Formal Verification of Complex Out-of-Order Pipelines by Combining Model-Checking and Theorem-Proving

Christian Jacobi*

IBM Deutschland Entwicklung GmbH, Processor Development II
71032 Boeblingen, Germany
cjacobi@de.ibm.com

Abstract. We describe a methodology for the formal verification of complex out-of-order pipelines as they may be used as execution units in out-of-order processors. The pipelines may process multiple instructions simultaneously, may have branches and cycles in the pipeline structure, may have variable latency, and may reorder instructions internally. The methodology combines model-checking for the verification of the pipeline control, and theorem proving for the verification of the pipeline functionality. In order to combine both techniques, we formally verify that the FairCTL operators defined in μ-calculus match their intended semantics expressed in a form where computation traces are explicit, since this form is better suited for theorem proving. This allows the formally safe translation of model-checked properties of the pipeline control into a theorem-proving friendly form, which is used for the verification of the overall correctness, including the functionality. As an example we prove the correctness of the pipeline of a multiplication/division floating point unit with all the features mentioned above.

1 Introduction

As microprocessor designs become increasingly complex, validation using traditional simulation becomes more and more insufficient to ensure the correctness of the design. Over the last years, formal methods have proved to be applicable to very complex systems such as out-of-order processors [10, 17, 14, 3, 13]. However, except for [13], these processors only contain very simple execution units. They can process only one instruction at a time, and their pipelines have a simple structure. Furthermore, the delay of the execution units is often assumed to be fixed. In contrast, modern execution units process multiple instructions simultaneously, may have branches and cycles in the pipeline structure (e.g., for iterative division algorithms), may have variable latency for each instruction, and may reorder instructions internally, i.e., instructions do not need to leave the pipeline in the order they entered it. In [13], a Tomasulo scheduler [19] has

* The work reported here was done while the author was affiliated with Saarland University.

been verified which is capable of using such execution units; however, neither the design nor the verification of the actual execution units is described in [13].

In this paper we describe a methodology for the verification of pipelined execution units with the features described above. As an example we describe the verification of the pipeline of a multiplication/division floating point unit, whose combinatorial datapaths have been verified in [4]. The pipeline can process up to six instructions simultaneously. The difficulty in the verification of such complex pipelines arises from the fact that pipelines consist of a control-dominated part which schedules the processing of the instructions in the pipeline, while simultaneously the effect of the datapaths on the data of each instruction has to be considered in order to guarantee *functional* correct behavior of the execution unit.

The use of theorem proving for the verification of complex pipelines would involve the construction of an inductive invariant to cope with the control-dominated part. The construction usually has to be performed manually, which is considered the hard part of the verification of out-of-order systems [10,17,13]. On the other hand, model-checking is suitable for the automatic verification of control-dominated systems, but becomes infeasible for the verification of complete pipelines due to the data part. Even if one uses abstract datapaths, e.g. uninterpreted functions [6], the state space grows huge due to the large number of (nested) function applications (e.g., due to possible cycles in the pipeline structure).

We propose a methodology which combines the best of both worlds: we use model-checking to verify the control part of the pipelines, and then use theorem proving to conclude overall correctness, including data correctness. We use the PVS theorem proving system [15] with its built-in model-checker [16].

In order to use model-checked properties for the further verification by theorem proving, the model-checked properties have to be translated into a form which is easy to use for theorem proving. In PVS, the FairCTL operators are defined as fixpoints in μ-calculus, which in turn are defined in terms of higher-order logic. These definitions are hard to use in theorem proving. It is more suitable for theorem proving to define computation traces explicitly, and to express temporal properties using standard mathematical quantifiers, e.g., $\forall t\colon p(t)$ to express a property p to hold for all times t. In order to translate model-checked properties safely from FairCTL to $\forall t$ form, we have proved theorems which relate the FairCTL operators defined in μ-calculus with their intended semantics expressed in $\forall t$ form. These relations are well known [7], but have not been verified using formal methods before.

The mathematics in this paper has been formalized and verified in PVS. For the sake of readability we use standard mathematical notation throughout the paper. All PVS specifications and proofs are available at our web site.[1]

Paper Outline. In the following section, we define the correctness criterion which the execution units shall obey. The correctness criterion is defined in terms of computation traces of a next-state function under a given input se-

[1] http://www-wjp.cs.uni-sb.de/projects/verification/{pvsctl,fpu}

quence. We then describe in section 3 how model-checked properties of a system can be translated into the computation trace form. In section 4 we show how model-checking and theorem proving is combined for the verification of complex pipelines. The discussion of related work is postponed to section 5. Section 6 gives a summary.

