Achieving Maximum Performance: A Method for the Verification of Interlocked Pipeline Control Logic

Kerstin Eder*
University of Bristol
Department of Computer Science
Woodland Road, MVB
Bristol BS8 1UB, GB
eder@cs.bris.ac.uk

Geoff Barrett
Broadcom DSL BU
320 Bristol Business Park
Coldharbour Lane
Bristol BS16 1EJ, GB
gbarrett@broadcom.com

ABSTRACT

Getting the interlock logic which controls pipeline flow correct is an important prerequisite for maximising pipeline performance. Unnecessary pipeline stalls can only be eliminated when they can be distinguished from those stalls which are necessary to preserve functional correctness.

We propose a method for deriving a maximum pipeline performance specification from a complete functional specification of the pipeline control logic. The performance specification can be used to generate simulation testbench assertions. On the other hand, the specification can serve as a basis for formal property checking. The most promising aspect of our work is, however, the potential to synthesise the actual control logic from its formal description.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*; B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Control Design, Pipeline*

General Terms

Performance, Verification

Keywords

Pipeline Stall, Interlock Logic, Verification

1. INTRODUCTION

1.1 Motivation

Unnecessary pipeline stalls decrease processor performance. Ensuring that a pipelined microprocessor utilises its maximum pipeline throughput is therefore an important task for processor designers. The complexity of today's microprocessor architectures demands a systematic and rigorous approach; one that can be provided by applying formal methods.

This paper describes the method developed while verifying the performance of the FirePath processor. FirePath [9] is a Broadcom proprietary LIW/SIMD processor targeted at communications and multimedia applications, its first use being embedded within a DSL system chip. The processor is based on a pipelined dual instruction multiple data microarchitecture. Pipeline flow is controlled via an interlock logic that prevents functional hazards. To achieve best performance, the pipeline should only stall when this is required to guarantee functional correctness. For example, competition for a completion bus results in pipeline congestion, thus potentially stalling all final pipeline stages, which did not win the grant for the completion bus. Stalls in a completion pipeline stage can cause preceding stages to stall.

From a performance point of view, the instructions in the pipeline should move into the next pipeline stage whenever possible. Hence, a performance bug is a pipeline stall for which there is no functional justification. Maximum performance can be achieved if there are no unnecessary stalls. With our method we can derive a maximum performance specification from a specification of the control logic which describes all functional constraints that cause a pipeline stall by applying fixed-point iteration. This performance specification can be included in a verification testbench in the form of an assertion. Alternatively, the specification can be used as a basis for formal property checking.

Our method is applicable to any pipelined microprocessor architecture that uses interlock logic to control pipeline flow, provided the functional specification of the flow control logic has the properties described in Section 3.1. These properties are by no means restrictive, in fact they are what one would naturally expect when specifying pipeline control flow.

A very promising aspect of our approach is that the functional specification can serve as the basis for RTL code synthesis. We are confident that the basic HDL code for the interlock logic can be generated fully automatically from its functional specification. This is one topic of our current research.

This paper introduces our method by means of a case study in Section 2. For this purpose and to focus our attention on the approach (rather than the architecture itself) we introduce a simple pipelined processor architecture in Section 2.1. The performance specification of our example architecture will be developed from its functional specification in Section 2.2. Section 3 establishes the theoretical soundness of our approach and outlines assumptions

*This publication is one result of work that was performed during a collaborative summer project between the University of Bristol and Broadcom Corporation. Kerstin Eder is grateful for the support of both organisations during this time.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

that need to be satisfied to apply the method. Our results are presented in Section 4, which is followed by some conclusions and an outlook on further work in Section 5.

1.2 Related Work

Approaches to verify the correctness of pipelined microprocessor designs range from traditional simulation-based verification via automatic property or model checking [4, 2] to semi-automatic theorem proving [1]. Combined approaches have also been successfully applied [3].

