A Hazards-Based Correctness Statement for Pipelined Circuits*

Mark D. Aagaard

Electrical and Computer Engr., University of Waterloo
maagaard@uwaterloo.ca

Abstract. The productivity and scalability of verifying pipelined circuits can be increased by exploiting the structural and behavioural characteristics that distinguish pipelines from other circuits. This paper presents a formal model of pipelines that augments a state machine with information to describe the transfer of parcels between stages, and reading and writing state variables. Using our model, we created a definition of correctness that is based on the well-established principles of structural, control, and data hazards. We have proved that any pipeline that satisfies our hazards-based definition of correctness is guaranteed to satisfy the conventional correctness statement of Burch-Dill style flushing.

1 Introduction

In early verifications of pipelined circuits, the manual effort to discover abstraction functions limited both the productivity and scalability of verification. Burch and Dill's use of flushing a pipeline to derive an abstraction function automatically [5] improved verification productivity and scalability by sheltering the user from the complexities of the pipeline. Unfortunately, realistic circuits are beyond the scope of such push-button verification. To scale verification to larger pipelines, researchers invented a variety of decomposition strategies. Jones *et al.* used knowledge about pipeline behaviour to create incremental flushing [8]. Pnueli *et al.* [4] and Sawada and Hunt [12] used pipeline behaviour as a guide for defining intermediate models. Hosabettu *et al.* developed completion functions to decompose pipelines stage-by-stage [7]. McMillan used knowledge about the behaviour of pipelines to guide assume-guarantee decomposition [10].

We believe that a model of state machines that captures the distinguishing structure and behaviour of pipelined circuits will improve verification productivity and scalability. The structure of a pipeline is a network of stages through which parcels (instructions) flow. The behaviour of a pipeline can be described using the principles of structural, control, and data hazards. This paper presents a formal model and a correctness statement for pipelines based on stages, parcels, and hazards. Our goals were: remain true to the intuitive meaning of pipelines and hazards, separate orthogonal concerns into distinct correctness obligations, and support cutting-edge optimizations.

Our model of pipelines augments a state machine with pipeline-specific functions and predicates (Section 2): transferring a parcel between stages, writing to a variable,

* This work was supported in part by the National Sciences and Engineering Research Council of Canada and by the Semiconductor Research Corporation Contract RID 1030.001

D. Geist and E. Tronci (Eds.): CHARME 2003, LNCS 2860, pp. 66–80, 2003.

and reading from a variable. The model supports superscalar and out-of-order execution, external kill signals, exceptions, external interrupts, bypass registers, and register renaming [2]. Our correctness statement, *PipeOk*, separates correctness obligations relating to different hazards, datapath functionality and flushing (Section 3). We have proved that any pipeline that satisfies *PipeOk* is guaranteed to satisfy the standard Burch-Dill flushing correctness statement (Section 4).

PipeOk contains thirteen correctness obligations that provide a natural decomposition strategy. Each obligation describes a single type of behaviour, for example, write-after-write hazards. Because hazards are well understood by both verification and design engineers, verification engineers will be able to more easily discuss test plans, verification strategies, and counter examples with designers. Because each obligation focuses on a single type of behaviour, verifying the obligations will be amenable to powerful abstraction mechanisms. For example, the ordering of reads and writes can be verified separately for each variable and need only reason about consecutive operations.

To prove that *PipeOk* implies Burch-Dill correctness, we prove that *PipeOk* implies Flushpoint Equality (flushed states are externally equivalent to specification states) and then use the previously proven result that Flushpoint Equality implies Burch-Dill correctness [3]. We prove that *PipeOk* implies Flushpoint Equality by showing: read and write operations happen in the correct order, the result of each write operation is correct, and finally that flushing works correctly.

2 Modelling Pipelines

This section describes our formal model of pipelines. We begin with an informal description of the "parcel view" of a pipeline, which motivates our approach. The remainder of the section presents the model, auxilliary functions to relate a pipeline to its specification, and conditions to ensure that the auxilliary functions are consistent.

2.1 The Parcel View of a Pipeline

A pipeline is a network of stages. Parcels, or instructions, flow through the stages and read-from and write-to variables, or signals, in the pipeline. Figure 1 shows the runs of a sample program on an instruction set architecture specification, a four-stage pipelined microprocessor, and a "parcel view" of the pipeline. Each run is annotated to show when each parcel moves between stages and when each variable is read or written. The value of a variable is denoted by the label of the instruction that writes to the variable.

Conventional verification strategies compare a snapshot of the pipeline state to a specification state. Because a pipeline state contains the effects of multiple partially executed parcels, it is difficult to relate the implementation to the specification. For example, step 4 of the pipeline contains parcels A, B, C, and D, which represents portions of steps 1, 2, 3, and 4 of the specification. A recent trend has been to examine the implementation only when it is in a flushed state, such as steps 0 and 9 of the pipeline, which are externally equivalent to steps 0 and 5 of the specification.

The parcel view shows slices of the pipeline state as perceived by each parcel. Different variables in the same slice come from different points in time. The slice to