2 Pipeline Correctness Criterion

In this section we describe the correctness criterions which our execution units (EU, also called *pipelines* in this paper) shall obey. An execution unit can be seen as a black box with inputs and outputs interconnecting the EU with the Tomasulo scheduled processor core. The core *dispatches* instructions by passing the instruction data (operands, op-code, etc.) to the EU along with a tag used to identify the instruction. The EU executes the instruction and returns the result with the corresponding tag to the core. The EU may process several instructions simultaneously, instructions may have variable latency, and the EU may reorder instructions internally, i.e., instructions do not need to leave the pipeline in the order they have entered it. The Tomasulo scheduler from [13] can cope with these possibilities.

The Tomasulo scheduler only dispatches instruction whose operands are available. Therefore, the pipelines do not have to cope with *data hazards*. The only hazards occurring in the pipelines are *structural hazards*, i.e., multiple instructions requiring the same resources in the pipeline.

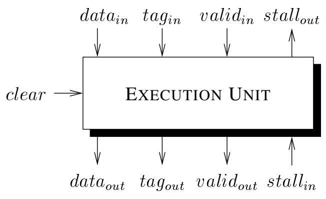

Fig. 1. Execution unit interface

Figure 1 shows a black-box view of an execution unit. The *clear* input is activated at power-up and during interupts in order to clear the pipeline. Instructions are dispatched into the EU by activating the $valid_{in}$ signal along with the instruction's $data_{in}$ and tag_{in}. The EU then computes the result and returns it by activating $valid_{out}$ along with the proper $data_{out}$ and tag_{out}. The $stall_{out}$ signal is activated if the EU cannot take further instructions; in this case, the scheduler must not dispatch instructions. Analogously, if the core activates the $stall_{in}$ signal, the EU must not return any instructions. In the following, we ignore the *clear* signal since the implementation and verification of *clear* is simple.

Formalization of the EU Interface. Let $\mathcal{S}$ denote the state set of the EU (usually the set of possible contents of the registers in the EU). Let $\mathcal{D}_i$, $\mathcal{D}_o$, and $\mathcal{T}$ denote the set of the input data, output data, and tags, respectively. The *valid* and *stall* signals are booleans. The EU is specified by the following five functions:

1. $ns(S_{cur}, data_{in}, tag_{in}, valid_{in}, stall_{in}) \rightarrow \mathcal{S}$: the next-state function, which computes the next state given the current state S_{cur} and the current inputs.
2. $data_{out}(S_{cur}, data_{in}, valid_{in}, stall_{in}) \rightarrow \mathcal{D}_o$: computes the data output of the EU given current state and inputs.

3. $tag_{out}(S_{cur}, tag_{in}, valid_{in}, stall_{in}) \to \mathcal{T}$: computes the output-tag.
4. $valid_{out}(S_{cur}, valid_{in}, stall_{in}) \to \mathbb{B}$: computes the valid output.
5. $stall_{out}(S_{cur}, stall_{in}) \to \mathbb{B}$: computes the stall output.

The functions $data_{out}$, tag_{out}, $valid_{out}$, and $stall_{out}$ model the combinatorial circuits computing the corresponding outputs from the (registered) state and the current inputs. Note that not all outputs may depend on all inputs. This is necessary to model absence of *combinatorial* dependencies between some inputs and outputs. For example, $stall_{out}$ only depends on the state and the current $stall_{in}$, i.e., whether the EU accepts a further instruction may not depend on the instruction data or tag.

Let $\mathcal{I} := \mathcal{D}_i \times \mathcal{T} \times \mathbb{B} \times \mathbb{B}$ denote the combination of the inputs of the EU. We recursively define the behavior of a pipeline under an infinite input sequence $I := (i_0, i_1, \ldots) \in \mathcal{I}^\infty$. We assume the pipeline to be in some initial state $init \in \mathcal{S}$ at time $t = 0$. The state $s^t(I)$ at time t is recursively defined as

$$s^0(I) := init, \;\; s^{t+1}(I) := ns(s^t(I), i_t).$$

We define $data^t_{out}(I)$, $tag^t_{out}(I)$, $valid^t_{out}(I)$, and $stall^t_{out}(I)$ to be the outputs of the pipeline during cycle t, e.g., $stall^t_{out}(I) := stall_{out}(s^t(I), i_t.stall_{in})$. For the sake of convenience, we omit the parameter I if it is clear from the context.

We say a tag $tg \in \mathcal{T}$ is dispatched at time t (denoted by $disp(tg, t)$), if $valid^t_{in}$ and $tag^t_{in} = tg$ hold. The tag is returned at time t (denoted by $ret(tg, t)$), if $valid^t_{out}$ and $tag^t_{out} = tg$ hold. The tag is in use at time t (denoted by $inuse(tg, t)$), if the tag was dispatched and not yet returned, i.e.,

$$inuse(tg, t) := \exists t' < t \colon disp(tg, t') \text{ and } \forall t'' \in \{t', \ldots, t-1\} \colon \neg ret(tg, t'').$$