In practice, simulation-based verification is still widely used to gain confidence in design correctness; this also applies to pipeline flow control. Kohno and Matsumoto describe in [5] a test program generation tool which supports the verification of pipeline design. Their approach requires a pipeline behaviour specification to be provided. Based on this specification their tool automatically generates test cases. The pipeline behaviour specification contains pipeline definitions comprising pipeline stages, corresponding data hazards, bypassing mechanisms and resource usage for each instruction category. Any additional stall conditions are specified in explicit stall definitions. Compared to the specification used in our approach, their pipeline behaviour specification is very similar, but also much more detailed since it is describing the behaviour on instruction category level while ours is, except for instructions which enforce an explicit pipeline stall, independent of the actual instruction set. The assertions obtained from our specification can easily be added to existing test benches which, as we have experienced, facilitates the integration of our method into an existing verification environment.

In [6], Kroening and Paul introduce a tool that supports the design of pipelined microprocessors from prepared sequential machines which are later used as reference models. Prepared in this context means, amongst other things, that the hardware of the original sequential machine has been partitioned into pipeline stages. The design method of pipelining sequential architectures is further explained in [7]. Under the assumption that a prepared sequential machine is given, they present a method that adds a stall engine and also forwarding logic. The correctness of the resulting pipelined design with respect to the original sequential design is then proven with the theorem prover PVS [8]. Designers are expected to provide register mappings for the intermediate results of all pipeline stages. Again, the logic description of the stall engine is very similar to our functional specification. From our point of view the tool proposed by Kroening and Paul is beneficial if the design is developed in a systematic sequential top-down refinement approach. However, from our experience, a combination of top-down and bottom-up design is often necessary to meet the tight time-to-market schedules engineers are faced with in practice. Our method only requires a functional specification of the pipeline flow control logic. Sufficient information to develop a functional specification is often available early in the design cycle *ie* in the microarchitectural description of the pipelined processor design. Like Kroening and Paul we are ultimately aiming to generate HDL code for the pipeline flow control logic from our functional specification.

2. AN EXAMPLE CASE STUDY

2.1 Example Architecture

To demonstrate our method we introduce a simple pipelined microprocessor architecture. Alongside the architectural description we also informally introduce our modelling language by giving a formal description of the features relevant to our analysis.

Our example architecture comprises of two pipes *long* and *short*

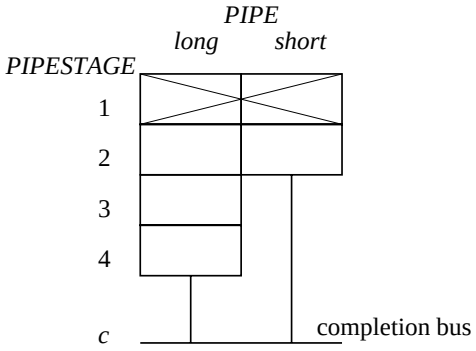

Figure 1: Example pipeline architecture with two pipes and one completion bus.

which share one combined fetch/decode/issue stage. The *long* pipe has two execution stages and one writeback stage, and the *short* pipe has one execution/writeback stage. The first stages of both pipelines operate in lock step, *ie* progress is only made in synchrony. Note that the pipeline stages in Figure 1 have been indexed starting from the fetch/decode/issue stages.

For our formal description we define the sets/types:

$$PIPE = \{long, short\}$$
$$PIPESTAGE = \{1,2,3,4\}$$

For flow control purposes each pipe stage has a moving_or_empty (*moe*) flag, which propagates backwards to its predecessor and indicates whether the stage is moving forward on the next cycle or is currently empty. It is used to determine whether the stage will block the preceding stage. If the *moe* flag is set, the respective stage is ready to take an instruction from the preceding stage on the next cycle. If the *moe* flag is clear, the preceding pipe stage gets blocked, *ie* does not move forward on the next cycle; it stalls.