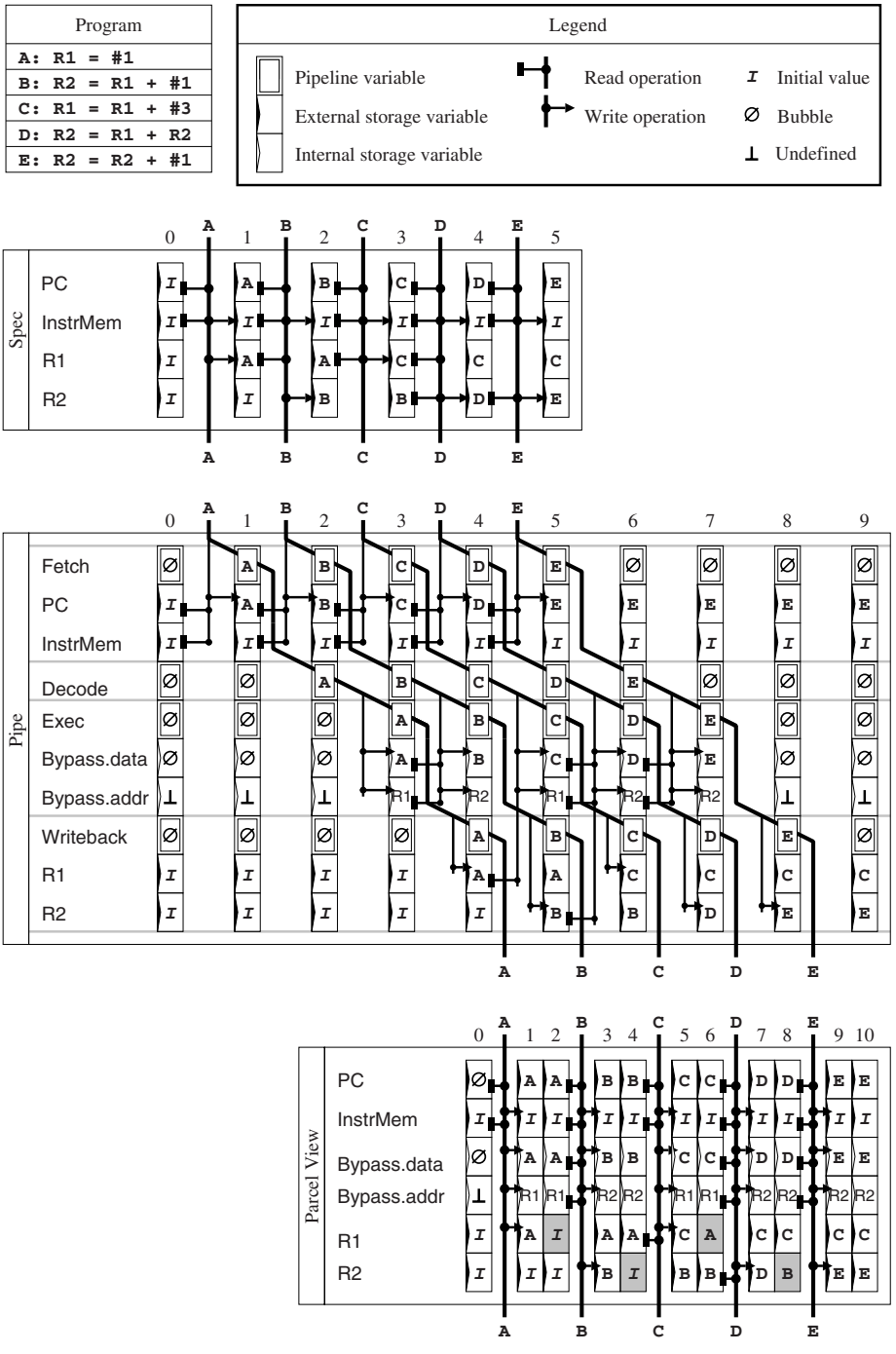


Fig. 1. Specification, pipeline and parcel view of a sample program

Table 1. Definition of a pipeline

Conventional state machine	
state	Set of states.
Nsr	: *state* → *state* → *bool* Next-state relation.
isInit	: *state* → *bool* Initial-state predicate.
Pipeline sets	
stage	Set of identifiers for stages in the pipeline, including *Top* and *Bot*
$addr_i$	Set of identifiers for data storage variables in the pipeline.
isExt	: $(a : addr_i) \rightarrow (q : state) \rightarrow bool$ Variable is externally visible.
isStore	: $(a : addr_i) \rightarrow (q : state) \rightarrow bool$ Variable is for data storage.
subPipes	: $(s : stage) \rightarrow pipe$ One pipe record for each stage
Probes	
xfr	: $(q : state) \rightarrow (s_1 : stage) \rightarrow (s_2 : stage) \rightarrow bool$ In state q, a parcel transfers from s_1 to s_2
Wr	: $(a : addr_i) \rightarrow (q : state) \rightarrow (s : stage) \rightarrow bool$ A parcel in s writes to address a in state q
Rd	: $(a : addr_i) \rightarrow (q : state) \rightarrow (s : stage) \rightarrow bool$ A parcel in s reads from address a in state q

the left (right) of each parcel shows the variables as read (written) by the parcel. Gray backgrounds denote values that are with the specification. For example, in step 2 of the parcel view, R1 is shown in gray, because R1 is `I` in the pipeline and `A` in the specification. The parcel for `B` is able to execute correctly, because it reads its operand from the bypass register, which corresponds to R1 at that time.

The parcel view of pipelines was inspired by two observations: first, for each parcel, the only state variables that are relevant to its correctness are those that it reads or writes; second, if every parcel is executed correctly, then the pipeline is correct. Our proof that our correctness statement, *PipeOk*, implies Burch-Dill flushing relies on the parcel view of the pipeline. We have proved that if the order of read and write operations with respect to *parcels* in the pipeline is the same as the order with respect to *states* in the specification, then data dependencies are obeyed.

2.2 Formal Model of Pipelines

Our formal model of pipelines (Table 1) augments a standard model of non-deterministic state-machines with predicates to detect when parcels transfer between stages, read from state variables, and write to state variables. We use these predicates to compute the parcel view of a pipeline from the next-state relation.

The predicate *xfr* detects the transfer of a parcel between two stages. We have defined instantiations of *xfr* for wide variety of protocols for transfering parcels [1]. Transfers can often be detected using one or two signals, such as the valid bits for the stages. In the set of stages, *Top* and *Bot* are *virtual* stages: they do not exist in the pipeline. For input/output pipelines, such as systolic arrays or execution units in microprocessors, *Top* represents the module in the environment from which parcels enter the pipeline and *Bot* represents the module to which parcels exit. For closed systems, such as microprocessors

Table 2. Functions for comparing a pipeline and specification

Sets	
$addr_s$	Set of identifiers for data storage variables in the specification.
$data_s$	Set of data values in the specification.
Structural-hazard correctness	
Match	$: (\sigma : run) \to (t_1 : time) \to (t_n : time) \to bool$ The parcel that enters at time t_1 exits at time t_n.
Control-hazard correctness	
ShouldExit	$: (\sigma : run) \to (t : time) \to bool$ The parcel that enters should eventually exit
Data-hazard and datapath correctness	
addrmap	$: (a : addr_i) \to (q : state) \to addr_s$ Maps addresses of implementation to addresses in the specification
datamap	$: (a : addr) \to (q : state) \to data_s$ Maps the data in $q.a$ to corresponding specification data value
Flushing correctness	
Flush	$: state \to state$ Flushes a state
IsFlushed	$: state \to bool$ A state is flushed

with built-in memory, transfering from/to *Top* and *Bot* is defined in terms of operations in the pipeline, such as fetching an instruction. Pipelines may contain atomic stages, which hold at most one parcel, and hierarchical stages, which may themselves be pipelines. We support this with the *subPipes* field.