Correctness Criterion. We can now define the correctness criterions for execution units. A $valid_{out}$ may only be signaled if $stall_{in}$ is not active:

$$\forall t \colon stall^t_{in} \implies \neg valid^t_{out}. \tag{P1}$$

The $stall_{out}$ signal is live, i.e., at each point in time t, it will eventually become inactive (at time t'):

$$\forall t \colon \exists t' \geq t \colon \neg stall^{t'}_{out}. \tag{P2}$$

Instructions dispatched into the EU at time t will eventually be returned (at time t'). We call this property *liveness* of the EU.

$$\forall t \colon disp(tg, t) \implies \exists t' \geq t \colon ret(tg, t'). \tag{P3}$$

The last property, called *tag-consistency*, states that instructions returned at time t by the EU have already been dispatched before (at time t'), and have not already been returned in between (at time t''):

$$\begin{aligned}\forall t \colon ret(tg, t) \implies \exists t' \leq t \colon disp(tg, t') \text{ and}\\ \forall t'' \in \{t', \ldots, t-1\} \colon \neg ret(tg, t'').\end{aligned} \tag{P4}$$

Note that the right side of the above definition does not exactly match $inuse(tg, t)$, since here $t' = t$ is allowed. However, it is sufficient to prove $\forall t\colon ret(tg,t) \implies inuse(tg,t)$ in order to assert tag-consistency. Note further that liveness and tag-consistency together yield a one-to-one mapping between dispatched and returned instructions.

Of course the execution unit cannot satisfy these properties if the input sequence does not satisfy some properties itself. The first required input property is that no instruction is dispatched if the $stall_{out}$ is active, analogously to (P1):

$$\forall t\colon stall_{out}^{t} \implies \neg valid_{in}^{t}. \tag{I1}$$

The analogue to (P2) is that the $stall_{in}$ signal is live:

$$\forall t\colon \exists t' \geq t\colon \neg stall_{in}^{t'}. \tag{I2}$$

The third input property is called *tag-uniqueness* and requires that no tag tg is dispatched into the EU if it is already in use:

$$\forall t\colon disp(tg,t) \implies \neg inuse(tg,t) \tag{I3}$$

We call an execution unit correct iff for all input sequences I and tags tg the properties (P1) to (P4) hold under the assumptions (I1) to (I3), where not all properties need all assumptions:

$$\begin{aligned} EUcorrect := {} & (I1) \implies (P1) \text{ and} \\ & (I1) \wedge (I2) \implies (P2) \wedge (P3) \text{ and} \\ & (I1) \wedge (I2) \wedge (I3) \implies (P4). \end{aligned} \tag{C}$$

This definition of correctness only covers the correct termination of instructions. In order to cover the input/output data relation, we introduce the notion of *functional correct execution units*. An EU is called *functional correct* with respect to a function $dp : \mathcal{D}_i \to \mathcal{D}_o$, iff $dp(data_{in}) = data_{out}$ holds for corresponding inputs and outputs. For example, a floating point unit can be described by a function dp reflecting the combinatorial datapaths, and the pipelined hardware shall compute this function. In order to model functional correctness, we strengthen the liveness property (P3) to cover the relation between data input and output of an instruction:

$$\forall t\colon disp(tg,t) \implies \big(\exists t' \geq t\colon ret(tg,t') \text{ and } dp(data_{in}^{t}) = data_{out}^{t'}\big). \tag{P3$'$}$$

Formally, we call an execution unit *functional correct* with respect to dp iff (C) holds where (P3) is replaced by (P3$'$).

Note that the definition of (functional) correctness allows multiple instructions (with distinct tags) in the EU simultaneously, and that no restriction on the order in which instructions leave the EU is imposed. Note further that not all EUs have a functional description; a memory unit, e.g., cannot be described by a function dp, since functions are by definition memory-less.

The correctness criterions of the EUs have been arranged with Kröning in order to allow the integration of our EUs into Kröning's Tomasulo core [13].

Example Pipeline. In [4], the verification of the combinatorial datapaths of an IEEE compliant floating point unit (FPU) is reported. Here, we aim at verifying the pipeline of this FPU as an example of our verification approach. Pipelining is not considered in [4]. Figure 2 shows the structure of the pipeline of the multiplication/division floating point unit.

The first pipeline stage performs unpacking of floating point operands, handles special cases (e.g., operations on $\pm\infty$), and initial approximation lookup in case of division. The next two stages comprise a pipelined multiplier. For division, the instructions have to iterate through these stages up to 8 times, depending on the precision of the floating point operation. The *selfd* stage is used for divisions only, multiplications skip this stage. Finally, the results are rounded by a two-stage rounder. Special cases do not flow through the pipeline, but are bypassed from the unpacker to the output.

