicate via *private channels* - represented by the dependency arcs - via message passing. In the actual hardware design, this is implemented by broadcasting values computed by an instruction instance on the common data-bus, thus updating private copies of the register in a unique buffer associated with each active instruction instance, a technique called *register renaming*, see Figure 5.

We now describe the concept of data-driven execution informally using the sample dependency graph. This can be viewed as restating the well-known concept of a data-driven execution of functional languages as implemented by data-flow architectures.

A *state* of the data-driven execution of P is a *prefix-closed* subset of instructions including all minimal elements together with a valuation function giving the values computed by all the instructions in this set. Intuitively, these are the instructions which have *completed* their execution. A not yet completed instruction is *enabled* in a state iff all of its immediate predecessors have completed their execution. A *step* in a data-driven execution of P takes a state s to a successor state obtained by *firing* some subset of instructions enabled in s. This will enable to smoothly embed processor cycles, in which none or - in the context of super-scalar architectures - more than one instructions have completed their execution. The current valuation is then extended by applying the fp-operations of selected instructions to the current valuation of their immediate predecessors (or the initial value of the source register for minimal instructions). In the example above, a possible sequence of fired instructions is $\{I_1, I_2, I_3\}, \{I_4, I_6\}, \{\}, \{I_5\}, \{I_7\}$. It is obvious that no RAW-hazards occur in such a data-driven execution. As previously explained, WAW- and WAR hazards are eliminated at the RT-level by register renaming; at the current level of abstraction, register-renaming is realized by having private *value*-registers for each instruction instance.

4.2 Specification

We now formalize this concept of data-driven execution by defining a system called NON-DET, which will serve as the specification for out-of-order execution, by capturing all the possible behaviors which are compatible with the dependency graph of the program. The set of nodes is simply given by the program, presented as an array of instructions, extended by index 0 representing the minimal element. A state of the data-driven execution is encoded by a valuation of system variables c and v, where the boolean array c identifies the completed instructions, while the floating-point array v represents their computed values. Note, that the minimal element is the only initially completed node. In Fig. 4, we present system NON-DET.

In addition to the arrays c and v, system NON-DET also declares a register file $reg[0..R]$. As evident from the system definition, these registers are initialized to 0, and assume non-zero values only when all instructions have been completed. At this point, $reg[r]$ assumes the value computed by the last writer on register r in program order.

$$
\begin{array}{l}
\textbf{system}\ \textsc{non-det}(R, N\colon \mathbb{N}^{+})\ \textbf{is} \\
\textbf{import}\ \mathrm{R/R\_P}\textsc{rogram}(R, N) \\
\textbf{variables} \\
\quad \begin{array}{rl}
c\colon & \textbf{array}[0..N]\ \textbf{of boolean init}\ \mathit{true}, \underbrace{\mathit{false}, \ldots, \mathit{false}}_{N}\ ; \\
v\colon & \textbf{array}[0..N]\ \textbf{of}\ \mathit{VALUE}\ \textbf{init}\ 0\ ; \\
\mathit{reg}\colon & \textbf{array}[0..R]\ \textbf{of}\ \mathit{VALUE}\ \textbf{init}\ 0
\end{array} \\
\textbf{definitions} \\
\quad \begin{array}{rcl}
\mathit{ready}(i\colon[1..N], j\colon[1..2])\colon \textbf{boolean} & = & c[\mathit{lw}(i-1, \mathit{prog}[i].\mathit{src}[j])]; \\
\mathit{gop}(i\colon[1..N], j\colon[1..2])\colon \mathit{VALUE} & = & v[\mathit{lw}(i-1, \mathit{prog}[i].\mathit{src}[j])];
\end{array} \\
\quad x \quad = \quad \textbf{choose}\ i \in [1..N]\ \textbf{s.t.}\ \neg c[i] \wedge \mathit{ready}(i,1) \wedge \mathit{ready}(i,2) \\
\textbf{behavior} \\
\quad \textbf{if}\ \neg c[x] \wedge \mathit{ready}(x,1) \wedge \mathit{ready}(x,2)\ \textbf{then} \\
\qquad \begin{array}{lcl}
c[x] & := & \mathit{true}; \\
v[x] & := & \mathit{do\_op}(\mathit{prog}[x].\mathit{op},\ \mathit{gop}(x,1),\ \mathit{gop}(x,2))
\end{array} \\
\quad \textbf{end if}\ ; \\
\quad \textbf{if}\ \bigwedge_{i=1}^{N} c[i]\ \textbf{then}\ \mathop{\Big\|}_{r=0}^{R} \mathit{reg}[r] := \mathit{gop}(N, r) \\
\textbf{end system}
\end{array}
$$

Figure 4: System NON-DET

Besides declarations of variables, Fig. 4 contains the definitions of two auxiliary functions which are used to simplify the following statements.

For $j \in \{1, 2\}$, the boolean function (predicate) $\mathit{ready}(i, j)$ is defined to be true iff operand j of instruction $\mathit{prog}[i]$ is *ready*. This is the case when operand j names a register r and the instruction that was supposed to compute the value of this register has already been completed. The index of this instruction is given by $\mathit{lw}(i-1, r)$. Therefore, operand j is ready iff instruction $\mathit{prog}[\mathit{lw}(i-1, r)]$ has completed its execution, i.e., iff $c[\mathit{lw}(i-1, r)] = \mathit{true}$.

