Formal Verification of Pipelined Microprocessors

Dissertation
zur Erlangung des Grades Doktor der
Ingenieurswissenschaften (Dr.-Ing.) der
Naturwissenschaftlich-Technischen Fakultät I der
Universität des Saarlandes

Daniel Kröning

Saarbrücken, 2001

Abstract

Subject of this thesis is the formal verification of pipelined microprocessors. This includes processors with state of the art schedulers, such as the Tomasulo scheduler and speculation. In contrast to most of the literature, we verify synthesizable design at gate level. Furthermore, we prove both data consistency and liveness. We verify the proofs using the theorem proving system PVS. We verify both in-order and out-of-order machines. For verifying in-order machines, we extend the stall engine concept presented in [MP00]. We describe and implement an algorithm that does the transformation into a pipelined machine. We describe a generic machine that supports speculating on arbitraty values. We formally verify proofs for the Tomasulo scheduling algorithm with reorder buffer.

Kurzzusammenfassung

Gegenstand dieser Dissertation ist die formale Verifikation von Mikroprozessoren mit Pipeline. Dies beinhaltet auch Prozessoren mit aktuellen Scheduling-Verfahren wie den Tomasulo Scheduler und spekulativer Ausführung. Im Gegensatz zu weiten Teilen der bestehenden Literatur führen wir die Verifikation auf Gatter-Ebene durch. Des weitern beweisen wir sowohl Datenkonsistenz als auch eine obere Schranke für die Ausführungszeit. Die Beweise werden mit dem Theorem Beweissystem PVS verifiziert. Es werden sowohl in-order Maschinen als auch out-of-order Maschinen verifiziert. Zur Verifikation der in-order Maschinen erweitern wir die Stall Engine aus [MP00]. Wir beschreiben und Implementieren ein Verfahren das die Transformation in die “pipelined machine” durchführt. Wir beschreiben eine generische Maschine die Spekulation auf beliebige Werte erlaubt. Wir verifizieren die Beweise für den Tomasulo Scheduler mit Reorder Buffer.

Extended Abstract

Microprocessors are in use in many safety-critical environments, such as cars or planes. We therefore consider the correctness of such components as a matter of vital importance. Testing microprocessors is limited by the huge state space of modern microprocessors. We therefore think formal verification is the sole way to obtain a guarantee.

This formal verification should be done such that any third party is able to verify the correctness with low effort, i.e., we aim to provide a proof of correctness that can be checked mechanically. In particular, we think that all critical designs should be delivered in form of a four-tuple: 1) the design itself, 2) a specification, 3) a human-readable proof, and 4) a machine-verified proof.

In this thesis, we present proofs of correctness for complex microprocessors. Designing microprocessors is considered an error-prone process. A well known example for this is the Pentium FDIV bug [Coe95, Pra95].

In this thesis, we provide a rigorously formal approach to hardware verification. The designs presented in this thesis include state of the art schedulers, such as the Tomasulo scheduler [Tom67] and speculation. In contrast to most of the literature, the designs we provide are very close to gate level. In particular, we are synthesizing some of the designs for the XILINX FPGA series.

These designs are of high complexity, and so are the proofs. In contrast to [MP95, Lei99, MP00], the proofs are machine verified using the theorem proving system PVS [CRSS94]. We do not present the original PVS proof in this thesis but aim to provide comprehensible paper-and-pencil proofs.

In order to verify sequential machines, we extend the data consistency invariant given in [MP00] by defining a “correct value” of an implementation register such as $IR.2$. Given the correctness of functional components such as the ALU, this allows for an almost fully automated proof of the data consistency of the prepared sequential machine using PVS. We argue that the correct functional components provide correct results if given correct inputs.

We extend the stall engine concept presented in [MP00] by providing

a fully generic stall engine design. In contrast to [MP00], our stall engine design supports an arbitrary number of stages and allows for stalling (and therefore clocking) all stages independently. Furthermore, it supports *pipeline bubble removal*, i.e., the stages are clocked whenether the in-order property permits this. This includes that bubbles are removed from the pipeline if necessary. We formally verify data consistency and liveness properties for this stall engine.

Using this extended stall engine, we improve the process of transforming the prepared sequential machine into the pipelined machine by providing a tool that does this transformation automatically. This includes the generation for forwarding and interlock hardware.

We then prove the data consistency of the pipelined machine. We do so by showing that the inputs of the pipeline stages are correct. Using this fact, we argue the correctness of the output values as we do for the prepared sequential machine, since the functional components of the machines are identical.

We present a generic approach to speculative execution and propose a data consistency criterion for such a machine. We then apply this method in order to implement and prove DLX pipelines with branch prediction and precise interrupts. It is a well-known fact that both techniques are implemented using speculation [SP88]. However, to the best of our knowledge, implementing both techniques as an instance of a generic speculation mechanism is done for the first time.

Besides the in-order pipelines, we verify the correctness of the Tomasulo scheduling algorithm with reorder buffer as described in [KMP99]. The reorder buffer realizes in-order termination, which allows implementing precise interrupts. The proof of correctness covers the arguments neccessary to show the uniqueness of the tags.

Furthermore, we rigorously prove the liveness of all machines we design, i.e., we prove that any given instruction sequence is executed within a finite amount of time. Although critical, liveness issues are often not covered in the open literature.