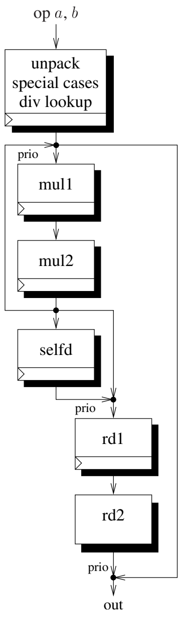

Fig. 2. FPU pipeline

Each instruction flows through the pipeline until it cannot flow further due to structural hazards, i.e., other instructions in the pipeline require the same resources. For example, if two divisions are iterating simultaniously through the two multiplication stages, a multiplication in the unpack stage has to be stalled. Out-of-order completion in this pipeline can occur in various ways: for example, an operation involving special cases is bypassed to the output while other operations are still in the pipeline. Another example is a multiplication which overtakes a division that iterates through the *mul1* and *mul2* stages.

The FPU from [4] is given as a function md. We have partitioned the computation of this function into sub-functions corresponding to the datapaths of the individual pipeline stages, e.g, functions unp, $mul1$, ... For multiplications on non-special operands $md = rd2 \circ rd1 \circ mul2 \circ mul1 \circ unp$ holds, i.e., multiplication can be performed by consecutive execution of the pipeline stage functions. Analogously, for non-special divisions $md = rd2 \circ rd1 \circ selfd \circ (mul2 \circ mul1)^i \circ unp$ holds, where i is the number of iterations depending on the precision. For the verification of the pipeline, the actual implementation of the datapaths is not important, i.e., the functions can be left uninterpreted. We only have to prove that instructions take the correct path through this pipeline, and that the correct stage functions are applied to the instruction data.

3 Translating FairCTL to $\forall t$ Form

Our goal is to use the PVS built-in model-checker for the verification of temporal properties of the pipeline control, and then to use the theorem prover to conclude overall correctness of the pipeline, including the datapaths. In PVS, the FairCTL operators are defined as fixpoint in μ-calculus [16], whereas we have used temporal properties in $\forall t$ form in the previous section. In order to transform model-checked statements from FairCTL to $\forall t$ form, we formally verify that the FairCTL operators defined as fixpoints in μ-calculus match their intended semantics expressed in $\forall t$ form. These theorems have first been proved in [8] and are well known. However, they have not been verified using formal methods, which is necessary to transform between μ-calculus and $\forall t$ form in a formally safe way. The formal verification depends on the definition of fixpoints and FairCTL operators in the PVS library, and on the Tarski-Knaster argument, which has been verified in PVS in [16]. We omit the proofs in this section, since they follow the very detailed "paper & pencil" proofs from [7].

In this section, systems are described by a state set $\mathcal{S}$ and a total next-state *relation* $N \subseteq \mathcal{S} \times \mathcal{S}$ which models a non-deterministic choice of the next state. In contrast, in the previous section systems were modeled by next state *functions* which deterministically compute the next state from the current state and some inputs. It is easy to transform between the two kinds of systems by "simulating" inputs by non-deterministic choice and vice versa. We come back to this difference at the end of this section.

Let $f \in 2^{\mathcal{S}}$ be a predicate on $\mathcal{S}$, and let ν denote the greatest fixpoint operator. In PVS, the **EX** and **EG** operators, for example, are defined as predicates

$$\begin{aligned}\mathbf{EX}(N,f) &:= \lambda s \in \mathcal{S}\colon \exists s' \in \mathcal{S}\colon\ f(s') \wedge N(s,s'),\\ \mathbf{EG}(N,f) &:= \nu(\lambda Q \in 2^{\mathcal{S}}\colon\ f \wedge \mathbf{EX}(N,Q)).\end{aligned}$$

An *N-path* is an infinite sequence $(p_0, p_1 \ldots) \in \mathcal{S}^\infty$ where successive states respect the next-state relation, i.e., $\forall t\colon N(p_t, p_{t+1})$ holds. We have proved the following theorem:

Theorem 1. $\mathbf{EG}(N,f)(s)$ *iff there exists an N-path* $p_0, p_1, \ldots$ *starting in* s*, i.e.* $p_0 = s$*, where all states satisfy* f*, i.e.,* $\forall t\colon f(p_t)$*.*

We omit the definitions and theorems or the other FairCTL operators due to lack of space. Instead, we restate the theorems for the **AG** and **fairAF** operators with respect to the semantics of deterministic systems with input sequences below.

Non-Determinism versus Input Sequences. As mentioned above, FairCTL is defined in the context of non-deterministic systems without inputs, whereas deterministic systems with inputs have been used in the previous section to define the correctness of execution units. The use of deterministic next state functions is better suited for the definition of execution units since it is closer to the actual implementation; furthermore, we believe it is simpler to handle in theorem proving. However, the definition of FairCTL in PVS imposes the use of non-deterministic systems for model-checking. It is easy to bridge this gap:

Let $\mathcal{S}$ be the state type, $\mathcal{I}$ be the input type, and $ns : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$ be the deterministic next-state function of a system as in section 2. Further, let $Ip \subseteq \mathcal{S} \times \mathcal{I}$ be an input predicate (e.g., $Ip \equiv stall_{out} \implies \neg valid_{in}$ to model the pipeline input property (I1)). Let $init \in \mathcal{S}$ be the initial state. We define a new state type $\mathcal{S}' := \mathcal{S} \times \mathcal{I}$ and a non-deterministic next-state relation $N \subseteq \mathcal{S}' \times \mathcal{S}'$ by

$$N\left((s_1, i_1), (s_2, i_2)\right) := \left(s_2 = ns(s_1, i_1) \wedge Ip(s_2, i_2)\right).$$