In a similar way, function $\mathit{gop}(i, j)$ fetches the value of operand j of instruction i. This value is given by $v[\mathit{lw}(i-1, \mathit{prog}[i].\mathit{SRC}[j])]$. Note that in the special case that $\mathit{lw}(i-1, r) = 0$, which means that no (sequentially) previous instruction has assigned a value to register r, this yields $v[\mathit{lw}(i-1, r)] = v[0] = 0$, which is the correct value of all unassigned registers.

The last definition uses the non-deterministic selection operator **choose**. This operator assigns to x any index i in the range $[1..N]$ which satisfies the requirement $\neg c[i] \wedge \mathit{ready}(i, 1) \wedge \mathit{ready}(i, 2)$. If no such index exists, x is assigned an arbitrary value in the specified range. This is why, even after the selection, it is necessary to test whether x satisfies the specified requirement.

The system selects an instruction x which is ready but has not been completed. It is this selection which introduces the high non-determinism of this system. If such instruction exists, the system proceeds to set $c[x]$ to *true* and to assign to $v[x]$ its proper value. The value of $v[x]$ is computed using the generic function *do_op* which performs the actual computation, applying the operation $prog[i].op$ to the appropriate arguments.

4.3 Properties

In this subsection we state and verify several important properties of system NON-DET.

As a first step, we establish the following property of function lw:

$$\psi_1: \quad lw(i,r) = j \neq 0 \quad \rightarrow \quad prog[j].t = r.$$

This property states that the only non-zero values that $lw(i,r)$ can assume are indices of instructions whose target register is r. The property can be proven by induction on i.

A second important property of function lw is given by

$$\psi_2: \quad lw(i,r) = j \quad \rightarrow \quad j \leq i \ \wedge \ \forall k \in (j..i].\, prog[k].t \neq r.$$

This property captures the intended role of $lw(i,r)$ as recording the latest instruction j writing on register r which precedes or coincides with instruction i. The property states that if $lw(i,r) = j$, then there exists no instruction k, where $j < k \leq i$, (i.e., no later instruction between j and i) which writes on $reg[r]$. Note that this property also holds for the case that $lw(i,r) = 0$ meaning that, up to (and including) $prog[i]$, no instruction writes to register r. Property ψ_2 can be proven by induction on i, using the inductive definition of $lw(i,r)$.

We proceed by presenting the following invariance properties for system NON-DET:

$$\begin{array}{lccl}
\psi_3: & c[0] \ \wedge \ v[0] = 0 & & \\
\psi_4: & i > 0 \wedge c[i] & \rightarrow & \left(\begin{array}{l} ready(i,1) \ \wedge \ ready(i,2) \ \wedge \\ v[i] \ = \ do_op(prog[i].op,\ gop(i,1),\ gop(i,2)) \end{array} \right) \\
\psi_5: & reg[r] & = & \textbf{if } \left(\bigwedge_{i=1}^{N} c[i]\right) \textbf{ then } v[lw(N,r)] \textbf{ else } 0
\end{array}$$

These three invariants claim that the initial values of $c[0]$ and $v[0]$ and the effects of the assignments to the variables v and reg are persistent. That is, once they are assigned a new value (different from their initial one) they retain the same value forever.

Even though system NON-DET is highly non-deterministic, the terminal state reached at the end of an execution is uniquely determined by the the input $prog[1..N]$. This is stated by the following theorem, claiming that system NON-DET is determinate:

Theorem 1 *The final state reached by an execution of system* NON-DET *is uniquely determined by the input $prog[1..N]$. In particular, every two executions of* NON-DET *over the same input terminate in the same final state, which is also the final state in a strictly sequential execution of the program.*

Proof: We first show a strong confluence property of system NON-DET: for any reachable state s, if $s \Rightarrow s_1 \wedge s \Rightarrow s_2$, then there exists s' s.t. $s_1 \Rightarrow s' \wedge s_2 \Rightarrow s'$ (where $\Rightarrow$ abbreviates the transition relation of NON-DET). Hence (by induction), for *any* reachable states s_1 and s_2 there exist (possibly empty) computation sequences originating from s_1 and s_2, respectively, leading to a common successor state. In particular, given P, all computations of NON-DET, among them the unique computation evaluating all instructions in program order, reach the same final state. Since by ψ_4 the final state uniquely determines the valuation of the register-file, any data-driven computation of P produces the same valuation of the register-file as the sequential execution of P.

5 Proving that NON-DET implements SEQ

The ultimate goal of this paper is to prove, that the Tomasulo Algorithm as presented in Section 6 is a correct implementation of system SEQ which serves as our reference model. However, as is often the case, this goal can best be achieved by breaking it into subgoals. We introduced the intermediate (specification) system NON-DETand intend to establish the two subgoals TOMASULO $\sqsubseteq$ NON-DET and NON-DET $\sqsubseteq$ SEQ. By transitivity of the refinement relation, it follows that TOMASULO is a refinement (and hence correct implementation) of SEQ. Actually, we start by proving refinement in the other direction. Namely, we first prove SEQ $\sqsubseteq$ NON-DET. Since both systems contain an array variable called *reg*, we denote its SEQ-instance by reg_C and its instance in NON-DET by reg_A. As observation functions we choose

$$\mathcal{O}_C = \textbf{if } top > N \textbf{ then } reg_C \textbf{ else } 0 \textit{ and } \mathcal{O}_A = reg_A.$$

As the refinement relation, we choose

$$\alpha: \quad \left(\begin{array}{l} \wedge \quad \mathcal{O}_C = \mathcal{O}_A \;\wedge\; \forall i{:}\,[0..N].\,(c[i] \leftrightarrow i < top) \\ \wedge \quad \forall r{:}\,[0..R].\, reg_C[r] = v[lw(top-1, r)] \end{array} \right)$$