State machines commonly distinguish internal and external variables (*isExt* for "is external"). We refine this by dividing variables into data-storage and pipeline variables (*isStore* for "is storage"). Data-storage variables are used to represent variables in the specification, and can be either internal (*e.g.*, bypass registers) or external (*e.g.*, register files). Pipeline variables are the registers that hold parcels in stages. They are internal and have no corresponding variables in the specification. Read and write predicates need only monitor storage variables.

2.3 Relating Implementations and Specifications

To verify a pipeline against a specification, we need to compare the behaviours of the pipeline and specification. Typically, this is done with a function to say how many instructions are fetched and an external-equivalence relation. Table 2 shows the analagous objects for our model.

We use *Match* to identify the entrance and exit time of each parcel. *Match* supports superscalar pipelines by instantiating the type *time* with a pair of a clock cycle and a port [1]. When working with hierarchical pipelines, we want to treat the stages as black boxes. The *Match* relation allows us to match parcels entering and exiting stages while hiding the internal structure of the stage. We have found five common instantiations for *Match*: degenerate, constant latency, in-order, unique tags, and tagged in-order [1].

Table 3. Consistency Conditions on Pipelines and Specifications

	Specification conditions
1	*The specification is deterministic. This is required for flushpoint-equality correctness to imply Burch-Dill correctness. Implementations may be non-deterministic.*
	Traversal conditions
2	*If ShouldExit is true, then a parcel entered the pipeline.*
3	*Parcels cannot transfer from the pipeline to the "Top" stage.*
4	*Parcels cannot transfer from the "Bot" stage to the pipeline.*
5	*Time increases monotonically as parcels traverse through the pipeline.*
6	*IsFlushed cannot be true while a parcel is traversing through the pipeline.*
7	*A storage operation can happen in a stage only if a parcel is in the stage.*
	Storage Conditions
8	*If an address map changed, then a write must have happened in Impl.*
9	*If a data map changed, then a write must have happened in Impl.*
10	*If a Spec variable changed, then a write must have happened in Spec.*
11	*When a pipeline is flushed, external equality and storage equality are identical.*
	Flushing conditions
12	*Flush is idempotent on flushed pipelines.*
13	*All reachable states are reachable from a flushed state.*
14	*From any state, a flushed state can be reached eventually.*

The predicate *ShouldExit* says whether a parcel that enters the pipeline should be executed. We have identified instantations for *ShouldExit* that include external kill signals, branch prediction, internal exceptions, and external interrupts [2].

We separate external equivalence into two functions: *addrmap*, which defines a mapping between variables in the pipeline and specification, and *datamap*, which maps data in the pipeline to the specification. Address maps may be dependent on the current state: the identity of the specification variable that a bypass register represents is dependent upon the contents the bypass register. When an implementation variable does not represent any specification variable (*e.g.*, a bypass register when it contains a bubble), *addrmap* returns $\perp$, as shown in steps 0–2 for the pipeline in Figure 1.

To relate *PipeOk* to flushpoint equality and Burch-Dill flushing, we require that each pipeline defines a function *Flush* and a predicate *isFlushed*.

2.4 Consistency Conditions

Table refconds summarizes the conditions required for the predicates and functions in the pipeline model to be consistent with the behaviour of the state machine in the model. The complete mathematical definitions appear in a technical report [2].

3 Correctness Obligations

We begin with a summary of our notation. We present our correctness obligations according to the different types of hazards, datapath functionality, and flushing

3.1 Notation

When working with theorems relating a run of a specification to a run of an implementation, we often find it useful to draw "box" or commuting diagrams (Figure 2a). In Figure 2a, x and y refer to the states shown as circles. Properties associated with states and edges are listed in Figure 2b. We denote the t^{th} element of a run σ as: σ^t. We use *run* $m\ \sigma$ to mean that σ is a run of the state-machine m, as defined by: $\forall\, t.\ m\ \sigma^t\ \sigma^{t+1}$. As a syntactic shorthand, we write $m\ q\ q'$ rather than $m.Nsr\ q\ q'$, and we drop the name of the pipeline when refering to parameters other than *Nsr*.

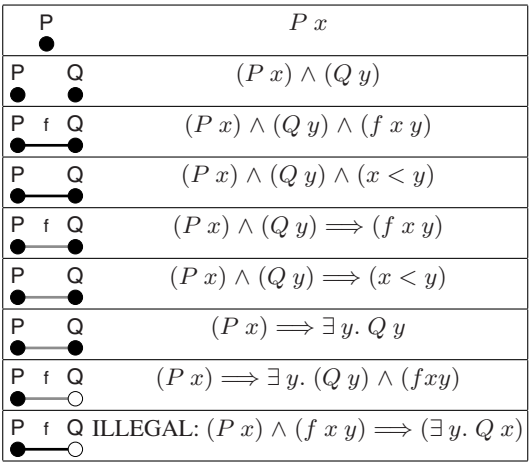


Fig. 2a. Graphical notation

0 The initial ("0^{th}") state
R A read is performed
W A write is performed
F The state is flushed
$\overline{\text{W}}$ No write is performed

Fig. 2b. State and step properties

a Address
q State
s Stage
t Time
σ Run of a state machine

Fig. 2c. Variable identifiers

Fig. 2. Notation and conventions

3.2 Top-Level Correctness Statements

Our top-level correctness statement, Definition 1, *PipeOk*, is the conjunction of thirteen correctness obligations. Each correctness obligation guarantees that a particular type of behaviour is implemented correctly. Section 3.3 describes structural-hazard correctness; Section 3.4 describes data-hazard correctness; Section 3.5 describes datapath functionality correctness; Section 3.6 describes additional correctness obligations needed to ensure that flushed states are externally equivalent to specification states. There are no correctness obligations that address only control hazards. Instead, control hazards permeate both structural hazard correctness and data hazard correctness. For structural hazards, we make sure that correctly speculated parcels are executed and incorrectly speculated parcels do not exit the pipeline. For data hazards, we make sure that incorrectly speculated parcels do not leave behind data results that are read by correctly speculated parcels.