Zusammenfassung

Mikroprozessoren werden in vielen sicherheitskritischen Bereichen eingesetzt, wie beispielsweise in Automobilen oder Flugzeugen. Wir erachten daher die Korrektheit solcher Komponenten als lebenswichtig. Der Test von Prozessoren ist durch den extrem großen Zustandsraum moderner Prozessoren nur eingeschränkt möglich. Wir sind daher der Meinung, daß formale Verifikation die einzige Möglichkeit darstellt, eine Garantie zu erhalten.

Diese formale Verifikation sollte so durchgeführt werden, daß Dritten die Möglichkeit offen steht, die Korrektheit mit geringen Aufwand nachzuvollziehen. Wir wollen daher einen Beweis zur Verfügung stellen, der automatisiert überprüft werden kann. Insbesondere sollten alle kritischen Designs in Form von vier-Tupeln ausgeliefert werden: 1) das Design selbst, 2) eine Spezifikation, 3) ein manuell nachvollziehbarer Beweis, und 4) ein maschinell verifizierbarer Beweis.

Gegenstand dieser Dissertation sind Korrektheitsbeweise für komplexe Mikroprozessoren. Die Erstellung von Mirkoprozessordesigns gilt als fehleranfällig. Ein bekanntes Beispiel ist der Pentium FDIV bug [Coe95, Pra95].

In dieser Dissertation wird das Problem der Korrektheit von Hardware streng formal behandelt. Die Designs beinhalten Prozessoren mit aktuellen Scheduling Verfahren, wie beispielsweise dem Tomasulo Scheduler aus [Tom67] und spekulativer Ausführung. Im Gegensatz zu weiten Teilen der bestehenden Literatur sind die Designs auf Gatter-Ebene spezifiziert. Insbesondere werden einige der Designs für die XILINX FPGA Serie synthetisiert.

Die Designs haben hohe Komplexität, was sich auf die Beweise auswirkt. Im Gegensatz zu [MP95, Lei99, MP00] sind die Beweise mit dem Theorem Beweissystem PVS verifiziert. Wir geben in dieser Dissertation nicht den originalen PVS Beweis an, sondern versuchen einen nachvollziehbaren Beweis in üblicher mathematischer Notation anzugeben.

Um sequentielle Maschinen zu verifizieren, erweitern wir die Datenkonsistenz-Invariante aus [MP00] indem wir einen “korrekten Wert” eines Implenentation Registers wie beispielsweise $IR.2$ definieren. Gegeben die

Korrektheit der funktionalen Komponenten, wie beispielsweise der ALU, erlaubt uns dies den Beweis der Datenkonsistenz der präpariert sequentiellen Maschine in PVS fast völlig zu automatisieren. Wir argumentieren, daß die funktionellen Komponenten korrekte Ergebnisse liefern wenn sie korrekte Eingaben erhalten.

Wir erweitern das Konzept der “stall engine” aus [MP00] indem wir eine vollständig generische stall engine angeben. Im Gegensatz zu der stall engine aus [MP00], erlaubt unsere stall engine eine beliebige Anzahl von Stufen und ermöglicht es, alle Stufen unabhängig voneinander anzuhalten. Des weiteren unterstützt unsere stall engine das Entfernen von “pipeline bubbles”. Das bedeutet, daß die Stufen immer dann in Betrieb sind, wenn dies die in-order Eigenschaft zuläßt. Das beinhaltet, daß “pipeline bubbles” wenn notwendig aus der Pipeline entfernt werden. Wir verifizieren die Datenkonsistenz dieser stall engine und geben Eigenschaften an, die es erlauben Laufzeitschanken zu beweisen.

Mit dieser erweiterten stall engine verbessern wir die Transformation der präpariert sequentiellen Maschine in die Maschine mit Pipeline indem wir ein Programm implementieren das diese Transformation automatisiert. Dies beinhaltet die Generierung von Forwarding und Interlock Schaltkreisen.

Anschließen beweisen wir die Datenkonsistenz der Maschine mit Pipeline. Dies wird dadurch erreicht, daß wir beweisen, daß die Eingaben der Pipeline Stufen korrekt sind. Damit können wir wie bei der präpariert sequentiellen Maschine argumentieren, daß die Ausgaben korrekt sind, da die funktionalen Einheiten identisch sind.

Wir geben einen generischen Ansatz zur Realisierung von spekulativer Ausführung an und stellen ein Datenkonsistenzkriterium dafür auf. Wir wenden diese Methode dann an um DLX Pipelines mit Branch Prediction und präzisen Interrupts zu implementieren und zu verifizieren. Es ist allgemein bekannt, daß beide Techniken mit spekulativer Ausführung zu implementieren sind [SP88]. Nach unserem Wissen ist dies jedoch das erste Mal, daß beide Techniken als Instanz eines generischen Mechanismus’ für spekulative Ausführung implementiert werden.

Neben den in-order Pipelines verifizieren wir die Korrektheit des Tomasulo Scheduling Algorithmus’ mit Reorder Buffer. Der Reorder Buffer bewirkt in-order Terminierung, was es erlaubt, präzise Interrupts zu implementieren. Der Korrektheitsbeweis beinhaltet die Argumente, die notwendig sind, um die Eindeutigkeit der Tags zu beweisen.