For the initialization function, we choose

$$\begin{array}{rcl} c_I & : & \lambda i{:}\,[0..N].\,(i = 0) \\ v_I & : & \lambda i{:}\,[0..N].\,0 \\ reg_{A_I} & : & \lambda r{:}\,[0..R].\,0 \end{array}$$

For the continuation function, we choose

$$\begin{array}{rcl} c_\delta & : & \lambda i{:}\,[0..N].\,\textbf{if } i = top \textbf{ then } \textit{true} \textbf{ else } c[i] \\ v_\delta & : & \lambda i{:}\,[0..N].\, \left(\begin{array}{ll} \textbf{if} & i = top \\ \textbf{then} & do\_op(prog[i].op,\, gop(i,1),\, gop(i,2)) \\ \textbf{else} & v[i] \end{array} \right) \\ reg_{A_\delta} & : & reg'_C \end{array}$$

Having shown, that SEQ refines NON-DET, we proceed to argue that, due to the determinateness of NON-DET established by Theorem1, it also follows, that NON-DETrefines SEQ, establishing our first subgoal. Let π be any given program serving as input to both SEQ and NON-DET. Let $\overline{f}_C$ be the valuation of $reg_{_C}$ at the end of an execution of system SEQ over π, and similarly, let $\overline{f}_A$ be the (by theorem 1 unique)valuation of $reg_{_A}$ at the end of a NON-DET execution over π. It is not difficult to see, that all observations of SEQ associated with π have the form $\overline{0}, \ldots, \overline{0}, \overline{f}_C$, and all π-associated observations of NON-DET have the form $\overline{0}, \ldots, \overline{0}, \overline{f}_A$. By SEQ $\sqsubseteq$ NON-DET we can conclude that $\overline{f}_C = \overline{f}_A$. Hence SEQ and NON-DET have *equal* sets of observations, which allows us to conclude that NON-DET $\sqsubseteq$ SEQ.

6 The Tomasulo Algorithm

6.1 The System TOMASULO

In this section, we give a formal definition of the Tomasulo Algorithm for data-driven execution of instructions, which is based on the informal presentation in [HP96]. Fig. 5 shows an instance of the data-path with three functional units. Instructions flow from the instruction queue to one of the reservation-stations, where they await availability of their sources. Once these are available, pipelined execution within the functional unit is initiated. When the result is available, it is passed on the common data bus to the register-file and all reservation-stations. The following subsections describe the structure of the controller realized through communicating subsystems, the employed system variables, and the transition relations of the controllers, as well as properties of the system.

In Fig. 6, we present the definition of system TOMASULO, which represents a possible implementation of the Tomasulo algorithm. As we see from the definition in Fig. 6, system TOMASULO consists of a combination of instances of several subsystem classes. The system contains a single subsystem of each of the following two types:

- **Dispatcher** This module dispatches instruction in *program-order* for execution in a free slot of the reservation station of a matching functional unit.
- **Write_back** This module writes back the result of completed instructions into the register file under appropriate conditions.

For each functional unit $FU \in [1..U]$, system TOMASULO contains an instance of subsystem FUN, defined as follows:

system FUN(FU: $[1..U]$) **is**
behavior
  EXEC[FU] $\|$ SNOOPER[FU] $\|$ COMPLETION[FU]
end system

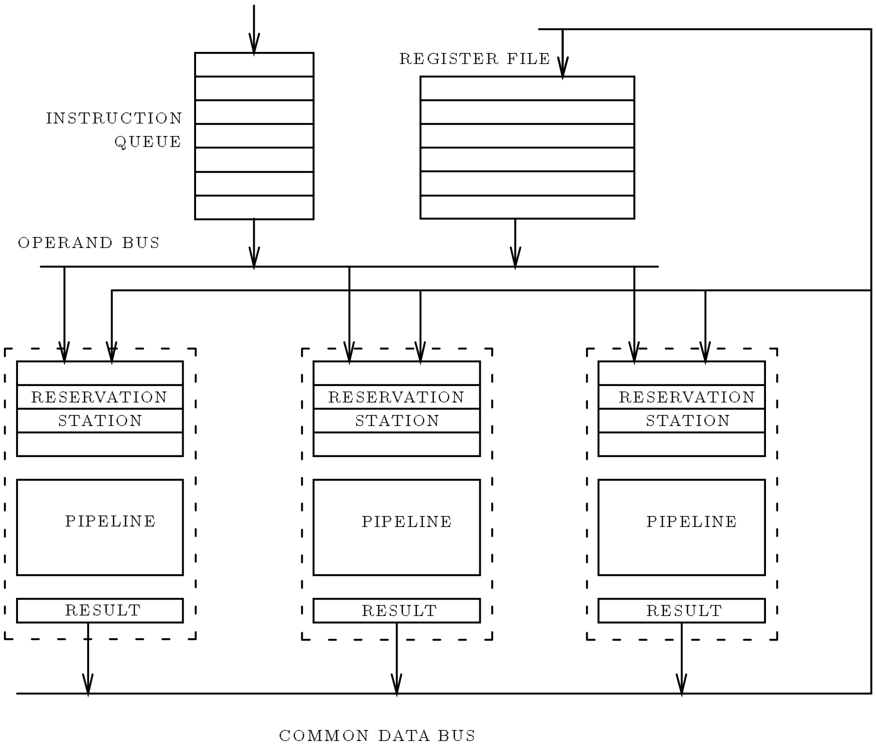


Figure 5: Data-path

System FUN consists of the following subsystems:

- **Exec** – This subsystem completes the execution of an enabled instruction and places the result in the *result* register.