Further, each pipe stage has a require_to_move (*rtm*) flag which propagates forwards to its successor and indicates whether the contents in that stage intends to move on. So when a stage does not require_to_move, it means either it is not valid (*ie* contains a pipeline bubble) or, if valid it does not need to move into the next pipeline stage (mostly because its processing finishes at that stage and no writeback is required).

$$BOOLEAN\ p.s.moe$$
$$BOOLEAN\ p.s.rtm$$
$$\text{where } p \in PIPE$$
$$s \in PIPESTAGE$$

A special (instruction-specific) flag, which can only be accessed in the fetch/decode/issue stage of the *long* pipe, indicates whether the machine is in a wait state.

$$BOOLEAN\ op\_is\_WAIT$$

Our example architecture has eight registers. The register use is recorded on an eight bit scoreboard, where the register address is used to access the scoreboard bit-array. Each instruction has one source and one destination register.

$$REGADDRESS = \{7..0\}$$
$$SDREG = \{src, dst\}$$

$$BOOLEAN\ scb[8]$$
$$REGADDRESS\ p.1.r.regaddr$$
$$\text{where } p \in PIPE$$
$$r \in SDREG$$

$$
\begin{array}{ll}
 & SPEC_{func} \\
= & \phi(\langle long.4.moe, long.3.moe, long.2.moe, long.1.moe, short.2.moe, short.1.moe\rangle) \\
= & (long.req \wedge \neg long.gnt \rightarrow \neg long.4.moe) \\
\wedge & (long.3.rtm \wedge \neg long.4.moe \rightarrow \neg long.3.moe) \\
\wedge & (long.2.rtm \wedge \neg long.3.moe \rightarrow \neg long.2.moe) \\
\wedge & ((long.1.rtm \wedge \neg long.2.moe \\
 & \vee\ op\_is\_WAIT \\
 & \vee \neg short.1.moe \\
 & \vee \exists r : SDREG\ .\ \exists a : REGADDRESS\ .\ long.1.r.regaddr = a \wedge scb[a] \wedge c.regaddr \neq a) \\
 & \rightarrow \neg long.1.moe) \\
\wedge & (short.req \wedge \neg short.gnt \rightarrow \neg short.2.moe) \\
\wedge & ((short.1.rtm \wedge \neg short.2.moe \\
 & \vee \neg long.1.moe \\
 & \vee \exists r : SDREG\ .\ \exists a : REGADDRESS\ .\ short.1.r.regaddr = a \wedge scb[a] \wedge c.regaddr \neq a) \\
 & \rightarrow \neg short.1.moe)
\end{array}
$$

Figure 2: A functional specification describing the necessary pipeline stalls.

The final stages of both execution subpipes connect to a completion bus named *c*; the *short* pipe has higher priority for completion than the *long* pipe. Each pipe has a dedicated signal to request the completion bus. There is one grant signal for each pipe. To target a register the completion bus holds the target register's address.

$$
\begin{array}{l}
BOOLEAN\ p.req, p.gnt \\
REGADDRESS\ c.regaddr \\
where\ p \in PIPE
\end{array}
$$

This concludes the example architecture. Note that the example processor is simpler than the FirePath in several ways. These include the fact that FirePath is two-sided and has more and deeper execution pipes than the example. Furthermore FirePath has several pipeline decouple stages (shunts), interrupt logic and several completion buses. FirePath has been successfully verified with our method. To keep the example comprehensible, however, we restrict this paper to the given simple architecture.

2.2 Specification

2.2.1 Functional Specification

Our aim is to demonstrate that the pipeline flow control allows maximum performance. By maximum performance we understand that there are no unnecessary pipeline stalls. In the setting described in Section 2.1 this means that there are no situations where the *moe* flag is clear when it should be set.

We can start from a specification, which describes under which conditions it is necessary to stall a pipe stage to avoid hazards. In practice, this information is typically contained in the microarchitectural description of the processor, and hence should be accessible at an early stage during processor design. The following specification describes the individual constraints under which a pipeline stall is necessary in our example architecture.