Definition 1 *Correctness of pipelines*

PipeOk Impl Spec ≡

$$
\begin{bmatrix}
\textbf{Struct-hazard correctness} \\
\quad 1\ \textit{EnterTotFun Impl} \\
\wedge\ 2\ \textit{ExitTotFun Impl} \\
\wedge\ 3\ \textit{MatchIffTrav Impl}
\end{bmatrix}
\wedge
\begin{bmatrix}
\textbf{Datapath correctness} \\
10\ \textit{DatapathOk Impl Spec}
\end{bmatrix}
$$

$$
\wedge
\begin{bmatrix}
\textbf{Data-hazard correctness} \\
\quad 5\ \textit{WawHazOk Impl Spec} \\
\wedge\ 4\ \textit{RawHazOk Impl Spec} \\
\wedge\ 6\ \textit{WarHazOk Impl Spec} \\
\wedge\ 7\ \textit{SpecRdTotFun Impl Spec} \\
\wedge\ 8\ \textit{SpecWrTotFun Impl Spec} \\
\wedge\ 9\ \textit{ImplWrTotFun Impl Spec}
\end{bmatrix}
\wedge
\begin{bmatrix}
\textbf{Flushing correctness} \\
\quad 11\ \textit{ImplWrFlush Impl Spec} \\
\wedge\ 12\ \textit{SpecWrFlush Impl Spec} \\
\wedge\ 13\ \textit{ImplInvalidateFlush Impl Spec}
\end{bmatrix}
$$

3.3 Structural-Hazard Correctness Obligations

Structural hazard correctness is concerned with contention between parcels for resources in the pipeline. Typical bugs associated with structural hazards are loss of parcels, duplication of parcels, generation of bogus parcels inside the pipeline, deadlock, and livelock. A pipeline handles its structural hazards correctly if there is a one-to-one mapping between parcels that enter the pipeline and should exit and those parcels that do exit, and if the parcels that exit do so in the correct order.

Definition 2 tracks a parcel as it traverses from stage to stage in a pipeline. The expression $(t_1, s_1) \overset{\sigma}{\rightsquigarrow} (t_n, s_n)$ means that in the run σ, a parcel enters the stage s_1 at t_1, traverses from s_1 to s_n, and exits the stage s_n at t_n. In the base case s_1 and s_n are the same stage. In the inductive case, there is an intermediate stage s_2 such that the parcel transfers from s_1 to s_2 and then traverses from s_2 to s_n. To detect when the parcel exits s_1, we use the matching relation provided by s_1, according to our hierarchical model of pipelines. Definition 2 supports pipelines with loops, because *Match* separately identifies each iteration. We use $\rightsquigarrow$ to define *Trav*, which means a parcel traverses through the pipeline from *Top* to *Bot*.

Definition 2 *Traversing between stages in a pipeline* ($\rightsquigarrow$)

$$
(t_1, s_1) \overset{\sigma}{\rightsquigarrow} (t_n, s_n) \equiv
\begin{bmatrix}
\wedge\ s_1 = s_n \\
\quad s_1.\textit{Match}\ \sigma\ t_1\ t_x
\end{bmatrix}
\vee
\begin{bmatrix}
\exists\, t_2, s_2. \\
\quad \wedge\ s_1.\textit{Match}\ \sigma\ t_1\ t_2 \\
\quad \wedge\ \textit{xfr}\ \sigma^{t_2}\ s_1\ s_2 \\
\quad \wedge\ (t_2, s_2) \overset{\sigma}{\rightsquigarrow} (t_n, s_n)
\end{bmatrix}
$$

Obligation 1, *EnterTotFun*, says that for each time (t_1) that a parcel enters the pipeline and should exit, there exists exactly one time (t_2) such that the parcels exits at t_2 (total and functional). Obligation 2, *ExitTotFun*, says that each parcel that exits the pipeline (*XfrOut*) comes from exactly one parcel that entered the pipeline and should have exited (surjective and injective). Together, Obligations 1 and 2 guarantee that the relationship between entering and exiting parcels is bijective.

Obligation 1 *Each entrance results in exactly one exit*

$$\begin{array}{l} \mathit{EnterTotFun}\ \mathit{Impl} \equiv \\ \quad \forall\, \sigma_i, t_1. \\ \qquad \left[\begin{array}{l} \mathit{run}\ \mathit{Impl}\ \sigma_i \\ \wedge\ \mathit{isFlushed}\ \sigma_i^0 \\ \wedge\ \mathit{ShouldExit}\ \sigma_i\ t_1 \end{array}\right] \Longrightarrow \exists!\, t_2.\ \mathit{Trav}\ \mathit{Impl}\ \sigma_i\ t_1\ t_2 \end{array}$$

Obligation 2 *Each exit comes from exactly one entrance*

$$\begin{array}{l} \mathit{ExitTotFun}\ \mathit{Impl} \equiv \\ \quad \forall\, \sigma_i, t_2. \\ \qquad \left[\begin{array}{l} \mathit{run}\ \mathit{Impl}\ \sigma_i \\ \wedge\ \mathit{IsFlushed}\ \sigma_i^0 \\ \wedge\ \mathit{XfrOut}\ \sigma_1^{t_2} \end{array}\right] \Longrightarrow \left[\begin{array}{l} \exists!\, t_1. \\ \quad \wedge\ \mathit{Trav}\ \mathit{Impl}\ \sigma_i\ t_1\ t_2 \\ \quad \wedge\ \mathit{ShouldExit}\ \sigma_i\ t_1 \end{array}\right] \end{array}$$

Obligation 3, *MatchIffTrav*, says that parcels that exit the pipeline do so in the correct order, as defined by the pipeline-specific matching relation (*Match*). *MatchIffTrav* allows pipelines to be treated as black boxes in hierarchical verification, by relating the traversal of parcels inside the pipeline, *Trav*, to the entrance and exit of parcels.