- **Snooper** – This subsystem continuously spies on the traffic passing on the *cdb*. Whenever it observes a result of an instruction producing a source-value for one of the instructions waiting at its reservation stations, it copies the observed value to the appropriate operand field in the record of the pending instruction.

- **Completion** – This subsystem clears the result register and the reservation slot corresponding to an instruction whose result has just been placed on the *cdb*.

In addition to the program parameters R and N, system TOMASULO expects the parameter U which specifies the number of functional units, and the parameter Z which specifies the number of slots in the reservation station of each functional unit.

In order to maintain the producer-consumer relation between instructions, the algorithm maintains HW_NAMEs of each instruction, which is a combination of the index of the functional unit and the index of the slot in its reservation station, in which the instruction is awaiting execution. Our version of the algorithm maintains the auxiliary field of the abstract *I_NAME* of an instruction (its index in the program's array) in order to facilitate the proof. Auxiliary variables are only updated and copied from one record to another but they never affect the flow of control, hence it is easy to prove that deleting these will not affect valuations of other system variables. The *producer* of a variable is hence characterized by both its *HW_NAME* and *I_NAME*. Floating-point values as well as floating-point operations are uninterpreted in the algorithm. Register-entries are structured into three fields, including a *BUSY* flag indicating that a dispatched and not yet completed instruction has the register as target. As long as the register is *busy*, the *PRODUCER* field maintains the identity of this instruction, otherwise its *VALUE* field carries the actual value of the register. Note, that exactly the same type of information is required for sources of an instruction maintained in a reservation station. In addition, a slot maintains an *OCCUPIED* field indicating whether the slot is occupied or free, a *BUSY* flag indicating whether the result of the instruction has already been computed, the *id* of the allocated instruction, as well as its target register and opcode-field. A *VALUE* field is included as auxiliary variable. The consistent use of *R_TYPE* indicates where *copies* of the information relevant to an instruction reside in the data-path, a concept elaborated in the definition of the refinement relation.

Given the pragmatics of the types, it suffices to relate the system variables to Fig. 5. We denote the register-file by RF. For each functional unit FU, $rs[FU, S]$ denotes slot S in its reservation station, while $result[FU]$ stores the result (including producer id as well as the target register) until availability of the cdb. Variable top points to the next instruction that should be dispatched.

The definitions of *master*, *bus_active*, and *cdb* handle the process of bus arbitration. We start by selecting an active result register, if one exists, and placing its index in *master*. We then define the boolean expression *bus_active* to be true if *master* indeed points to an active result register. Finally, we define *cdb* to be this active register.

In a similar way, the definitions of FU_n, S_n, and *dispatch* handle the process of selecting a functional unit and a slot in which the next dispatched instruction should be placed. We consult a table *fu_table* in order to decide on the functional unit appropriate for dispatching the next instruction pointed to by *top*, and place its index in FU_n. We then select a free slot in the reservation station indexed by FU_n. Finally, the boolean expression *dispatch* is defined to be true if the program has not terminated yet ($top \leq N$), and the slot selected is indeed free.

We proceed to present and discuss the subsystems comprising the system TOMASULO and start with the definition of system DISPATCHER shown in Fig. 7. As implied by the TOMASULO definitions, the dispatcher becomes active, whenever the top instruction can be allocated, as is indicated by *dispatch* being true. The system places in $rs[FU_n, S_n]$ a new record, whose fields are mostly taken from the instruction $prog[top]$ and its index ($i = top$). Special care is taken in defining the two operand fields $ss[1]$ and $ss[2]$. Normally, $ss[1]$ should be set to a copy of $RF[f_1]$, which provides all the currently available information about the most recent value of this register. An exceptional situation is when precisely in the same clock cycle that the instruction is dispatched, the common data bus happens to carry the value for which $RF[f_1]$ is currently waiting. This is identified by observing that the bus is active, $RF[f_1]$ is waiting for a value, and that the producer for the awaited value matches the producer address in *cdb*. Under these conditions, we should place in $ss[1]$ a record with the value obtained from the *cdb*. This is expressed in the definition of src_1 and similarly, that of src_2

Following is the definition of system WRITE_BACK:

system WRITE_BACK **is**
definitions
$\quad r = cdb.t$
behavior
$\quad$ **if** $\quad bus\_active \;\wedge\; RF[r].b \;\wedge\; RF[r].h = cdb.h$
$\quad\quad\quad \wedge\; (\neg dispatch \;\vee\; f \neq prog[top].t)$
$\quad$ **then** $\quad RF[r] := cdb.rt$
end system

System WRITE_BACK becomes active only if *cdb* is active and the available value is still relevant, i.e. no other instruction has been dispatched to the register named as target by *cdb* since the dispatch-time of the completing instruction. In this case, the default operation of the controller is to pick over the value from the bus and place it in the register. However, if an instruction is currently being dispatched, we should make sure that it does not name f as its target register.