Des weiteren beweisen wir eine obere Schranke für die Ausführungszeit von Programmen auf allen Maschinen. Obwohl dies eine kritische

Eigenschaft darstellt, wird dieses Thema in oder offenen Literatur oft über-
gangen.

Contents

Chapter 1

Introduction

1.1 Formal Verification of Microprocessors

NOWADAYS, microprocessors are in use in many safety-critical environments, such as cars or planes. We therefore consider the correctness of such components as a matter of vital importance.

Verifying the correctness of microprocessors used to be done by extensive tests. However, the state space of modern microprocessors is huge and tests never attain full coverage, especially for 64-bit designs. We therefore think formal verification is the sole way to obtain a guarantee.

This formal verification should be done such that any third party is able to verify the correctness with low effort, i.e., we aim to provide a proof of correctness that can be checked mechanically. In particular, we think that all critical designs should be delivered in form of a four-tuple: 1) the design itself, 2) a specification, 3) a human-readable proof, and 4) a machine-verified proof. Moreover, we think that there will be a considerable market for such four-tuples.

Let us motivate why we distinguish human-readable proofs and machine-readable proofs and why we demand for both. This is not a common demand. In industrial environments, low-effort but automatized verification is preferred.

However, proofs written for theorem proving systems tend to be hard to read. This becomes worse the higher the grade of automatization of the theorem proving system is. We think that this leads to two drawbacks: Without a human-readable proof, one completely depends on the soundness of the theorem proving system. This includes that one depends on the clarity and accuracy of the specification language of the theorem proving system.

The second drawback is that automatized design verification is of no aid in understanding the designs. In contrast to that, we have experienced that writing proofs, in particular the human-readable proofs, is producing generic theories and design approaches previously unknown. We therefore claim that providing human-readable proofs will aid automatizing the actual design process, since generic theories allow for the development of non-specialized tools with diversified use.

In this thesis, we present proofs of correctness for complex microprocessors. Designing microprocessors is considered an error-prone process. Due to the complexity of the designs, errors often remain undiscovered even in case extensive testing is done. A well known example for this is the Pentium FDIV bug [Coe95, Pra95].

1.2 Related Work

There are many publications on the formal verification of sequential machines, e.g., Cohn verified the VIPER processor [Coh87], Joyce verified the Tamarack [Joy88a, Joy88b], Hunt verified the FM8501 [Hun94], and Windley verified the AVM-1 [Win95].

In [HP96, PH94], Hennessy and Patterson describe a 32-bit RISC architecture, the DLX, which serves as basis for many microprocessor verification projects. In [MP95], Mueller and Paul describe sequential DLX designs at gate level, including a machine with precise interrupts.

The formal verification of a pipelined processor is reported in [BS89]: Bickford and Srivas verify a three stage DLX-like RISC processor. In [LO96], Levitt and Olukotun verify a five-stage DLX pipeline by transforming it back into a sequential machine by removing stalling and rollback logic.

In [Hos00], Hosabettu verifies both in-order and out-of-order DLX implementations that are not synthesizeable. The pipelined implementation has a trivial stalling logic. The verification is done using the completion function approach and PVS.

Further literature on the verification of pipelined machines is [LO96], which covers automatic verification of pipelined microprocessors, [BM96] provides a manual proof of a DLX pipeline, Burch, Dill [BD94] verify a very simple pipeline. Henzinger et.al. [HQR98] use refinement mappings in order to model-check a RISC pipeline.

Besides PVS, there are more theorem proving systems that are applied for hardware verification, such as HOL [CGM86] or ACL2 [KM96]. There has been much success in verifying complete, complex systems using theorem provers [BS89, HGS99, SH99]. However, theorem proving systems always involve much manual work.

Recent papers show the correctness of complex designs or schedulers in theorem proving systems such as PVS. Hosabettu et al. [HGS99] prove both safety and liveness of Tomasulo's algorithm using PVS. Swada and Hunt [SH99] provide an ACL2 proof of a complete design implementing a Tomasulo scheduler with reorder buffer.

Henzinger et al. [HQR98] verify a simple pipelined processor using a model checker. McMillan [McM98] partly automates the proof by refinement of Tomasulo's algorithm presented in [DP97] with the help of compositional model checking. This technique is improved in [McM99b] by theorem proving methods to support an arbitrary register size and number of function units.

In the literature cited above, the complex designs are verified at very high levels of abstraction. In particular, there is even not much literature on details of actually implementing complex microprocessors. Gate-level descriptions of microprocessors usually never go beyond simple machines, with the exception of [Lei99] and [MP00]: In [Lei99], Holger Leister presents out-of-order designs and evaluates the architectures regarding hardware cost and performance. The correctness is argued using paper-and-pencil proofs but not verified by means of machine.

In [MP00], Silvia M. Mueller and Wolfgang J. Paul present gate-level designs of pipelined DLX implementations including a machine with full IEEE floating point arithmetic and interrupts. The correctness of the machines is argued as follows: The authors build a sequential machine but with the structure of a pipelined machine. This machine is called *prepared sequential machine*. The authors transform this prepared sequential machine into a pipelined machine by adding interlock and forwarding hardware. This is supported by introducing the concept of a *stall engine*. The stall engine encapsulates the logic required for generating clock enable signals for the individual pipeline stages.

The correctness of the pipelined machine is argued as follows: given the correctness of the prepared sequential machine, the authors prove the pipeline to be correct by arguing that it simulates the prepared sequential machine. This is done using a *scheduling function*. This function maps a configuration of the physical machine to a configuration of the abstract reference machine.