Obligation 3 *Match correctly identifies when a parcel traverses the pipeline*

$$\begin{array}{l} \mathit{MatchIffTrav}\ \mathit{Impl} \equiv \\ \quad \forall\, \sigma, t_1, t_2. \quad [\mathit{Match}\ \mathit{Impl}\ \sigma\ t_1\ t_2] \Longleftrightarrow [\mathit{Trav}\ \mathit{Impl}\ \sigma\ t_1\ t_2] \end{array}$$

3.4 Data-Hazard Correctness Obligations

A data-depenency exists between a producing (writing) instruction and a consuming (reading) instruction if the producing instruction writes to an address that the consuming instruction reads from and no instruction between the producer and the consumer writes to that address. A pipeline implements data dependencies correctly if every data dependency in the specification is obeyed in the implementation.

Data hazards are categorized as: read-after-write, write-after-read, and write-after-write. If a pipeline handles all three types of data hazards correctly, then it implements data dependencies correctly. In Figure 3, the gray lines represent orderings between specification and implementation operations that will violate the dependency between $\mathsf{W_i}$ and $\mathsf{R_i}$. Read-after-write (Raw) hazard correctness guarantees that $\mathsf{R_i}$ occurs after $\mathsf{W_i}$. Together, write-after-write and write-after-read hazard correctness guarantee that no write will occur to this address between $\mathsf{W_i}$ and $\mathsf{R_i}$. Write-after-write (Waw) correctness guarantees that no programmatically earlier write happens after $\mathsf{W_i}$. Write-after-read (War) correctness guarantess that no programmatically later write will occur before $\mathsf{R_i}$. Figure 3 has many simplifications that are violated by optimizations such as bypass registers, register renaming, and out-of-order execution. Our formalization supports these optimizations using dynamic address maps, multiple writes, and out-of-order writes [2].

The data hazard obligations ensure that reads and writes in the implementation occur in the correct order. We use the symbols $\mathsf{Wr}\!<\!\mathsf{Rd}$, $\mathsf{Rd}\!<\!\mathsf{Wr}$, and $\mathsf{Wr}\!<\!\mathsf{Wr}$ to denote consecutive write and read operations in a run. Definition 3 describes a read following a write to the address a in the run σ. To the right of the text is an illustration of the definition using the graphical notation presented in Figure 2a. The definitions for a write following a read and a write following a write are similar.

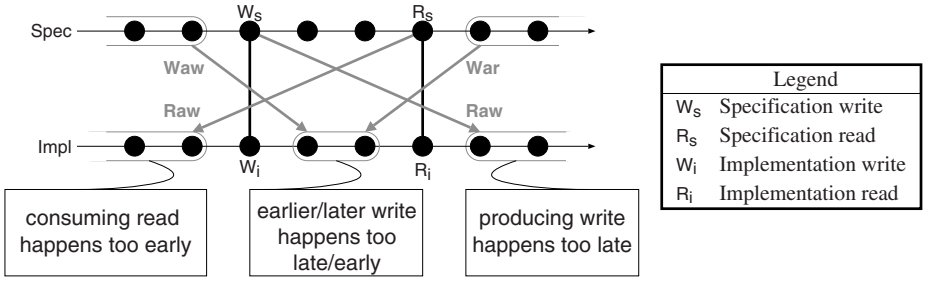
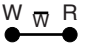


Fig. 3. Data-dependencies and the three types of data hazards

Definition 3 *Consecutive read-after-write ordering*

$$
\begin{aligned}
&(t_w, s_w)\ \overset{\sigma}{\mathsf{Wr}\underset{a}{<}\mathsf{Rd}}\ (t_r, s_r) \equiv \\
&\quad \wedge\ \mathit{Wr}\ a\ \sigma^{t_w}\ s_w \\
&\quad \wedge\ \mathit{Rd}\ a\ \sigma^{t_r}\ s_r \\
&\quad \wedge\ t_w < t_r \\
&\quad \wedge\ \forall\, t \in \{t_w + 1..t_r - 1\}.\ \forall\, s.\ \neg(\mathit{Wr}\ a\ \sigma^{t}\ s)
\end{aligned}
$$

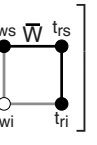


Obligation 4, *RawHazOk*, says that if there is a data-dependency in the specification and a corresponding read in the implementation (a_i, t_{ri}, s_r), then there must exist a corresponding write (a_i, t_{wi}, s_w) that happens before the read.

Obligation 4 *Correctness of read-after-write data hazards*

RawHazOk Impl Spec $\equiv$

$$
\forall\, \sigma_s, \sigma_i,\ a_s, t_{ws}, t_{rs},\ a_i, t_{ri}, s_r.
\left[
\begin{aligned}
&\wedge\ \mathit{Spec}\ \overset{\sigma_s\ \sigma_i}{\overline{\overline{\mathsf{RUN}}}}\ \mathit{Impl} \\
&\wedge\ t_{ws}\ \overset{\sigma_s}{\mathsf{Wr}\underset{a_s}{<}\mathsf{Rd}}\ t_{rs} \\
&\quad (a_s, t_{rs})\ \overset{\sigma_s\ \sigma_i}{\overline{\overline{\mathsf{Rd}}}}\ (a_i, t_{ri}, s_r)
\end{aligned}
\right]
\Longrightarrow
\left[
\begin{aligned}
&\exists\, t_{wi}, s_w. \\
&\quad \wedge\ (a_s, t_{ws})\ \overset{\sigma_s\ \sigma_i}{\overline{\overline{\mathsf{Wr}}}}\ (a_i, t_{wi}, s_w) \\
&\qquad t_{wi} < t_{ri}
\end{aligned}
\right]
$$



RawHazOk contains the first appearance of the relation $\mathit{Spec}\ \overset{\sigma_s\ \sigma_i}{\overline{\overline{\mathsf{RUN}}}}\ \mathit{Impl}$ ("run correspondence") which says that: σ_s is a run of *Spec*, σ_i is a run of *Impl* from a flushed state, and the initial states of σ_s and σ_i are externally equivalent.