For each functional unit, $FU = 1, \ldots, f$, the system contains a subsystem EXEC$[FU]$, which is presented below:

system EXEC$(FU\colon[1..U])$ **is**
definitions
$\quad enabled(f\colon[1..U], S\colon[1..Z])$: **boolean**
$$= rs[f,S].b \;\wedge\; \bigwedge_{j=1}^{2} \neg rs[f,S].ss[j].b;$$
$\quad e \quad = \;$ **choose** $k\colon[1..Z]$ **s.t.** $enabled(FU,k)$;
$\quad pipe\_req \quad = \; enabled(FU,e)$;
$\quad delay \quad = \;$ **choose** $d \in \{true, false\}$;
$\quad sr \quad = \; rs[FU,e]$;
$\quad nrt \quad = \;$ **if** $pipe\_req$
$\quad\quad\quad$ **then** $\langle false,\; sr.p,\; do\_op(sr.op,\; sr.ss[1].v,\; sr.ss[2].v)\rangle$
$\quad\quad\quad$ **else** $sr.rt$
behavior
$\quad$ **if** $\neg delay \;\wedge\; pipe\_req \;\wedge\; \neg result[FU].a$ **then**
$\quad\quad result[FU] \;:=\; \langle true, nrt, sr.t\rangle$;
$\quad\quad rs[FU,e].rt \;:=\; nrt$
$\quad$ **end if**
end system

System EXEC checks for enabledness of instructions in its reservation station and conceptually initiates pipelined execution. We abstract from the actual processing within the pipeline and guess the delay encountered by waiting for the internal pipeline to become available as well the actual computation time. This guess is represented by the local variable *delay* which is assigned a non-deterministic boolean value. A choice of $delay = true$ represents a decision that no new instruction of this functional unit will be initiated this cycle. Hence a cycle of this controller may be either an idling cycle (modeling passage of time), or an active cycle in which not only the instruction is enabled but also conceptually the time for processing the instruction in the pipeline has passed. In this case, the result register is initialized, in particular its value field contains the result of the application of the required floating-point operation to the floating-point number residing in the source fields. The result-registers active flag is used to prevent destroying a computed, but not yet visible instruction value.

For each functional unit, $FU = 1, \ldots, U$, the system contains a subsystem SNOOPER$[FU]$, defined as follows:

$$
\begin{array}{l}
\textbf{system } \text{SNOOPER}(FU\colon [1..U]) \textbf{ is}\\
\textbf{behavior}\\
\quad \textbf{if } bus\_active \textbf{ then}\\
\qquad \mathop{\Big\|}_{S=1}^{Z} \mathop{\Big\|}_{j=1}^{2} \left[\begin{array}{ll} \textbf{if} & rs[FU,S].b \ \wedge \ rs[FU,S].ss[j].b \\ & \quad \wedge \ rs[FU,S].ss[j].h = cdb.h \\ \textbf{then} & rs[FU,S].ss[j] := cdb.rt \end{array}\right]\\
\textbf{end system}
\end{array}
$$

Whenever the bus is active, the snooper tries to match the producer indicated on the bus with a producer awaited in a source field on one of its busy slots. If successful, all such entries are updated by copying the value and resetting the busy-flag.

For each functional unit, $FU = 1, \ldots, f$, the system contains a subsystem COMPLETION$[FU]$, defined as follows:

$$
\begin{array}{l}
\textbf{system } \text{COMPLETION}(FU\colon [1..U]) \textbf{ is}\\
\textbf{behavior}\\
\quad \textbf{if } bus\_active \ \wedge \ cdb.f = FU \textbf{ then}\\
\qquad\quad rs[FU, cdb.s].o \ := \ false;\\
\qquad\quad result[FU].b \quad := \ false\\
\quad \textbf{end if}\\
\textbf{end system}
\end{array}
$$

An instruction completes its EXE-stage by making the result visible on the *cdb*, i.e. when the functional unit becomes the bus master. Recall from the global declarations, that in such a cycle, *cdb* is just an acronym for the masters result register, hence only protocol actions like freeing the slot, the bus, and the result-register itself are required.

6.2 Properties of System TOMASULO

In this subsection we state and verify several important properties of system TOMASULO. In formulating these properties, we refer to the function lw defined as part of system NON-DET.

The following invariant specifies properties of the data stored in an occupied reservation-station slot:

$$\varphi_1: \quad \left(rs[FU,S] = \begin{array}{l} \langle \ \ true,\ \langle b_0, \langle i, \langle FU_0, S_0\rangle\rangle, v_0\rangle, opr, F, \\ \qquad\qquad \langle b_1, \langle i_1, \langle FU_1, S_1\rangle, v_1\rangle, \\ \qquad\qquad \langle b_2, \langle i_2, \langle FU_2, S_2\rangle, v_2\rangle \quad \rangle \end{array} \right) \quad \rightarrow$$

$$\left(\begin{array}{l} \wedge \ \ i > 0 \ \wedge \ FU = FU_0 \ \wedge \ S = S_0 \\ \wedge \ \ opr = prog[i].op \ \wedge \ F = prog[i].t \\ \wedge \ \ i_1 = lw(i-1, prog[i].src[1]) \ \wedge \ i_2 = lw(i-1, prog[i].src[2]) \end{array} \right)$$

This invariant states that if an occupied entry at slot $rs[FU,S]$ has the producer address $\langle i, \langle FU_0, S_0\rangle\rangle$ then $FU = FU_0$ and $S = S_0$, the operation code and the target address match those of the instruction $prog[i]$, and the source addresses match the instructions that the two operands of $prog[i]$ refer to. Slot S at functional unit FU is considered occupied if $rs[FU,S].o = true$.