Read the new state type as current state and input. Then there is a transition from (s_1, i_1) to (s_2, i_2), iff the next-state function ns takes the transition $s_1 \to s_2$ under input i_1. Furthermore, the next-state relation N non-deterministically chooses the next input i_2, which has to satisfy the input-predicate Ip. We define $init' := \{(s, i) \mid s = init \wedge Ip(s, i)\}$ as the initial state set of the new system.

We now state the theorems for the **AG** and **fairAF** operators with respect to deterministic systems:

Theorem 2. $(\forall s' \in init' \colon \mathbf{AG}(N, f)(s'))$ *iff for all input sequences* $I = (i_0, i_1, \ldots) \in \mathcal{I}^\infty$ *satisfying the input predicate, the predicate* f *holds globally:*

$$\left(\forall t \colon Ip(s^t(I), i_t)\right) \implies \left(\forall t \colon f(s^t(I))\right),$$

where s^t *is defined as in section 2.*

Theorem 3. *Let fair be a predicate.* $(\forall s' \in init' \colon \mathbf{fairAF}(N, f)(fair)(s'))$ *iff for all input sequences* $I := (i_0, i_1, \ldots) \in \mathcal{I}^\infty$ *satisfying the input predicate and yielding a path on which fair holds infinitly often, the predicate* f *holds eventually. Formally: for all input sequences* I *holds*

$$\left(\left(\forall t \colon Ip(s^t(I), i_t)\right) \wedge \left(\forall t \colon \exists t' \geq t \colon fair(s^t(I))\right)\right) \implies \left(\exists t \colon f(s^t(I))\right).$$

In the following, we do not explicitly distinguish between systems stated as next-state function or relation. Of course, one has to deal with the differences in PVS, but for reasons of readability we omit this in the rest of this paper.

4 Pipeline Verification

4.1 Separating Pipeline Control and Datapaths

In order to use model-checking on the pipeline control we have to separate the control and datapath circuits in the pipeline. Figure 3 shows a simple pipeline example. The control registers consist of valid bits indicating that a stage contains a valid instruction, the tags, and some auxiliary control data, e.g., a counter to keep track of the number of iterations to go through during divisions. The control circuit maintains the control registers, and computes the control outputs $valid_{out}, tag_{out}$, and $stall_{out}$.

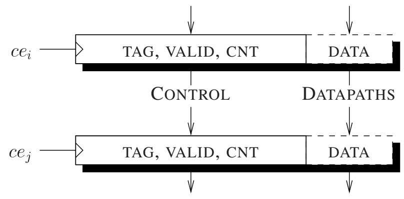

Fig. 3. Separating Control and Datapaths

The control interacts with the datapaths by computing the clock-enables *ce* for each stage and the multiplexer control signals where multiple inputs lead to the same pipeline stage (e.g, to the *mul1* stage in Fig. 2). The clock-enables control whether the register keeps its data from the previous cycle, or if new data is clocked into the register. If a stage i contains no valid instruction, it is always clocked ($ce_i = 1$), i.e., a potentially valid instruction is taken over from the preceeding stage. Otherwise, if stage i contains a valid instruction, it is only clocked if itself can pass its instruction to the succeeding stage j. This may not be possible due to several reasons: 1) the stage j may itself contain a valid instruction which it is unable to pass to the next stage. 2) there are multiple valid instructions aiming for stage j, and the instruction in stage i has lower priority. For instance, this may occur above stage $rd1$ in the FPU pipeline. 3) the instruction result has to be returned to the CPU from stage i, but the CPU has asserted the $stall_{in}$ signal. We refer the reader to [12] for details on the pipeline control.

According to the separation of control and data in the pipeline, we split the next-state function ns of the pipeline into a next-state function ns_{ctrl} of the control part, and a next-state function ns_{data} of the data part.

4.2 Verification of the Pipeline

In the following, we describe how we verify the liveness (P3) and tag-consistency (P4) properties of pipelines. We will not discuss the (P1) and (P2) properties, since these are fairly simple in comparison. Furthermore, we will only give the idea of the actual verification, since the mathematical details are tedious and straightforward.

Liveness. We start with the verification of liveness. In order to prove *functional* correctness of the pipelines, we will prove the strengthened liveness (P3′) covering the functionality of the pipeline. The verification idea is as follows: we first use model-checking to show that each pipeline stage is live, i.e., that its clock-enable becomes eventually active. We then use theorem-proving to show that the instruction take the correct path through the pipeline and hence the correct result is computed.