We start with the completion stages which compete for access to the completion bus. A completion stage does not move on when it does not win the grant to use the completion bus.

$$
\begin{array}{l}
long.4.rtm \wedge \neg long.gnt \rightarrow \neg long.4.moe \\
short.2.rtm \wedge \neg short.gnt \rightarrow \neg short.2.moe
\end{array}
$$

Note that for the completion stages the bus request flag should be set when the *rtm* flag is set, which transforms the above statements to:

$$
\begin{array}{l}
long.req \wedge \neg long.gnt \rightarrow \neg long.4.moe \\
short.req \wedge \neg short.gnt \rightarrow \neg short.2.moe
\end{array}
$$

Note further that the completion logic, *eg* the arbitration scheme of the bus, can also be included in the functional specification. For the sake of simplicity we will concentrate on the *moe* flags only.

To prevent hazards caused by overwriting, the *long* pipe's intermediate stages should stall when their preceding pipe stage stalls.

$$
\begin{array}{l}
long.2.rtm \wedge \neg long.3.moe \rightarrow \neg long.2.moe \\
long.3.rtm \wedge \neg long.4.moe \rightarrow \neg long.3.moe
\end{array}
$$

In the same way, a fetch/decode/issue stage should stall when the respective issue pipe is stalled.

$$
\begin{array}{l}
long.1.rtm \wedge \neg long.2.moe \rightarrow \neg long.1.moe \\
short.1.rtm \wedge \neg short.2.moe \rightarrow \neg short.1.moe
\end{array}
$$

Further, if the machine is in a wait state then the *long* pipe can not issue.

$$
op\_is\_WAIT \rightarrow \neg long.1.moe
$$

In addition, both the *long* and the *short* pipe operate in lock step during fetch, decode and issue.

$$
\begin{array}{l}
\neg long.1.moe \rightarrow \neg short.1.moe \\
\neg short.1.moe \rightarrow \neg long.1.moe
\end{array}
$$

Note that the above two statements describe the logical equivalence of the *moe* flags of the initial stages of both pipelines. We use two implications instead of one equivalence. This makes our specification compositional with respect to individual pipeline stages, which facilitates the maintenance of our specification.

If a source or destination register is outstanding then the instruction cannot be issued. A register is outstanding if it is on the scoreboard, *ie* its scoreboard flag is set, and it is not bypassed, *ie* it will not be written to in the current cycle, which means it is not a completion target register.

$$
\begin{array}{l}
\forall p : PIPE\ .\ \exists r : SDREG\ .\ \exists a : REGADDRESS\ . \\
p.1.r.regaddr = a \wedge scb[a] \wedge c.regaddr \neq a \rightarrow \neg p.1.moe
\end{array}
$$

This statement completes our specification. To get a more compact description we can transform the above specification such that there is exactly one statement for each pipeline stage as given in Figure 2.

$$
\begin{array}{ll}
 & SPEC_{perf} \\
= & (long.req \wedge \neg long.gnt \leftarrow \neg long.4.moe) \\
\wedge & (long.3.rtm \wedge \neg long.4.moe \leftarrow \neg long.3.moe) \\
\wedge & (long.2.rtm \wedge \neg long.3.moe \leftarrow \neg long.2.moe) \\
\wedge & ((long.1.rtm \wedge \neg long.2.moe \\
 & \vee\ op\_is\_WAIT \\
 & \vee \neg short.1.moe \\
 & \vee \exists r : SDREG\ .\ \exists a : REGADDRESS\ .\ long.1.r.regaddr = a \wedge scb[a] \wedge c.regaddr \neq a) \\
 & \leftarrow \neg long.1.moe) \\
\wedge & (short.req \wedge \neg short.gnt \leftarrow \neg short.2.moe) \\
\wedge & ((short.1.rtm \wedge \neg short.2.moe \\
 & \vee \neg long.1.moe \\
 & \vee \exists r : SDREG\ .\ \exists a : REGADDRESS\ .\ short.1.r.regaddr = a \wedge scb[a] \wedge c.regaddr \neq a) \\
 & \leftarrow \neg short.1.moe)
\end{array}
$$

Figure 3: A maximum performance specification.