We formalize an operation in a run of an implementation *corresponding* to an operation in the specification by tracking a parcel as it traverses the pipeline. The n^{th} parcel that enters the pipeline and should exit corresponds to the n^{th} step of the specification. The expression $t_s\ \overset{\sigma_s\ \sigma_i}{\overline{\overline{\mathsf{PCL}}}}\ (t_{in}, s_n)$ means that at time t_{in}, the t_s^{th} parcel that entered the pipeline and should exit is either inside the stage s_n or is just exiting s_n.

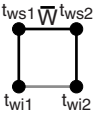
Read and write correspondences are defined in terms of parcel correspondence ($\overline{\overline{\mathsf{PCL}}}$). The expression $(a_s, t_{ws})\ \overset{\sigma_s\ \sigma_i}{\overline{\overline{\mathsf{Wr}}}}\ (a_i, t_{wi}, s)$ means: the specification instruction at time t_{ws} writes to address a_s, the instruction corresponds to the parcel in stage s at time t_{wi}, the parcel writes to a_i, and the address map of a_i at time t_{wi} points to a_s.

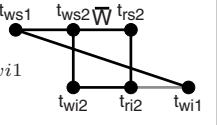
Write-after-write and write-after-read hazards are dealt with by Obligations 5 and 6, both of which have a case for in-order writes and a case for out-of-order writes.

The in-order cases are simpler, because they deal only with consecutive operations, as denoted by $\overline{\mathsf{W}}$. The out-of-order cases require looking beyond consecutive operations, because we do not know how far out-of-order the operations will be. We use "$\overset{}{\underset{}{\ll}}_{\mathsf{Wr}}$" and "$\overset{}{\underset{}{\ll}}_{\mathsf{Rd}}$" for the transitive ordering of write and read operations.

Obligation 5 *Correctness of write-after-write data hazards*

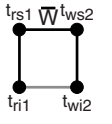
$$
\begin{array}{l}
\mathit{WawHazOk}\ \mathit{Impl}\ \mathit{Spec} \equiv \\
\quad \forall\, \sigma_s, \sigma_i,\ a_s, t_{ws1}, a_i, t_{wi1}, s_{w1},\ t_{wi2}, s_{w2}. \\
\qquad \left[\begin{array}{l} \mathit{Spec} \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{RUN}}{=\!=}} \mathit{Impl} \\ \wedge \\ (a_s, t_{ws1}) \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{Wr}}{=\!=}} (a_i, t_{wi1}, s_{w1}) \end{array}\right] \\
\qquad \Longrightarrow \\
\qquad \left[\begin{array}{l} \forall\, t_{ws2}. \\ \qquad \left[\begin{array}{l} t_{ws1} \overset{\sigma_s}{\underset{a_s}{\mathsf{Wr}<\mathsf{Wr}}} t_{ws2} \\ \wedge \\ (a_s, t_{ws2}) \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{Wr}}{=\!=}} (a_i, t_{wi2}, s_{w2}) \end{array}\right] \Longrightarrow t_{wi1} < t_{wi2} \\ \vee \\ \forall\, t_{ws2}, t_{rs2}, t_{wi2}, s_{w2}, t_{ri2}, s_{r2}. \\ \qquad \left[\begin{array}{l} \wedge\ t_{ws1} \overset{\sigma_s}{\underset{a_s}{\mathsf{Wr}\ll\mathsf{Wr}}} t_{ws2} \\ \wedge\ t_{ws2} \overset{\sigma_s}{\underset{a_s}{\mathsf{Wr}<\mathsf{Rd}}} t_{rs2} \\ \wedge\ (a_s, t_{ws2}) \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{Wr}}{=\!=}} (a_i, t_{wi2}, s_{w2}) \\ \quad (a_s, t_{rs2}) \overset{\sigma_i\ \sigma_i}{\underset{\mathsf{Rd}}{=\!=}} (a_i, t_{ri2}, s_{r2}) \end{array}\right] \Longrightarrow t_{ri2} \leq t_{wi1} \end{array}\right]
\end{array}
$$

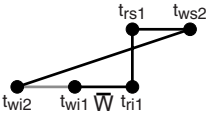




Obligation 6 *Correctness of write-after-read data hazards*

$$
\begin{array}{l}
\mathit{WarHazOk}\ \mathit{Impl}\ \mathit{Spec} \equiv \\
\quad \forall\, \sigma_s, \sigma_i,\ a_s, t_{rs1},\ a_i, t_{ri1}, s_{r1}. \\
\qquad \left[\begin{array}{l} \mathit{Spec} \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{RUN}}{=\!=}} \mathit{Impl} \\ \wedge \\ (a_s, t_{rs1}) \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{Rd}}{=\!=}} (a_i, t_{ri1}, s_{r1}) \end{array}\right] \\
\qquad \Longrightarrow \\
\qquad \left[\begin{array}{l} \forall\, t_{ws2}. \\ \qquad \left[\begin{array}{l} t_{rs1} \overset{\sigma_s}{\underset{a_s}{\mathsf{Rd}<\mathsf{Wr}}} t_{ws2} \\ \wedge \\ (a_s, t_{ws2}) \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{Wr}}{=\!=}} (a_i, t_{wi2}, s_{w2}) \end{array}\right] \Longrightarrow t_{ri1} \leq t_{wi2} \\ \vee \\ \forall\, t_{ws2}, t_{wi1}, s_{w1}. \\ \qquad \left[\begin{array}{l} t_{rs1} \overset{\sigma_s}{\underset{a_s}{\mathsf{Rd}\ll\mathsf{Wr}}} t_{ws2} \\ \wedge \\ (t_{wi1}, s_{w1}) \overset{\sigma_i}{\underset{a_i}{\mathsf{Wr}<\mathsf{Rd}}} (t_{ri1}, s_{r1}) \end{array}\right] \Longrightarrow t_{wi2} < t_{wi1} \end{array}\right]
\end{array}
$$





The out-of-order case for Obligation 5 requires that t_{wi1} does not corrupt data by occurring between another implementation write (t_{wi2}) and its dependent read (t_{ri2}).