For model-checking the liveness of each of the clock-enables, the liveness of $stall_{in}$ is presumed. We model-check the following property for each stage i and an arbitrary, not necessarily initial control state s:

$$\mathbf{fair AF}(ns_{ctrl}, ce_i)(\neg stall_{in})(s).$$

Using theorem 3 we conclude that the clock-enable ce_i is live in all computations starting in arbitrary states s under all input sequences where $stall_{in}$ is live, i.e., for all $I := (i_0, i_1, \ldots)$:

$$\left(\forall t\colon \exists t' \geq t\colon \neg stall_{in}^{t'}\right) \implies \left(\exists t\colon ce_i^t\right). \tag{1}$$

Using theorem proving, it is easy to extend 1 to 2 by exploiting that (1) holds for arbitrary states:

$$\left(\forall t\colon \exists t' \geq t\colon \neg stall_{in}^{t'}\right) \implies \left(\forall t\colon \exists t' \geq t\colon ce_i^{t'}\right). \tag{2}$$

Note that the left-hand side of the equation matches the pipeline input property (I2).

Having proved the liveness of the clock-enables, it is relatively easy to verify liveness of the complete pipeline including the datapaths by pushing instructions through the pipeline stage by stage. This is done using theorem proving. We exemplarily prove the liveness property (P3′) of the multiplicative FPU for multiplication instructions:

Theorem 4. *Assume that the input properties (I1) and (I2) hold. Assume further that a multiplication with tag* tg *is dispatched at time* t*, i.e.,* $disp(tg, t)$ *holds. Then there exists* $t' \geq t$ *such that* $ret(tg, t')$ *and* $data_{out}^{t'} = rd2 \circ rd1 \circ mul2 \circ mul1 \circ unp(data_{in}^t)$ *hold, i.e., the multiplication eventually terminates with the correct data.*

Proof. We only sketch the proof, because its details are long and tedious. By input property (I1) we know that $stall_{out}^t$ is inactive, since otherwise the instruction cannot be dispatched. Since the definition of $stall_{out}$ directly depends on ce_{unp} (cf. [12, App. D]), one trivially concludes that the instruction is clocked into the register stage unp at time t. The data in this register are the outputs of the combinatorial unpack circuit.

From (2) we know that there exists a (minimal) time $t_1 > t$ such that $ce_{unp}^{t_1}$ is active, i.e., the unp stage is clocked at time t_1, and is not clocked in between. Hence, the data at time $t_1 - 1$ in the register stage unp is the same as at time t.

The unp stage is only clocked if its valid instruction proceeds to the next stage (this follows trivially from the definition of ce_{unp}^t). We conclude that the instruction with tag tg is clocked from the unp stage into stage $mul1$ at time t_1. The data at this time is computed from $mul1 \circ unp$, i.e., the composition of the first two combinatorial stages.

Analogously, we derive times $t_2 > t_1$, $t_3 > t_2$, and $t_4 > t_3$ where the instruction proceeds to $mul2$, $rd1$, $rd2$, respectively. When the instruction is in stage RD2, it is returned to the CPU immediately when the $stall_{in}$ signal becomes inactive. Hence, there exists $t' > t$ where the instruction is returned with $data_{out}^{t'}$ computed from $data_{in}^t$ by the combinatorial circuits between the register stages.

Note that the actual computation performed in the datapaths plays no role in the above proof, and hence the datapath functions may be left uninterpreted. □

Tag-Consistency. We now describe the verification of tag-consistency. We want to express tag-consistency (P4) in FairCTL in order to allow model-checking. Therefore we need a FairCTL formalization of "tag has been dispatched previously", and a formalization of tag-uniqueness. It would be useful to have temporal operators reaching in the past; however, FairCTL does not provide such operators. In order to circumvent this problem, we introduce an auxiliary variable $inuse_{tg}$ for each tag $tg \in \mathcal{T}$ representing that an instruction with tag tg is currently in the pipeline. The meaning of this variable is exactly the same as the predicate $inuse$ from section 2. The variable $inuse_{tg}$ is set whenever an instruction with tag tg enters the pipeline, and it is cleared whenever the tag tg leaves the pipeline. Tag-uniqueness can hence be modeled as input predicate Ip checking that the tag tg is not dispatched when the variable $inuse_{tg}$ is already set. Vice versa, tag-consistency can be modeled as an invariant stating that a tag tg can only leave the pipeline if $inuse_{tg}$ is set.

Let $\tilde{ns}_{ctrl}$ denote the next-state function of the modified model including the $inuse$ variables, and let Ip denote the input predicate modeling tag-uniqueness (I3). We verify the property

$$\forall tg\colon \mathbf{AG}\big(\tilde{ns}_{ctrl}, (valid_{out} \wedge tag_{out} = tg) \implies inuse_{tg}\big)(init),$$

where $init$ is an initial state in which all pipeline stages are empty (i.e., $valid_i = 0$), and all $inuse_{tg}$ variables are cleared. From this we conclude using theorem 2: for all input sequences $I = (i_0, i_1, \ldots) \in \mathcal{I}^\infty$ and for all tags tg

$$\big(\forall t\colon \ (valid_{in}^t \wedge tag_{in}^t = tg) \implies \neg inuse_{tg}^t\big) \implies$$
$$\big(\forall t\colon \ (valid_{out}^t \wedge tag_{out}^t = tg) \implies inuse_{tg}^t\big)\,.$$

One can see (and easily verify in PVS) that the left-hand side of the implication matches tag-uniqueness, and that the right-hand side implies tag-consistency.