1.3 Contribution

In this thesis, we provide a rigorously formal approach to hardware verification. The designs presented in this thesis include state of the art schedulers, such as the Tomasulo scheduler [Tom67] and speculation. In contrast to most of the literature, the designs we provide are very close to gate level. In particular, we are synthesizing some of the designs for the XILINX FPGA series.

These designs are of high complexity, and so are the proofs. In contrast to [MP95, Lei99, MP00], the proofs are machine verified using the theorem proving system PVS [CRSS94]. However, we never present the original PVS proof in this thesis. We aim to provide proofs that come close to comprehensible paper-and-pencil proofs in the tradition of [KP95, MP95, MP00]. We aim to maintain the full formal reasoning of the PVS proofs, to the extent that the proofs are reviewable on a line-per-line basis. This resulted in several PVS proofs to be re-written due to better readability of the paper version of the proof.

In order to verify sequential machines, we extend the data consistency invariant given in [MP00] by defining a “correct value” of an implementation register such as $IR.2$. Given the correctness of functional components such as the ALU, this allows for an almost fully automated proof of the data consistency of the prepared sequential machine using PVS. We argue that the correct functional components provide correct results if given correct inputs.

We extend the stall engine concept presented in [MP00] by providing a fully generic stall engine design. In contrast to [MP00], our stall engine design supports an arbitrary number of stages and allows for stalling (and therefore clocking) all stages independently. We formally verify data consistency and liveness properties for this stall engine.

Using this extended stall engine, we can significantly improve the process of transforming the prepared sequential machine into the pipelined machine by providing a tool that does this transformation automatically. This includes the generation for forwarding and interlock hardware. In particular, the transformation of the PC environment of the DLX with Delayed PC, i.e., removing the *DPC* register, turns out to be a special case of adding forwarding.

We then prove the data consistency of the pipelined machine. We do so by showing that the inputs of the pipeline stages are correct. Using this fact, we argue the correctness of the output values as we do for the sequential prepared machine, since the functional components of the machines are identical.

We present a generic approach to speculative execution and propose a data consistency criterion for such a machine. We then apply this method in order to implement and prove DLX pipelines with branch prediction and precise interrupts. It is a well-known fact that both techniques are implemented using speculation [SP88]. However, to the best of our knowledge, implementing both techniques as an instance of a generic speculation mechanism is done for the first time.

Besides the in-order pipelines, we verify the correctness of the Tomasulo scheduling algorithm with reorder buffer as described in [KMP99]. The reorder buffer realizes in-order termination which allows implementing precise interrupts. The proof of correctness covers the arguments neccessary to show the uniqueness of the tags.

Furthermore, we rigorously prove the liveness of all machines we design, i.e., we prove that any given instruction sequence is executed within a finite amount of time. Although critical, liveness issues are often not covered in the open literature.

1.4 Organization

Chapter 2 describes basic concepts. We introduce the mathematical hardware model, and describe the implementation and verification of basic circuits, such as adders. We use these basic circuits in order to implement and verify an ALU. We then provide a formal specification of a DLX RISC microprocessor without interrupts and floating point instructions.

In chapter 3, we describe how we model the hardware of a microprocessor. We describe the extended stall engine for the prepared sequential machine. We introduce the functions used in order to model the registers, the circuits between the registers and the forwarding logic. We use this formalism in order to implement and verify a prepared sequential DLX. We also show the liveness of the prepared sequential machine.

In chapter 4, we describe how the stall engine is modified in order to get a pipelined machine. We describe how to add the forwarding and interlock hardware and prove the correctness of the pipelined machine. This comprises of both data consistency and liveness.

In chapter 5, we describe a generic approach to speculative execution. We prove its data consistency and liveness. We implement two machines as examples: the first machine guesses whether branches are taken or not. The second machine guesses whether we have an interrupt or not. We prove that this realizes precise interrupts according to the specification given in [MP00].

In chapter 6, we describe the results of verifying an out-of-order DLX with Tomasulo scheduler as presented in [Krö99].

Chapter
2

Basic Concepts

2.1 Specifying Machines

2.1.1 Mathematical Machines

THE SUBJECT of this thesis is to present a provably correct microprocessor. A microprocessor is said to be correct if it interprets a given instruction set architecture (ISA). The instruction set architecture is usually given as an informal list of registers and instructions, and a specification of the impact of these instructions on the values of the registers. The implementation of this ISA, the microprocessor, is a piece of hardware.

In order to make a formal proof of the correctness of such a processor, it is necessary to formalize the specification, the implementation, and the correctness criterion.

Mathematical machines are a common method to model the behavior of arbitrary microprocessor systems. There are different definitions of mathematical machines. In this thesis, the mathematical machine is used to specify both the microprocessor hardware and the instruction set architecture. The correctness criterion and its proof then rely on arguments on these two mathematical machines.

The model used in this thesis is similar to the synchronous transition states (STS) model used in [KP96, DP97]. In contrast to [DP97], the mathematical machines here work fully deterministic to allow direct hardware synthesis from the mathematical machine. A very similar approach is also used in [Cyr93].