An assignment to the vector of moving_or_empty flags comprising

$$\langle long.4.moe, long.3.moe, long.2.moe, long.1.moe, short.2.moe, short.1.moe\rangle$$

satisfies this specification iff

$$\vdash \phi(\langle long.4.moe, long.3.moe, long.2.moe, long.1.moe, short.2.moe, short.1.moe\rangle).$$

2.2.2 Performance Specification

The functional specification in Figure 2 contains formulas of the form $condition \rightarrow \neg moe$, where *condition* represents a disjunction of the individual constraints that should lead to a pipeline stall. Note that a violation of this specification occurs when *condition* is satisfied, but the *moe* flag is set, corresponding to a situation where despite the condition holding, the respective pipeline stage signals to its predecessor that it is either moving or empty, so it could be overwritten on the next cycle. The result is that the pipe is moving even though it should not, which will cause a hazard. This corresponds to a functional bug; we therefore call a specification of the above form a *functional* specification.

However, our aim is to identify cases where the pipeline stalls although it is safe to move on. Formally, this can be expressed by specifying that $\neg moe \wedge \neg condition$ should not occur, where *condition* represents a disjunction of all individual constraints that should lead to a pipeline stall. This is logically equivalent to the formula $\neg moe \rightarrow condition$; stating that if the *moe* flag is not set then we would expect *condition* to hold. If *condition* does not hold, then we have indeed identified an unnecessary pipeline stall. A specification containing formulas of this form will be called a *performance* specification.

The performance specification that corresponds to the functional specification introduced in Section *2.2.1* is given in Figure 3. It can be included into a testbench in the form of an assertion. Alternatively, the performance specification can serve as a basis for property checking; we have left property checking for future work.

2.2.3 Concluding Remarks on our Case Study

For obvious reasons, a design should satisfy both its functional specification *and* its performance specification. Hence, the combined specification would contain formulas of the form $condition \leftrightarrow \neg moe$, expressing that the pipeline stalls if and only if *condition* is satisfied. The combined pipeline specification thus corresponds to the specification obtained by changing all $\rightarrow$ in Figure 2 into $\leftrightarrow$.

Because the focus of our project was on performance verification, we only intended to verify the performance part of the combined specification. In practice, it is common to use other verification methods to gain confidence in the functional correctness of the design.

To include the assertions into a testbench, what remains to be done is to translate them into the HDL used for RTL design and simulation. In addition, the signals referred to in the assertion need to be connected to the corresponding signals in the RTL code of the pipeline design. Close cooperation with designers is required to ensure that the RTL signals have the intended semantics.

3. DERIVING THE PERFORMANCE SPECIFICATION FROM THE FUNCTIONAL SPECIFICATION

In the previous section we gave an informal and intuitive understanding of the relationship between the functional and the performance specification. The careful reader will have noticed that it is possible to satisfy our functional specification and yet never move at all. There are, in fact, many possible implementations of this specification of varying performance. So we must ask which of these specifications we require. Of course, we want the best one, *ie* the one which stalls least often. In this section, we will show that the best solution exists and prove that it is derived by changing each $\rightarrow$ in Figure 2 into $\leftrightarrow$ as we have indicated to obtain the combined functional and performance specification in the previous section. In general, the best solution may be more complicated than this but only if control flows in both directions along the pipeline.

3.1 Properties of the Functional Specification

We first establish some properties of the functional specification. It is important that the functional specification can be provided in a form that satisfies these properties; they are preconditions for deriving a maximum performance specification. Section 3.2 describes how the derivation is performed.