4.3 Some Practical Considerations

In order to verify tag-consistency, we have changed the model and added the auxiliary variables $inuse_{tg}$. It is easy to prove that these auxiliary variables do not affect the outputs of the actual pipeline implementation and hence can be omitted. They are used solely to prove the correctness of the pipeline.

The state-space for model-checking becomes very large due to the tags and the $inuse_{tg}$ variables. Of course, one can abstract the tags by means of scalarsets [11] in the sense of data-type reduction as in SMV [14]. Model-checkers such as SMV support this as a built-in feature. In PVS the abstraction has to be done manually. We have abstracted the tags and proved the correctness of this abstraction, but omit the details since they are well known.

A major disadvantage of the PVS model-checker is that it is not capable of providing counter-examples when the verification of a FairCTL formula fails. Since the design of complex pipelines is very error-prone and debugging is hard, such counter-examples are very useful. We therefore developed and debugged the pipelines (without datapaths) in SMV, and then manually translated the pipeline control to PVS. We then used the PVS model-checker to re-check the properties.

We have manually performed the “pushing through the pipeline” in theorem 4 stage by stage during liveness verification. The proofs for each stage are very similar. We therefore believe that it is possible to create a proof strategy which performs the “pushing through the pipeline” automatically. This would result in a mostly automatic method for the verification of complex pipelines.

5 Related Work

There are some papers which report on the verification of out-of-order processors, e.g., by Hosabettu et.al. [10], by Sawada and Hunt [17], by McMillan [14], and by Berezin et.al. [3]. None of the cited papers mentions multi-cycle execution units, or even execution units which have a cycle in the pipeline structure or may reorder instructions internally. Kröning is the first who reports on the verification of a Tomasulo scheduler capable of handling such complex pipelines [13], although the design and the verification of the actual pipelines is not part of Kröning’s work. In this paper we have presented a methodology to verify complex pipelines, and have presented the pipeline of a multiplication/division floating point unit as an example. Kröning is currently integrating this example (among other pipelined FPUs for other operations) into his Tomasulo CPU.

Aagaard and Leeser [2] propose a methodology for the verification of pipelines: they decompose pipelines into segments, and then further decompose the correctness proof of individual segments into smaller proof goals. Their work describes only how one *could* employ a theorem prover for the verification of pipelines, but they do not actually use formal methods (in the sense of a computer tool). We have tried a similar approach to the verification of our pipelines using solely theorem proving, but failed because very complex inductive invariants had to be constructed manually [12].

Another approach to the verification of pipelines is the use of a logic with uninterpreted functions that are used to model the datapath functionality. The use of uninterpreted functions is comparable to the separation of the EU into pipeline control and datapaths, since the actual datapath implementation has no impact on the pipeline verification (cf. sect. 4). Bryant et.al. [5] describe how a logic with equality and uninterpreted functions can be reduced to propositional logic. In [20], Velev and Bryant describe how this reduction can be used to verify in-order microprocessors with variable-latency EUs. They do not verify the actual EU, but use an abstract execution unit model in order to verify the processor core. The EUs modeled by the abstraction process only one instruction at a time, and hence do not reorder instructions internally. Velev and Bryant only verify in-order processors; the verification of out-of-order designs would probably require the manual construction of a complex inductive invariant, and hence automation would be lost. In our approach, this is not the case due to the use of model-checking.

Another approach is the use of uninterpreted functions within a model-checker such as SMV. Data-type reduction and case-splitting is used in order to reduce the state space [14]. This is used in [14] to verify a Tomasulo scheduler, where the functionality of the EUs is defined by uninterpreted functions. The state space and the number of cases to be checked grows rapidly in the number of function applications, which is large in our example due to the cycle in the pipeline structure. We have modeled the FPU pipeline in SMV with uninterpreted functions for the datapaths, and have tried to verify liveness with functional correctness using model-checking. This was infeasible due to the huge state space and number of cases. The verification of some cases aborted with a memory usage of >2GB, other cases ran for more than 5 days without terminating.

In [3], Berezin et.al. prove the correctness of a simple Tomasulo processor by combining model-checking with uninterpreted functions and theorem proving. They use SMV to verify an invariant of an abstraction of the processor, and then use PVS to conclude overall correctness of the concrete machine. Their translation from SMV to PVS is not formally safe in the sense that they introduce a new, manually written axiom in PVS which hopefully reflects exactly the model-checked property. In contrast, we use the PVS built-in model-checker, and then use the theorems from section 3 to safely translate the model-checked properties to a form suitable for theorem proving.