Definition 2.1 ▶ *Mathematical Machine*

A mathematical machine, as used in this thesis, is a triple $M = (C, c^0, \delta)$ that consists of the following components:

- C is the set of all possible configurations of M. An element c of C is called configuration or state of the machine.
- The initial configuration c^0 is a configuration of M.
- The transition function $\delta : C \to C$ maps one configuration c^T to its successor c^{T+1}.

The sequence c^0, c^1, ... of configurations is called computation of M. The configuration c^T is called configuration in cycle T. The configurations of M in cycles $T \geq 1$ are defined recursively as follows:

$$c^T \quad = \quad \delta(c^{T-1})$$

In the literature, the transition function is often called next state function [Cyr93].

2.1.2 Notation

Registers Both the specification and the implementation of a microprocessor use registers. A register is a place where a value can be stored and re-read in later cycles. In terms of mathematical machines, a value of a register is part of the configuration c.

Let $\mathsf{R} = \{R_1, \ldots, R_n\}$ be a finite set of registers. Each register R can have a value within a finite domain $\mathsf{W}\ (R)$, i.e., $R_i \in \mathsf{W}\ (R_i)$.

In order to allow an easy identification of the value of a register in the configuration of a mathematical machine, all valid configurations in C are expected to be a tuple of the values of all registers:

$$C \quad = \quad W(R_1) \times W(R_2) \times \ldots \times W(R_n)$$

The value of a given register R_i can be extracted from a configuration c with a projection function φ_i. Let c be $(a_1, a_2, \ldots, a_n)$.

$$\varphi_{R_i} : C \to \mathtt{W}\,(R_i), \qquad \varphi_{R_i}(c) = a_i$$

Let $c = c^T$ be part of a computation of a mathematical machine. In this case, let R^T be a shorthand for $\varphi_R(c^T)$.

Let $c.R$ be a shorthand for the value of the projection φ_R applied to c:

$$c.R \quad := \quad \varphi_R(c)$$

In analogy to that, let $\delta.R$ be a shorthand for the restriction of a state transition function to a register value:

$$\delta.R : C \to \mathtt{W}\,(R), \qquad \delta.R = \varphi_R \circ \delta$$

Signals

◄ Definition 2.2 *Signal*

A signal s is defined as a mapping from the set of configurations into an arbitrary domain $\mathtt{W}\,(s)$:

$$s : C \to \mathtt{W}\,(s)$$

Signals are therefore a shorthand for a calculation on a given configuration.

2.1.3 Bits and Bit Vectors

In order to model gates and wiring between gates in a formal way, the theorem proving system PVS [CRSS94] provides a bit vector library. Bits are defined as a boolean value and bit strings are defined as a vector of boolean values.

◄ Definition 2.3 *Vector*

An n-dimensional vector on a domain D is a mapping from $\{i \in \mathbb{N}_0 \mid i < n\}$ into D.

Let a_n denote the component n of the vector a:

$$a_n := a(n)$$

Definition 2.4 ► *Bits and Bit Vectors* A bit is a value in the domain $\mathsf{B} = \{0,1\}$. The value 0 is called FALSE and the value 1 is called TRUE. An n-bit bit vector is an n-dimensional vector on B. The number n is called length of the bit vector. If a is an n-bit bit vector, this is denoted by:

$$a \in bvec[n]$$

There is a projection function to get a subpart of an n-bit bit vector. Let $x < n$ and $y \leq x$. The function $a[x:y]$ takes a bit vector a and returns the subvector from a_x downto a_y:

$$[x:y] : bvec[n] \longrightarrow bvec[x-y+1]$$

$$a[x:y](i) \quad := \quad a(i+y) \quad \forall 0 \leq i \leq (x-y)$$

Dots Notation Let $\circ$ be a binary operator on a set T:

$$\circ : T \times T \to T$$

Let n, a, b be nonnegative integers with $b \geq a$. Let X be an n-dimensional vector on T. The following definition is used for the common "dots notation":

$$X_a \circ X_{a+1} \circ \ldots \circ X_b \quad := \quad r_{\circ,a,b}(b,X)$$

The function $r_{\circ,a,b}$ is defined recursively as follows: Let $v[n]$ denote the set of n-dimensional vectors on T.

$$r_{\circ,a,b} : \{a,\ldots,b\} \times v[b-a+1] \to T$$

$$r_{\circ,a,b}(i,X) \quad := \quad \begin{cases} X_a & : \quad i = a \\ r_{\circ,a,b}(i-1,X) \circ X_i & : \quad \text{otherwise} \end{cases}$$

In case a is omitted, zero is assumed:

	$T=0$	$T=1$	$T=2$	$T=3$	$T=4$
A^T	0	1	0	1	1
B^T	0	0	1	1	1

Table 2.1 The computation of the example machine

$$r_{\circ,b} : \{0,\dots,b\} \times v[b+1] \to T$$

$$r_{\circ,b}(i,X) \quad := \quad \begin{cases} X_0 & : \quad i=0 \\ r_{\circ,b}(i-1,X) \circ X_i & : \quad \text{otherwise} \end{cases}$$

2.1.4 Gates

Using the definition of bits above, the basic gates such as AND and OR are defined in a obvious way: a gate like AND with two inputs and one output is a mapping on two bits:

$$AND : \mathsf{B} \times \mathsf{B} \longrightarrow \mathsf{B}$$

As an example, consider the following mathematical machine (a two bit saturating counter): It has two one bit registers $\mathsf{R} = \{A,B\}$ with $\mathsf{W}\ (A) = \mathsf{W}\ (B) = \mathsf{B}$. The configuration set C therefore is B^2. Let the transition function δ be defined as follows:

$$\begin{aligned} \delta.A(c) &= \overline{c.A} \vee c.B \\ \delta.B(c) &= c.A \vee c.B \end{aligned}$$

Let the initial configuration c_0 be $\{0,0\}$. This mathematical machine models hardware: in order to illustrate the hardware modeled by mathematical machines, the symbols from figure 2.1 are used.