For a variable rc: R_TYPE, we define

$$visible(rc): \quad \exists\ FU{:}\,[1..U],\ S{:}\,[1..Z],\ j{:}\,\{1,2\},\ r{:}\,[0..R]$$

$$\left(\begin{array}{l} \vee \ \ rs[FU,S].rt.o \ \wedge \ \left(\begin{array}{l} \vee \ \ rc = rs[FU,S].rt \\ \vee \ \ rc = rs[FU,S].rt.ss[j] \end{array} \right) \\ \vee \ \ result[FU].a \ \wedge \ rc = result[FU].rt \\ \vee \ \ rc = RF[r] \end{array} \right)$$

Thus, a record rc (of type R_TYPE) is considered *visible* if it is either the main record part of an occupied slot, or one of the operand fields of an occupied slot, or is the record part of an active result ($result[FU].a = true$), or equal to one of the registers.

The following invariant states that any of the subfields in a visible producer sub-record $\langle i, \langle FU, S\rangle\rangle$ equals zero iff all subfields are.

$$\varphi_2: \quad visible(\langle b, \langle i, \langle FU, S\rangle\rangle, v\rangle) \quad \rightarrow \quad (i = 0 \ \leftrightarrow \ FU = 0 \ \leftrightarrow \ S = 0)$$

A visible record rc is defined to be *in result* if it is contained in a result register with a true a field.

$$in\_result(rc): \quad visible(rc) \ \wedge \ result[rc.p.h.f].a \ \wedge \ rc = result[rc.p.h.f].rt$$

We define a visible record rc to be *active* if either $rc.b$ is on, or rc is *in result*.

$$active(rc): \quad visible(rc) \ \wedge \ (rc.b \ \vee \ in\_result(rc))$$

Every record of type *R_TYPE* points to a slot through its h: *HW_NAME* field which consists of a functional unit identifier FU and a slot number S. The following invariant states that if record rc is active, then its producer address is nontrivial and the slot to which h points is occupied and contains rc as its rt-field.

$$\begin{array}{ll}\varphi_3\text{:} & rc = \langle b, \langle i, \langle FU, S\rangle\rangle, v\rangle \ \wedge \ active(rc) \quad \rightarrow \\ & i > 0 \ \wedge \ rs[FU, S].o \ \wedge \ rs[FU, S].rt = rc\end{array}$$

The following invariant states that if record rc is visible and $rc.b = true$, then $rc.v = 0$.

$$\varphi_4\text{:} \quad rc = \langle true, \langle i, \langle FU, S\rangle\rangle, v\rangle \ \wedge \ visible(rc) \quad \rightarrow \quad v = 0$$

The following invariant states that if record rc is active and visible and equals the record field of $result[FU]$, then the functional unit mentioned in the hardware address of rc equals FU and the target field of $result[FU]$ matches that of the slot to which rc points.

$$\begin{array}{ll}\varphi_5\text{:} & result[FU] = \langle true, \langle b, \langle i, \langle FU_0, S\rangle\rangle, v\rangle, F\rangle \quad \rightarrow \\ & \qquad \neg b \ \wedge \ FU_0 = FU \ \wedge \ F = rs[FU, S].t\end{array}$$

The following invariant considers two visible records rc_1 and rc_2. It characterizes the correspondence between their n: *I_NAME* field and their h: *HW_NAME* field. In one direction, the invariant claims that if rc_1 and rc_2 have the same instruction number, then they also have the same hardware name (pair $\langle FU, S\rangle$). In the other direction, two *active* records which have the same hardware name have the same instruction number.

$$\begin{array}{ll}\varphi_6\text{:} & \left(\begin{array}{ll}\wedge & visible(rc_1) \ \wedge \ rc_1 = \langle b_1, \ \langle i_1, \langle FU_1, S_1\rangle, v_1\rangle \\ \wedge & visible(rc_2) \ \wedge \ rc_2 = \langle b_2, \ \langle i_2, \langle FU_2, S_2\rangle, v_2\rangle\end{array}\right) \quad \rightarrow \\ & \left(\begin{array}{ll}\wedge & i_1 = i_2 \ \rightarrow \ \langle FU_1, S_1\rangle = \langle FU_2, S_2\rangle \\ \wedge & active(rc_1) \wedge active(rc_2) \wedge \langle FU_1, S_1\rangle = \langle FU_2, S_2\rangle \ \rightarrow \ i_1 = i_2\end{array}\right)\end{array}$$

The following invariant refers to the case that a record resides in register r (and is therefore visible). It identifies its instruction number as the instruction that has last stored a value in register F in a sequential execution which has just executed instruction $top - 1$.

$$\varphi_7\text{:} \quad RF[r] = \langle b, \langle i, \langle FU, S\rangle\rangle, v\rangle \quad \rightarrow \quad i = lw(top - 1, r)$$

7 Proving Refinement

In this section, we complete the proof that system TOMASULO refines system NON-DET. As observation functions, we take:

$$\begin{array}{lll}\mathcal{O}_A & : & reg \\ \mathcal{O}_C & : & \lambda r\colon [0..R].\ \textbf{if}\ top > N\ \textbf{then}\ RF[r].VALUE\ \textbf{else}\ 0\end{array}$$

Applying proof rule REF, we should present a refinement relation $\alpha(V_C, V_A)$, an initialization function $I(V_C)$, and a continuation function $\delta(V_C, V_C', V_A)$.