In [9], Ho et.al. use the abstraction of the datapaths of pipelines to token nets for the automatic verification of pipeline control properties. Their approach is not applicable to pipelines with cycles in the pipeline structure, and is not suitable to verify functional correctness of the pipelines.

In [1], Aagaard et.al. verify iterative circuits using Intel's Forte system. They use symbolic simulation and LTL model-checking for the verification of bit-level invariants of iterative floating point circuits, and then use theorem proving to conclude "numerical" correctness of the floating point results. Though Intel's circuits are most probably much more complex than ours in terms of gate count, the verified pipelines are simple in the sense that they seem to support only one instruction at a time and hence do not reorder instructions. Moreover, the work from [1] is not reproducible since Intel's Forte system is not publicly available.

Schneider and Hoffmann [18] report on the definition of LTL in the theorem prover HOL, and on the automatic translation of LTL to ω-automata within HOL. The ω-automata are used as input for a model-checker. Their definition of LTL is close to our $\forall t$ form. Hence, their work could be used to verify pipelines in HOL in a similar way as described here.

6 Summary

We have presented a methodology for the verification of complex pipelines. The pipelines may process several instructions simultaneously, may have variable latency, cycles and branches in the pipeline structure, and may reorder instructions internally. The pipelines are used as execution units in the Tomasulo scheduler verified by Kröning [13]. As an example, we have presented the pipeline of a floating point unit, whose combinatorial correctness has been proved in [4].

Verification of the pipelines using solely theorem proving is hard since one has to manually construct a complex inductive invariant. The verification of the pipelines using solely model-checking is infeasible due to the large state space which arises from the datapaths, even if these are modeled as uninterpreted functions (cf. sect. 5). We therefore combine model-checking and theorem proving for the verification of the pipelines. Model-checking is used to verify properties of the pipeline control, theorem proving is then used to conclude overall correctness of the pipeline including the datapaths.

The correctness criterions for the execution units are given as temporal properties of the form $\forall t\colon p(t)$ (cf. sect. 2), which is suitable for theorem proving. In contrast, the FairCTL operators used for model-checking are defined as fixpoints in μ-calculus. We therefore have formally proved that the FairCTL operators, as defined in μ-calculus, match their intended semantic expressed in $\forall t$ form. This has been shown previously with "paper & pencil" proofs [8], but it has never been proved using formal methods before. Having proved the correspondence of the FairCTL operators expressed in μ-calculus and $\forall t$ form allows us to translate between both languages in a formally safe way. This is necessary to prevent errors which may be introduced by translating properties between two systems or languages by hand.

Acknowledgements

The author would like to thank Michael Backes, Christoph Berg, Sven Beyer, Daniel Kröning, Wolfgang Paul, and Jochen Preiß for valuable discussions.

References

1. M. Aagaard, R. B. Jones, R. Kaivola, K. R. Kohatsu, and C.-J. H. Seger. Formal verification of iterative algorithms in microprocessors. In *DAC-00*. ACM/IEEE, 2000. 321
2. M. Aagaard and M. Leeser. Reasoning about pipelines with structural hazards. In *TPCD'94*, volume 901 of *LNCS*. Springer, 1994. 320
3. S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *FMCAD '98*, LNCS 1522. Springer, 1998. 309, 320, 321
4. C. Berg and C. Jacobi. Formal verification of the vamp floating point unit. In *CHARME 2001*, LNCS 2144. Springer, 2001. 310, 314, 322
5. R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. on Computational. Logic (TOCL)*, 2(1):1–41, Jan 2001. 320
6. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*, LNCS 818. Springer, 1994. 310
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999. 310, 315

8. E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, LNCS 85. Springer, 1980. 315, 322
9. P.-H. Ho, A. J. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *ICCAD-98*. ACM, 1998. 321
10. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying microarchitectures that support speculation and exceptions. In *CAV '00*, volume 1855 of *LNCS*. Springer, 2000. 309, 310, 320
11. C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996. 319
12. C. Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. PhD thesis, Saarland University, Germany, 2002. handed in; draft available at `www-wjp.cs.uni-sb.de/~cj/phd.ps`. 317, 318, 320
13. D. Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001. 309, 310, 311, 314, 320, 322
14. K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000. 309, 319, 320, 321
15. S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *CADE 11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992. 310
16. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *CAV'95*, volume 939. Springer, 1995. 310, 315
17. J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In *CAV '98*, volume 1427 of *LNCS*. Springer, 1998. 309, 310, 320
18. K. Schneider and D. W. Hoffmann. A HOL conversion for translating linear time temporal logic to ω-automata. In *TPHOL 99*, volume 1690 of *LNCS*. Springer, 1999. 321
19. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11 (1), pages 25–33. IBM, 1967. 309
20. M. N. Velev and R. E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exception, and branch prediction. In *DAC '00*. ACM/IEEE, 2000. 320