The transition function δ models two OR-gates and one inverter. The configuration set models two one-bit registers. In hardware, registers usually do not have defined initial values. In order to get the initial configuration c_0, an external signal *reset* is assumed. This signal is active during

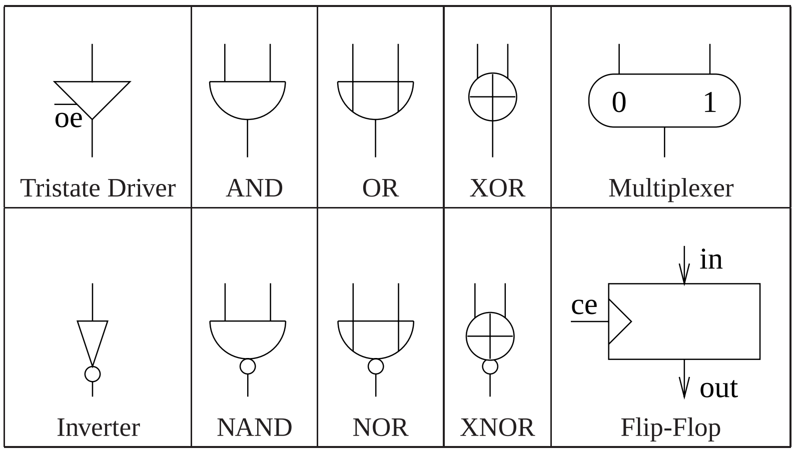


Figure 2.1 Symbols of the basic gates

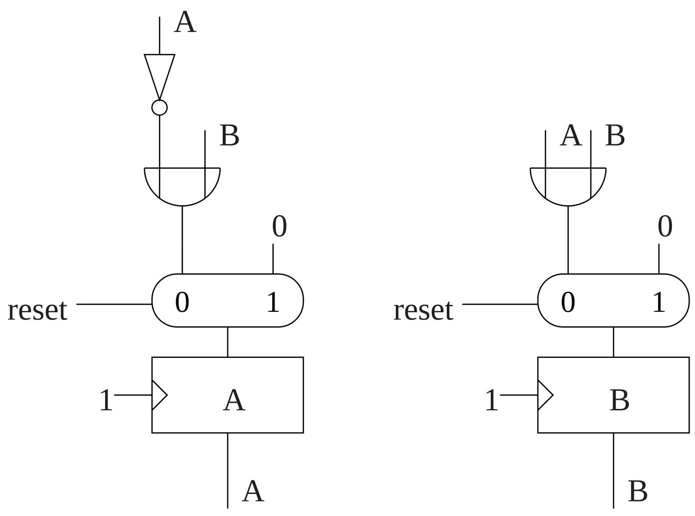


Figure 2.2 A two bit saturating counter

cycle -1. Using multiplexers, this allows calculating the initial configuration.

The hardware modeled by the mathematical machine described above is illustrated by figure 2.2. Table 2.1 lists the values of the registers A and B in the configurations c^0 to c^4.

2.1.5 Interpretations of Bit Vectors

The interpretation of a bit vector a as a binary number is a mapping from the n-bit bit vectors into $\{0,\ldots,2^n-1\}$. The mapping is denoted by $\langle a\rangle_n$. If the length of the bit vector argument is obvious in the context, just $\langle a\rangle$ is used.

$$\langle\rangle_n : bvec[n] \longrightarrow \{0,\ldots,2^n-1\}$$

$$\langle a\rangle_n \quad := \quad \sum_{i=0}^{n-1} a_i \cdot 2^i$$

The PVS bit vector library provides the function `bv2nat[n]` for this purpose. The value of this function is defined by a recursive function that takes an n-bit bit vector and an index i: the function sums up the first i addends of the sum above:

$$\langle\rangle_n^i : \{0,\ldots,n\} \times bvec[n] \longrightarrow \{0,\ldots,2^n-1\}$$

$$\langle a\rangle_n^i \quad = \quad \sum_{j=0}^{i-1} a_j \cdot 2^j$$

In PVS, this is defined using a recursion:

$$\langle a\rangle_n^i \quad := \quad \begin{cases} 0 & : \quad i=0 \\ 2^{i-1}\cdot a_{i-1} + \langle a\rangle_n^{i-1} & : \quad \text{otherwise} \end{cases}$$

It is easy to prove that both definitions are equivalent and that $\langle a\rangle_n^n = \langle a\rangle_n$ holds.

The interpretation of a bit vector a as a two's complement number is a mapping from the n-bit bit vectors into $\{-2^{n-1},\ldots,2^{n-1}-1\}$:

$$[\,]_n : bvec[n] \longrightarrow \{0,\ldots,2^n-1\}$$

$$[a]_n := -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle_{n-1}$$

The bit a_{n-1} is called *sign bit*.