We start by presenting the refinement relation α. We present this relation as a conjunction of several assertions as follows:

$$\begin{array}{lrcl}
\alpha_1: & visible(\langle b, \langle i, \langle FU, S\rangle, V\rangle) & \rightarrow & c[i] \leftrightarrow \neg b \\
\alpha_2: & visible(\langle b, \langle i, \langle FU, S\rangle, V\rangle) & \rightarrow & v[i] = \textbf{if } b \textbf{ then } 0 \textbf{ else } V \\
\alpha_3: & \left(\bigwedge_{i=1}^{N} c[i]\right) & \leftrightarrow & top > N \wedge \neg\exists rc.\, active(rc)
\end{array}$$

Next, we present the initialization function I. This will be presented by writing expressions for the initial values of the system variables for system NON-DET, as follows:

$$\begin{array}{rcl}
N_I & = & N \\
prog_I & = & prog \\
c_I & = & \lambda i{:}\,[0..N].\ (i = 0) \\
v_I & = & \lambda i{:}\,[0..N].\ 0 \\
reg_I & = & \lambda r{:}\,[0..R].\ 0
\end{array}$$

Finally, we present the continuation function δ. This will be presented by writing expressions for the continuation values of the system variables for system NON-DET. Note that expressions can refer to the old system variables of NON-DET as well as to the old and new values of the system variables of TOMASULO.

$$\begin{array}{rcl}
N_\delta & = & N \\
prog_\delta & = & prog \\
c_\delta & = & \lambda i{:}\,[0..N].\ c[i] \ \vee\ \exists FU{:}\,[1..f] \left(\begin{array}{ll} \wedge & result[FU].rt.p.n = i \\ \wedge & result[FU].a \ \wedge\ \neg result'[FU].a \end{array}\right) \\
v_\delta & = & \lambda i{:}\,[0..N] \left(\begin{array}{ll} \textbf{if} & \neg c[i] \ \wedge\ c_\delta[i] \\ \textbf{then} & do_op(prog[i].op,\ gop(i,1),\ gop(i,2)) \\ \textbf{else} & v[i] \end{array}\right) \\
reg_\delta & = & \lambda r{:}\,[0..R].\ \textbf{if } top' > N \wedge \neg\exists rc.\, active'(rc) \textbf{ then } RF'[r].v \textbf{ else } 0
\end{array}$$

8 Conclusion

We have presented a refinement-based proof-method for the verification of modern processor architectures, and demonstrated its applicability by showing the correctness of a data-path involving multiple functional units, register-renaming, dynamic scheduling, and out-of-order execution, thus significantly exceeding the complexity of static pipelines as e.g. treated in [BD94]. Still, this work needs to be extended in a number of ways.

- So far, verification of invariants and the refinement relation within PVS [ORSS94] is not completed and relying on hand-translation rather than compilation of the employed notation for STS;
- Ongoing work considers to considerably lift the degree of automation achieved so far within PVS and providing links to VHDL based verification tools developed in the FORMAT project;
- A paper under preparation shows how to extend the framework of this paper to deal with load-store buffers in the context of shared memory machines with weak memory consistency models.

References

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. 6^{th} Intl. Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 68–80. Springer-Verlag, 1994.

[BM97] G. Barrett and A. McIsaac. Model-checking in a microprocessor design project. In O. Grumberg, editor, *Proc. 9^{th} Intl. Conference on Computer Aided Verification (CAV'97)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1997. to appear.

[CCH+96] Y.A. Chen, E.M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J.OLeary, and X. Zhao. Verification of all circuits in a floating point unit using word-level modelchecking. In M. Srivas and A. Camilleri, editors, *Proc. 1^{st} Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lect. Notes in Comp. Sci.*, pages 1–18. Springer-Verlag, 1996.

[DHP95] G. Doehmen, R. Herrmann, and H. Pargmann. Translating vhdl into functional symbolic finite-state models. *J. on Formal Methods in System Design*, 7:125–148, 1995.

[DJS94] W. Damm, B. Josko, and R. Schlör. Specification and verification of VHDL-based system-level hardware design. In E. Börger, editor, *Specification and Validation Methods*, pages 331–410. Oxford University Press, 1994.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[HIKB96] R. Hojati, A. Isles, D. Kirkpatrick, and R.K. Brayton. Verification using uninterpreted functions and finite instantiations. In *Proc. 1^{st} Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 218 – 232, 1996.

[HP96] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc, 1996.

[J.B96] J.Burch. Techniques for verifying superscaler microprocessors. In *Proc. Design Automation Conference 96 (DAC96)*, 1996.

[JDJ95] R.B. Jones, D.L. Dill, and J.R.Burch. Efficient validity checking for processor verification. In *Intl. Conf. on Computer-Aided Design*. IEEE, 1995.

[Keu96] K. Keutzer. The need for formal methods for integrated circuit design. In M. Srivas and A. Camilleri, editors, *Proc. 1^{st} Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lect. Notes in Comp. Sci.*, pages 1–18. Springer-Verlag, 1996.

[KP96] Y. Kesten and A. Pnueli. An αSTS-based common semantics for SIGNAL, STATECHART, DC+, and C. Technical report, Dept. of Comp. Sci., Weizmann Institute, March 1996.

[MP96] Z. Manna and A. Pnueli. Clocked transition systems. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering*, pages 3 – 42. World Scientific, Singapore, 1996.

[ORSS94] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. A tutorial on using PVS for hardware verification. In R. Kumar and T. Kropf, editors, *Proceedings of the Second Conference on Theorem Provers in Circuit Design*, pages 167–188. FZI Publication, Universität Karlsruhe, 1994. Preliminary Version.

[PS97] A. Pnueli and E. Singerman. Fair synchronous transition systems and their liveness proofs. Technical report, Dept. of Comp. Sci., Weizmann Institute, Jan 1997.