Clearly, the functional specification given in Section *2.2.1* does not uniquely determine the values that should be assigned to the *moe* flags. Note for instance that our specification is satisfied if all *moe* flags are set to false.

$$\vdash \phi(\langle False, False, False, False, False, False\rangle) \qquad (1)$$

Assigning false to all *moe* flags blocks the entire pipeline. But,

from a functional point of view, correctness is certainly preserved when the pipeline stalls completely. This is why the specification in Section *2.2.1* is called a functional (correctness) specification.

Note further that our specification can be split into two separate pipeline specifications of the form:

$$\phi(\langle long.4.moe, \ldots, long.1.moe\rangle) = \bigwedge_{i=4}^{1} long.i.F(\neg long.i+1.moe, \neg short.i.moe) \rightarrow \neg long.i.moe$$

and

$$\phi(\langle short.2.moe, short.1.moe\rangle) = \bigwedge_{i=2}^{1} short.i.F(\neg short.i+1.moe, \neg long.i.moe) \rightarrow \neg short.i.moe$$

where the function F, which describes the individual stalling constraints, is constructed using only conjunction and disjunction (on its argument variables), *ie* it is *monotonic*. With suitable indexing the above specification can be further generalised to:

$$\phi(\langle moe\rangle) = \bigwedge(\langle F\rangle(\neg\langle moe\rangle) \rightarrow \neg\langle moe\rangle)$$

such that

$$\langle F\rangle[i](\neg\langle moe\rangle) \rightarrow \neg\langle moe\rangle[i]$$

From the monotonicity of F it follows that ϕ is *disjunctive* in the sense that if two assignments to *moe* flag vectors satisfy ϕ then their bitwise disjunction also satisfies ϕ:

$$\frac{\vdash \phi(\langle moe\rangle_1) \quad \vdash \phi(\langle moe\rangle_2)}{\vdash \phi(\langle moe\rangle_1 \vee \langle moe\rangle_2)} \tag{2}$$

where

$$\langle moe_{n..1}\rangle_1 \vee \langle moe_{n..1}\rangle_2 = \langle\langle moe\rangle_1[n] \vee \langle moe\rangle_2[n], \ldots, \langle moe\rangle_1[1] \vee \langle moe\rangle_2[1]\rangle$$

Now we can show that:

$$F(\neg(\langle moe\rangle_1 \vee \langle moe\rangle_2)) \rightarrow \neg(\langle moe\rangle_1 \vee \langle moe\rangle_2)$$

Assume: $\vdash \phi(\langle moe\rangle_1)$ which means $F(\neg\langle moe\rangle_1) \rightarrow \neg\langle moe\rangle_1$ and $\vdash \phi(\langle moe\rangle_2)$ which means $F(\neg\langle moe\rangle_2) \rightarrow \neg\langle moe\rangle_2$.

Proof:

	$F(\neg(\langle moe\rangle_1 \vee \langle moe\rangle_2))$	
$=$	$F(\neg\langle moe\rangle_1 \wedge \neg\langle moe\rangle_2)$	
$\rightarrow$	$F(\neg\langle moe\rangle_1) \wedge F(\neg\langle moe\rangle_2)$	since $\neg\langle moe\rangle_1 \wedge \neg\langle moe\rangle_2 \rightarrow \neg\langle moe\rangle_i$ where $i \in \{1,2\}$ and F monotonic
$\rightarrow$	$\neg\langle moe\rangle_1 \wedge \neg\langle moe\rangle_2$	by assumption
$=$	$\neg(\langle moe\rangle_1 \vee \langle moe\rangle_2)$	q.e.d.