This allows defining several operations on bit vectors such as addition and subtraction:

$$+, - : bvec[n] \times bvec[n] \longrightarrow bvec[n]$$

$$\begin{aligned} a+b := c \quad &\text{such that} \quad \langle c \rangle_n = \langle a \rangle + \langle b \rangle \bmod 2^n \\ a-b := c \quad &\text{such that} \quad \langle c \rangle_n = \langle a \rangle - \langle b \rangle \bmod 2^n \end{aligned}$$

A similar definition is used for operations on a bit vector and an integer:

$$+, - : bvec[n] \times \mathbb{Z} \longrightarrow bvec[n]$$

$$\begin{aligned} a+b := c \quad &\text{such that} \quad \langle c \rangle_n = \langle a \rangle + b \bmod 2^n \\ a-b := c \quad &\text{such that} \quad \langle c \rangle_n = \langle a \rangle - b \bmod 2^n \end{aligned}$$

An unary minus on bit vectors is defined as follows:

$$- : bvec[n] \longrightarrow bvec[n]$$

$$-a := c \quad \text{such that} \quad \langle c \rangle_n = -\langle a \rangle \bmod 2^n$$

The function *zero_extend*$_k$ extends a given n-bit bit vector to $k \geq n$ bits by adding zeros:

$$zero\_extend_k : \; bvec[n] \longrightarrow bvec[k]$$

$$zero\_extend_k(a)_i \quad = \quad \begin{cases} a_i & : \quad i < n \\ 0 & : \quad \text{otherwise} \end{cases}$$

The function *sign_extend*$_k$ extends a given n-bit bit vector to $k \geq n$ bits by adding the sign bit:

$$sign\_extend_k : \; bvec[n] \longrightarrow bvec[k]$$

$$sign\_extend_k(a)_i \quad = \quad \begin{cases} a_i & : \quad i < n \\ a_{n-1} & : \quad \text{otherwise} \end{cases}$$

2.2 Basic Circuits

2.2.1 Binary Trees

◀ Definition 2.5 *Binary Tree Circuit*

Let n be a power of two, i.e., $n = 2^k$, $k \in \mathbb{N}_0$. Let $\circ : T \times T \longrightarrow T$ be a dyadic function that is associative. Let T denote a set and let $v[n]$ denote the set of n-dimensional vectors on T.

The binary tree is implemented as follows:

$$btree_{\circ,k} : v[2^k] \longrightarrow T$$

$$btree_{\circ,k}(X) \quad = \quad \begin{cases} X_0 & : \quad k = 0 \\ btree_{\circ,k-1}(X(0),\ldots,X(2^{k-1}-1))\circ & : \quad \text{otherwise} \\ btree_{\circ,k-1}(X(2^{k-1}),\ldots,X(2^k-1)) & \end{cases}$$

◀ Lemma 2.1

The binary tree circuit $btree_{\circ,k} : v[n] \longrightarrow T$ calculates the following function:

$$btree_{\circ,k}(X) \quad = \quad X_0 \circ X_1 \circ \ldots \circ X_{n-1}$$

PROOF

This is shown by induction on k. For $k = 0$, the claim is obviously true. For $k+1$, the claim is:

$$btree_{\circ,k+1} = X(0) \circ \ldots \circ X(2^{k+1}-1)$$

By definition of *btree*, this is equivalent to:

$$btree_{\circ,k}(X(0),\ldots,X(2^k-1)) \circ btree_{\circ,k}(X(2^k),\ldots,X(2^{k+1}-1)) =$$

$$X(0) \circ \ldots \circ X(2^{k+1}-1)$$

By the induction premise for both *btree* instances, this is equivalent to:

$$(X(0) \circ \ldots \circ X(2^k-1)) \circ (X(2^k) \circ \ldots \circ X(2^{k+1}-1))$$

$$= X(0) \circ \ldots \circ X(2^{k+1}-1)$$

This is shown by induction using that $\circ$ is associative.

2.2.2 Zero Tester

Let n be a power of two. The zero tester is implemented as follows:

$$zerotester : bvec[n] \longrightarrow \mathsf{B}$$

$$zerotester(a) \quad = \quad \overline{btree_{OR}(a)}$$

Lemma 2.2 ► The zero tester calculates the following function:

$$zerotester(a) \quad = \quad (\forall i : \overline{a_i})$$

This is shown by induction on n using lemma 2.1.

2.2.3 Equality Tester

Using the zero tester, an equality tester is constructed as follows:

$$equalitytester : bvec[n] \times bvec[n] \longrightarrow \mathsf{B}$$

$$equalitytester(a,b) \quad = \quad zerotester(a \oplus b)$$

Lemma 2.3 ► The equality tester is correct:

$$equalitytester(a,b) \quad = \quad (a = b)$$

The correctness is shown easily with lemma 2.2.