[SM96] M. Srivas and S. Miller. Applying formal verification to the aamp5 microprocessor: A case study in the industrial use of formal methods. *J. on Formal Methods in System Design*, 8:153–188, 1996.

system TOMASULO(R, N, U, Z: $\mathbb{N}^+$) **is**
import R/R_PROGRAM(R, N)
types

$$
\begin{array}{lcl}
\mathit{FU\_ID} & = & [1..U]; \\
\mathit{SLOT\_ID} & = & [1..Z]; \\
\mathit{HW\_NAME} & = & [f\colon [0..U],\ s\colon [0..Z]]; \\
\mathit{I\_NAME} & = & \mathbb{N}; \\
\mathit{PRODUCER} & = & [n\colon \mathit{I\_NAME},\ h\colon \mathit{HW\_NAME}]; \\
\mathit{BUSY} & = & \mathbf{boolean}; \\
\mathit{OCCUPIED} & = & \mathbf{boolean}; \\
\mathit{ACTIVE} & = & \mathbf{boolean}; \\
\mathit{R\_TYPE} & = & [b\colon \mathit{BUSY},\ p\colon \mathit{PRODUCER},\ v\colon \mathit{VALUE}]; \\
\mathit{SRC\_TYPE} & = & \mathbf{array}[1..2]\ \mathbf{of}\ \mathit{R\_TYPE}; \\
\mathit{S\_TYPE} & = & [\ o\colon \mathit{OCCUPIED},\ \ rt\colon \mathit{R\_TYPE},\ \ op\colon \mathit{OP\_TYPE}, \\
 & & \qquad\qquad t\colon \mathit{TARGET},\ \ ss\colon \mathit{SRC\_TYPE}\ \ ]
\end{array}
$$

variables

$$
\begin{array}{l}
RF\colon\ \mathbf{array}[0..R]\ \mathbf{of}\ \mathit{R\_TYPE}\ \mathbf{init}\ \langle \mathit{false}, \langle 0, \langle 0,0\rangle\rangle, 0\rangle; \\
rs\colon\ \mathbf{array}[1..U, 1..Z]\ \mathbf{of}\ \mathit{S\_TYPE}\ \mathbf{init} \\
\langle \mathit{false},\ \langle \mathit{false}, \langle 0, \langle 0,0\rangle\rangle, 0\rangle,\ \mathbf{fpadd},\ 0,\ \langle \mathit{false}, \langle 0, \langle 0,0\rangle\rangle, 0\rangle,\ \langle \mathit{false}, \langle 0, \langle 0,0\rangle\rangle, 0\rangle\rangle; \\
\mathit{result}\colon\ \mathbf{array}[1..U]\ \mathbf{of}\ [a\colon \mathit{ACTIVE},\ rt\colon \mathit{R\_TYPE},\ t\colon \mathit{TARGET}]\ \mathbf{init} \\
\qquad\qquad \langle \mathit{false},\ \langle \mathit{false}, \langle 0, \langle 0,0\rangle\rangle, 0\rangle,\ 0\rangle; \\
\mathit{top}\colon\ [1..N{+}1]\ \mathbf{init}\ 1
\end{array}
$$

definitions

$$
\begin{array}{lcl}
\mathit{master} & = & \mathbf{choose}\ FU\colon [1..U]\ \mathbf{s.t.}\ \mathit{result}[FU].a\,; \\
\mathit{bus\_active} & = & \mathit{result}[FU].a; \\
\mathit{cdb} & = & \mathit{result}[\mathit{master}]; \\
FU_n & = & \mathit{fu\_table}[\mathit{prog}[\min(\mathit{top}, N)].\mathit{op}]; \\
S_n & = & \mathbf{choose}\ S\colon [1..Z]\ \mathbf{s.t.}\ \neg rs[FU_n, S].o\,; \\
\mathit{dispatch} & = & \mathit{top} \le N\ \wedge\ \neg rs[FU_n, S_n].o
\end{array}
$$

behavior

$$
\textsc{dispatcher}\ \ \|\ \ \textsc{write\_back}\ \ \|\ \ \big\|_{FU=1}^{U}\ \textsc{fun}(FU)
$$

end system

Figure 6: System TOMASULO

system DISPATCHER **is**
definitions

$$
\begin{array}{lcl}
i & = & top; \\
\langle op, f, f_1, f_2 \rangle & = & prog[top]
\end{array}
$$

$$
\begin{array}{lcll}
src_1 & = & \textbf{if} & bus\_active \ \wedge \ RF[f_1].b \ \wedge \ RF[f_1].h = cdb.h \\
 & & \textbf{then} & \langle false, cdb.p, cdb.v \rangle \ \ \textbf{else} \ \ RF[f_1]; \\
src_2 & = & \textbf{if} & bus\_active \ \wedge \ RF[f_2].b \ \wedge \ RF[f_2].h = cdb.h \\
 & & \textbf{then} & \langle false, cdb.p, cdb.v \rangle \ \ \textbf{else} \ \ RF[f_2]
\end{array}
$$

behavior
if *dispatch* **then**

$$
\begin{array}{rcl}
rs[FU_n, S_n] & := & \left(\begin{array}{ll} \langle & true, \\ & \langle true, \langle top, \langle FU_n, S_n \rangle \rangle, 0 \rangle, \\ & prog[top].op, \\ & prog[top].t, \\ & \langle src_1, \ src_2 \rangle \qquad \rangle \ ; \end{array} \right) \\
RF[prog[top].t] & := & \langle true, \langle top, \langle FU_n, S_n \rangle \rangle, 0 \rangle \ ; \\
top & := & top + 1
\end{array}
$$

end if
end system

Figure 7: System DISPATCHER