Properties 1 and 2 of our functional specification form the theoretical foundation for the derivation of the performance specification. The two properties do not restrict the pipeline control flow logic. Establishing the first property is trivial, since our specification does not state anything about when pipeline stages do not stall. To implement pipeline functionality that allows instructions to enter at the fetch/decode/issue stage, progress though the intermediate stages, and exit at a completion stage or possibly earlier, the control flows backwards, starting from the completion stages. For any such pipeline the conditions which cause necessary pipeline stalls can be specified by defining a suitable function F which takes the negated moving_or_empty flags as arguments. Monotonicity should be a natural property of the resulting functional specification.

3.2 Derivation

To get a unique description of the values that should be assigned to the *moe* flags to achieve maximum performance we need to add, to the functional specification, the requirement that the desired set of assignments to *moe* flags is the most liberal, *ie* the assignment which makes more *moe* flags true than any other.

It is important to prove that there is a unique most liberal assignment. To prove this we need to show that there is at least one assignment possible; this is the one that sets all flags in the *moe* vector to false - see property (1), and that or'ing the individual flags of two valid assignments results in another valid assignment; which is established through property (2).

Under the assumption that the functional specification has these two properties, the most liberal assignment to *moe* flags, written $\langle MOE\rangle$, can be obtained by disjunctively combining the individual bits of all valid assignments to *moe* flags.

$$\frac{\langle MOE\rangle = \bigvee\langle moe\rangle \text{ s.t.} \vdash \phi(\langle moe\rangle)}{\vdash \phi(\langle MOE\rangle)} \tag{3}$$

Fixed-point iteration can now be applied to find the least stalling solution. The resulting specification $\phi(\langle MOE\rangle)$ states under which conditions most *moe* flags are set, *ie* how maximum pipeline performance can be achieved. It establishes that, to get maximum performance, each individual *moe* flag should be assigned as follows:

$$\langle MOE\rangle[i] := \neg\langle F\rangle[i](\neg\langle MOE\rangle) \tag{4}$$

This confirms our intuitive approach from Section *2.2.2* and formally establishes the relationship between the functional and the performance parts of our combined specification.

It remains to show that $\langle MOE\rangle$ is indeed the solution that provides the maximum performance. We need to show that any valid assignment to *moe* flags is subsumed by the assignment in *MOE*.

Suppose: $\vdash \phi(\langle moe\rangle')$.

Assume: $\langle moe\rangle'[j] \rightarrow \langle MOE\rangle[j]$ for $j > i$ under the given indexing. This assumption is based on the observation that the *moe* flag of a final stage depends not on *moe* flags but solely on a completion bus grant which is treated as a constant in F.

We can now prove for each *moe* flag *i* in the vector that:

$$\langle moe\rangle'[i] \rightarrow \langle MOE\rangle[i]$$

Proof:

$\langle moe\rangle'[i]$	$\rightarrow$	$\neg\langle F\rangle[i](\neg\langle moe\rangle')$	by 4
	$\rightarrow$	$\neg\langle F\rangle[i](\neg\langle MOE\rangle)$	F monotonic
	$=$	$\langle MOE\rangle[i]$	by 4

Hence:

$$\langle moe\rangle' \rightarrow \langle MOE\rangle \qquad \text{q.e.d.}$$

Note that the inductive proof above is possible because control flows, starting from the completion stages, to the respective predecessor pipeline stages only. If control flows in both directions along the pipeline the best solution may be more complicated.

4. RESULTS

Our aim was to identify possible performance bugs in the pipeline flow control logic of the FirePath processor. A performance bug, in this context, is defined as an unnecessary pipeline stall. Using the architecture and microarchitecture manuals, and in close collaboration with designers, we investigated all constraints that stall the pipeline to preserve functional correctness, and wrote them down in the form of a functional specification that has the properties discussed in Section 3.1. This specification is already a valuable reference for design engineers. We have shown how a maximum performance specification can be obtained from the functional specification. Ideally the combined specification is used as a basis for formal property checking. Alternatively it can be translated into testbench assertions which are checked during simulation.