The out-of-order case of Obligation 6, *WarHazOk*, is simpler than that of Obligation 5, *WawHazOk*, because we do not need to mention the specification write that corresponds to t_{wi1}. The purpose of the out-of-order case is to allow t_{wi2} to happen before t_{ri1} while ensuring that t_{wi2} does not corrupt the data intended for t_{ri1}. If t_{wi2} corrupts the data, then t_{wi2} will be the producer for t_{ri1}, which causes the right-hand-side of the implication to be $t_{wi2} < t_{wi2}$, which is clearly false.

Obligations 4–6 guarantee that, if read and write operations occur in the implementation, then they will occur in the correct order. These obligations do not guarantee that the operations actually do occur in the implementation. Obligations 7–9 ensure that reads and writes in the specification will also occur in the implementation and that writes that occur in the implementation correspond to writes in the specification. For brevity, we omit the mathematical definitions, which can be found elsewhere [2].

Obligation 7 *SpecRdTotFun Impl Spec* $\equiv$ *Each read operation in Spec corresponds to exactly one read operation in Impl*

We allow multiple writes in the implementation to correspond to a single write in the specification, so long as the writes are to different variables (Obligation 8, *SpecWrTotFun*). This feature is required to support simple optimizations, such as bypass registers, as well as complex optimizations, such as retirement register files.

Obligation 8 *SpecWrTotFun Impl Spec* $\equiv$ *Each write in Spec has at least one corresponding write in Impl. If two writes in Impl correspond to the same write in Spec, then the Impl writes must be to different addresses in Impl.*

We allow implementations to perform writes that do not correspond to writes in the specification, so long as these writes are not read (Obligation 9, *ImplWrTotFun*). This freedom provides a uniform mechanism for implementations to invalidate data, (remapping a register in register renaming) as well as modify the contents of variables that are not needed (bubbles changing the value of a bypass register as they propagate through it). A variable is invalid if its address map is changed so that it no longer points to an address in the specification. As shown in Figure 1, when a bypass register contains a bubble, we say that its address map returns $\perp$. Obligations 11–13 in Section 3.6 ensure that these writes do not corrupt data before a flushed state.

Obligation 9 *ImplWrTotFun Impl Spec* $\equiv$ *Each write in Impl that is the last write before a read from the same address must have a corresponding write in Spec.*

3.5 Datapath Correctness Obligation

Definition 4 describes when two storage variables are equivalent: their address maps point to the same address and their data maps return the same data value.

Definition 4 *Equality of storage variables*

$$(a_1, q_1) \underset{\text{STORE}}{=\!=} (a_2, q_2) \equiv$$
$$[a_1 = \mathit{addrmap}\ a_2\ q_2] \wedge [q_1.a_1 = \mathit{datamap}\ a_2\ q_2]$$

The datapath of a pipeline is correct if, assuming every read operation that a parcel performs will consume the correct data, then every write that parcel performs must produce the correct data (Obligation 10, *DatapathOk*). The clause dealing with reads is nested within the antecedent to provide a uniform way of dealing with both parcels that

performs reads and those whose results are independent of the contents of the pipeline storage variables.

Obligation 10 *Correctness of datapath (DatapathOk)*

$$\begin{array}{l} \mathit{DatapathOk}\ \mathit{Impl}\ \mathit{Spec} \equiv \\ \quad \forall\, \sigma_s, \sigma_i,\ a_s, t_s,\ a_{wi}, t_{wi}, s_w. \\ \quad \left[\begin{array}{l} \wedge\ \mathit{Spec} \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{RUN}}{=\!=}} \mathit{Impl} \\ \quad \left[\begin{array}{l} \forall\, a_{rs}, a_{ri}, t_{ri}, s_r. \\ \quad (a_{rs}, t_s) \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{Rd}}{=\!=}} (a_{ri}, t_{ri}, s_r) \\ \quad \Longrightarrow \\ \quad (a_{rs}, \sigma_s^{t_s}) \underset{\mathsf{STORE}}{=\!=} (a_{ri}, \sigma_i^{t_{ri}}) \end{array}\right] \\ \wedge\ (a_{ws}, t_s) \overset{\sigma_s\ \sigma_i}{\underset{\mathsf{Wr}}{=\!=}} (a_{wi}, t_{wi}, s_w) \end{array}\right] \Longrightarrow \left[(a_{ws}, \sigma_s^{t_s+1}) \underset{\mathsf{STORE}}{=\!=} (a_{wi}, \sigma_i^{t_{wi}+1})\right] \end{array}$$

3.6 Flushing Correctness Obligations

Using Obligations 1–10, we have proved that every parcel that enters the pipeline and should exit, will produce the correct result (*WriteOk* in Figure 4). It may seem that this is a sufficient definition of correctness, however it allows externally visible state variables that are written but never read to contain incorrect data. We solve this problem with Obligations 11–13 (mathematical definitions appear elsewhere [2]). Obligation 11, *ImplWrFlush*, is analogous to Obligation 9, *ImplWrTotFun*, except that it is concerned with writes before flushed states, rather than writes before reads. Obligation 12, *SpecWrFlush*, ensures that in a flushed implementation state, the last writes that happened in the specification have corresponding writes in the implementation. Finally, Obligation 13, *ImplInvalidateFlush*, ensures that for each specification variable, there is at least one corresponding implementation variable. This is done by preventing the invalidation of the last corresponding implementation variable.

Obligation 11 *ImplWrFlush Impl Spec* $\equiv$ *Last visible writes in impl before flushed states correspond to writes in spec.*

Obligation 12 *SpecWrFlush Impl Spec* $\equiv$ *Last visible writes in spec occur in impl*

Obligation 13 *ImplInvalidateFlush Impl Spec* $\equiv$ *If the address map of a variable* (a_i) *changes, then in the next clock cycle there must be another implementation variable* (a_2) *such that the address map of* a_2 *points to the same specification address as* a_1 *used to point to.*

4 Proof That Hazard-Correctness Implies Burch-Dill Correctness

The proof that *PipeOk* implies Burch-Dill flushing (Theorem 1) contains four major steps that are linked by transitivity (Figure 4). In the first step, we used the correctness obligations for structural, control, and data hazards (Obligations 1–9) to prove that the read and write operations in the implementation obey data dependencies in the specification. That is, the operations exist and occur in the correct order (*DataDepOk*). In

the second step, we combined the ordering of data-storage operations with the correctness of the datapath (Obligation 10) to prove that every write operation writes the correct data (*WriteOk*). In the third step, we combined the correctness of write operations with the correctness obligations for flushing (Obligations 11–13) to prove that when a pipeline is in flushed state, it will correspond to the specification (*FlushedEq*). The definition of *FlushedEq* comes from the Microbox work of Aagaard *et al* [3], where it is identified by the acronym iFEND for "informed-flushpoint with equality between a non-deterministic implementation and a deterministic specification".