The specification of the FirePath pipeline design is now a permanent part of the processor's testbench. It ensures that any modifications of the pipeline flow control logic preserve the initial intent. Even the best simulation is by no means exhaustive, hence the fact that the assertions are not triggered during simulation does not imply that the design satisfies the specification. A more thorough approach is to use a property checking tool instead of simulation. Our investigation has found that the specification can also be translated into statements that can be verified by property checkers. Running a property checker means exhaustive verification and is therefore preferable to simulation.

The project took 10 weeks of one person's time from a cold start on the processor architecture and microarchitecture (and Verilog). In summary, we uncovered inefficiencies in the pipeline control flow, and also some incorrect initialisation values of control signals. The completion logic has been redesigned as a consequence of our analysis, resulting in efficiency increase at the pipeline completion stages.

Another achievement of this project is the gain in design understanding amongst engineers. The entire pipeline flow control was formalised and is now documented in the functional specification which serves as a design reference. It greatly helped to clarify how the pipeline flow is controlled, and bridged gaps when parts were designed by different teams. In several cases, functional equivalence of different implementations needed to be established before a more abstract description was accepted across the design teams. An instance of this, which is not addressed in our example architecture, are shunt stages, where the same effect can be achieved via various implementations, all of which should satisfy the same functional specification on a more abstract level.

5. CONCLUSION AND FURTHER WORK

Interlocked pipeline flow control is designed to prevent hazards by stalling the pipeline. The design of the interlock logic is perceived to be tricky and debugging it can delay the design process considerably. There is a danger of introducing unintended pipeline stalls during debugging and more generally whenever the design is modified.

While the prevention of hazards is very important to achieve functional correctness, the detection and prevention of unnecessary pipeline stalls is crucial to maximise pipeline performance and thus obtain high processor performance. The aim of our project was to develop a specification that detects and thus helps to prevent unnecessary pipeline stalls.

We believe that our method can be applied to any pipelined microprocessor design that uses interlock logic to prevent hazards, provided the functional specification of the flow control logic can be given in a form that satisfies the properties detailed in Section 3.1. These properties are not restrictive, in fact they state what one would naturally expect when specifying pipeline control flow.

The simplicity of our approach is based on the fact that it relies purely on a functional understanding of the pipeline flow control logic, which many designers are familiar with at an early stage of the design process. In addition, the specification is, except for instructions which enforce an explicit pipeline stall, independent of the actual instruction set, allowing the control flow to be developed and verified in separation from the data flow. We have found that our method can already deliver useful results, even when the design is not yet complete.

For this pilot project we have derived the performance specification manually. We are now working on a tool which, given a functional specification that has the properties mentioned in Section 3.1, generates the corresponding performance specification and also Verilog/VHDL assertions. This tool will form the first module of a (semi-)automatic pipeline flow control design environment. Ultimately, we would like to generate the HDL code that implements the pipeline flow control logic from the functional specification. This is much more ambitious. Issues like timing and the introduction of shunt stages to decouple pipelines if signal propagation times cannot meet cycle times need to be addressed. A project that aims to show that this is feasible is currently in progress.

6. REFERENCES

[1] M. Bickford and M. Srivas. Verification of a pipelined microprocessor using CLIO. In *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *Lecture Notes in Computer Science*. Springer, 1989.

[2] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *11th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, pages 60–71. Springer, 1999.

[3] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *6th International Conference on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 1994.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[5] K. Kohno and N. Matsumoto. A new verification methodology for complex pipeline behavior. In *DAC 2001 Conference Proceedings*, June 2001.

[6] D. Kroening and W. J. Paul. Automated pipeline design. In *DAC 2001 Conference Proceedings*, June 2001.

[7] S. Müller and W. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[8] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. SRI International, version 2.3 edition, September 1999. Available at `http://pvs.csl.sri.com`.

[9] S. Wilson. Broadcom's FirePath Processor Architecture. In *Embedded Processor Forum*, June 2001.