Definition 5 *Burch-Dill correctness*

$$\begin{aligned} &\mathit{BurchDillOk}\ \mathit{Impl}\ \mathit{Spec} \equiv \\ &\quad \forall\, q_i, q_s, q_i'. \\ &\quad \begin{bmatrix} \mathit{Flush}\ q_i \underset{\text{EXT}}{=\!=} q_s \\ \wedge\ \mathit{Impl}\ q_i\ q_i' \\ \wedge\ \mathit{DoesFetch}\ q_i\ q_i' \end{bmatrix} \Longrightarrow \begin{bmatrix} \exists\, q_s'. \\ \quad \wedge\ \mathit{Flush}\ q_i' \underset{\text{EXT}}{=\!=} q_s' \\ \quad\ \ \mathit{Spec}\ q_s\ q_s' \end{bmatrix} \end{aligned}$$

Theorem 1 *Pipeline correctness implies Burch-Dill correctness*

$$\begin{aligned} &\mathit{PipeOkImpBurchDillOk} \equiv \\ &\quad \forall\, \mathit{Impl}, \mathit{Spec}. \\ &\quad\ \ \mathit{PipeOk}\ \mathit{Impl}\ \mathit{Spec} \quad \Longrightarrow \quad \mathit{BurchDillOk}\ \mathit{Impl}\ \mathit{Spec} \end{aligned}$$

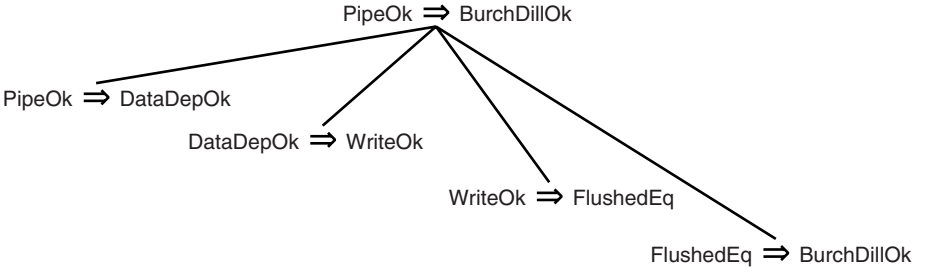


Fig. 4. Proof sketch that PipeOk implies Burch-Dill flushing

5 Conclusions

Some related work has been on correctness for pipelined circuits. Tahar and Kumar defined correctness statements for the different types of hazards in a single-scalar, in-order microprocessor [13]. Manolios has used bisimulation and retiming to relate the run of a pipeline to a specification using state-based abstraction functions, such as flushing [9]. Mishra *et al* defined correctness for pipelined microprocessors with the restriction that instructions proceed from stage to stage in a lockstep order [11].

Some of the lemmas and decomposition strategies used by others are similar to correctness obligations in our work. McMillan's inductive proof to show that each instruction that reads correct data will write correct results [10] is similar to our obligation for datapath correctness. Sawada's MAETT annotates implementation states with history and prophecy variables to facilitate separating the effects of individual instructions [12].

This is similar in flavour to our use of read and write operations to identify the relevant state variables for each instruction. Ho's *token networks* [6] are a verification strategy that might yield useful abstractions to verify our structural hazard obligations.

The goal of the work presented here was to establish a formal foundation for pipelined circuits that would increase verification capacity and productivity, be intuitive to both verification engineers and design engineers, and handle cutting-edge optimizations in pipelines. We have defined a formal model and correctness statement (*PipeOk*) based upon conventional notions of stages, parcels, and hazards. We have proved that the correctness statement guarantees Burch-Dill flushing correctness. *PipeOk* is comprised of thirteen correctness obligations: three for structural hazards, six for data hazards, one for the datapath, and three for flushing. Control hazards are integrated into structural and data hazard correctness. The correctness obligations each deal with a specific type of behaviour, which should make them amenable to powerful abstraction and problem reduction techniques. We have begun several case studies to evaluate the effectiveness of *PipeOk* using a combination of model checking and theorem proving. After the case studies indicate that our model and correctness statement are effective, we will mechanize the proof that *PipeOk* implies Flushpoint Equality.

References

1. M. D. Aagaard. *A Framework for the Specification, Design, and Verification of Pipelined Circuits*. PhD thesis, School of Elect. Eng, Cornell Univ., Jan. 1995.
2. M. D. Aagaard. A formal model and correctness statement for pipelined circuits. Technical report, E&CE, Univ. of Waterloo, Apr. 2003.
3. M. D. Aagaard, N. A. Day, and M. Lou. Relating multi-step and single-step microprocessor correctness statements. In *FMCAD*, pages 123–141. Springer Verlag, 2002.
4. T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution with exceptions. In *TACAS*, pages 487–502. Springer Verlag, 2000.
5. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, pages 68–60. Springer Verlag, 1994.
6. P.-H. Ho, A. J. Isles, and K. Timothy. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *ICCAD*, pages 529–536. 1998.
7. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *CAV*, pages 521–537. Springer Verlag, 2000.
8. R. Jones, J. Skakkebæk, and D. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *FMCAD*, pages 2–17. Springer Verlag, 1998.
9. P. Manolios. Correctness of pipelined machines. In *FMCAD*, pages 161–178. Springer Verlag, 2000.
10. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV*, pages 110–121. Springer Verlag, 1998.
11. P. Mishra, H. Tomiyama, N. Dutt, and A. Nicolau. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *DATE*, pages 36–43, 2002.
12. J. Sawada and W. A. Hunt. Verifying the FM9801 microarchitecture. *IEEE Micro*, 19(3):47–55, May/June 1999.
13. S. Tahar and R. Kumar. A practical methodology for the formal verification of RISC processors. *Formal Methods in System Design*, 13(2):159–225, Sept. 1998.