2.2.4 Parallel Prefix

Definition 2.6 ► *Parallel Prefix* Let T denote a set and let $v[n]$ denote the set of n-dimensional vectors on T. Let $\circ : T \times T \longrightarrow T$ be an associative dyadic function. The n-fold generic parallel prefix circuit $PP_{\circ,n} : v[n] \longrightarrow v[n]$ calculates the following function:

$$PP_{\circ,n}(X)_i = X_0 \circ X_1 \circ \ldots \circ X_i \qquad i \in \{0, \ldots, n-1\}$$

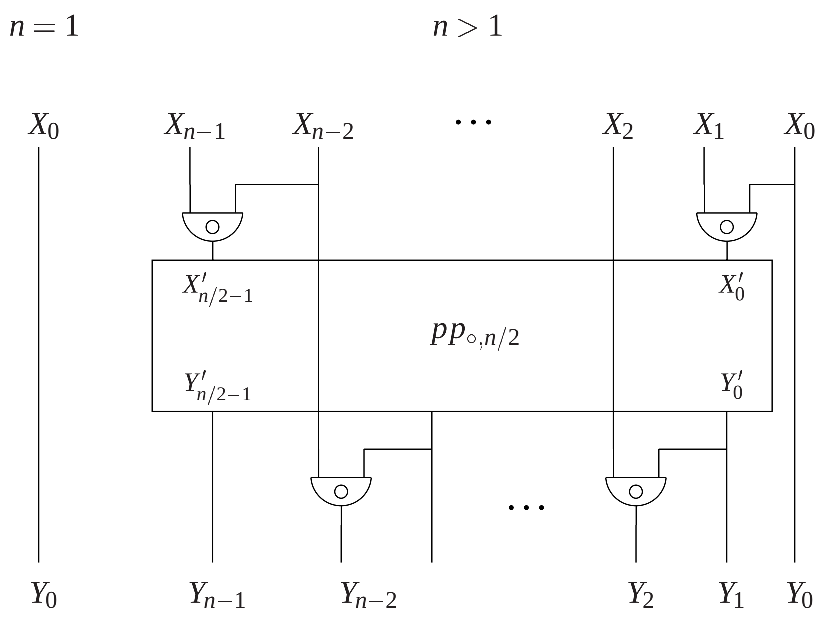


Figure 2.3 The recursive specification of an n-fold parallel prefix circuit

The parallel prefix circuit is implemented by means of a recursive definition (figure 2.3). Let n be a power of two, i.e., $n = 2^K$ with $K \in \mathbb{N}$, and let $X \in v[2^K]$ be the inputs of the circuit.

The function $ppX'_\circ$ calculates the inputs X'_0 to $X'_{n/2-1}$ for the next recursion step. The recursion depth is given by the first parameter K:

$$ppX'_\circ : \mathbb{N} \times v[2^K] \longrightarrow v[2^{K-1}]$$

$$ppX'_\circ(K,X)_i := X(2\cdot i) \circ X(2\cdot i+1)$$

Given those inputs, the function $ppY_\circ$ calculates the outputs Y_0 to Y_{n-1}. As above, the recursion depth is given by the first parameter K:

$$ppY_\circ(K,X)_i = \begin{cases} X_0 & i=0 \\ ppY_\circ(K-1, ppX'_\circ(K,X))_{\frac{i-1}{2}} & \text{odd } i \\ ppY_\circ(K-1, ppX'_\circ(K,X))_{\frac{i}{2}-1} \circ X_i & \text{even } i \end{cases}$$

The outputs of the parallel prefix circuit are the values Y_0 to Y_{n-1}:

$$pp_\circ(X)_i \quad := \quad ppY_\circ(K,X)_i$$

The parallel prefix circuit is correct: ◀ Theorem 2.4

$$pp_\circ(X)_i \quad = \quad X_0 \circ X_1 \circ \ldots \circ X_i$$

In order to prove theorem 2.4, the definition *pp1* is used. The first parameter defines the number of inputs, the second parameter is the index of the output, the third parameter is the input vector.

$$pp1 : \mathbb{N} \times \{0, \ldots, 2^K - 1\} \times v[2^K] \longrightarrow T$$

$$\mathit{pp1}(K,i,X) \quad := \quad \begin{cases} X_0 & : \quad i = 0 \\ \mathit{pp1}(K,i-1,X) \circ X_i & : \quad \text{otherwise} \end{cases}$$

Lemma 2.5 ► This definition is equivalent to $PP_{\circ,n}$, which is an easy proof by induction:

$$\mathit{pp1}(K,i,X) \quad = \quad PP_{\circ,n}(X)_i$$

Lemma 2.6 ► If i is odd, applying *pp1* to X'_0 to $X'_{(i-1)/2}$ is equivalent to applying *pp1* to X_0 to X_i:

$$\mathit{pp1}(K-1,(i-1)/2,ppX'(K,X)) \quad = \quad \mathit{pp1}(K,i,X)$$

If i is even and not zero, appending X_i to the sequence above on the left hand side produces the desired result:

$$\mathit{pp1}(K-1,i/2-1,ppX'(K,X)) \circ X_i \quad = \quad \mathit{pp1}(K,i,X)$$

PROOF This is shown by induction on i. For $i = 0$, the claim is obvious. For odd $i+1$, the claim is:

$$\mathit{pp1}(K-1,i/2,ppX'(K,X)) \quad = \quad \mathit{pp1}(K,i+1,X)$$

By definition of *pp1*, this is equal to:

$$\mathit{pp1}(K-1,i/2-1,ppX'(K,X)) \circ ppX'(K,X)(i/2) \quad = \quad \mathit{pp1}(K,i+1,X)$$

Unfolding $ppX'_\circ$, this results in:

$$\mathit{pp1}(K-1,i/2-1,ppX'(K,X)) \circ (X_i \circ X_{i+1}) \quad = \quad \mathit{pp1}(K,i+1,X)$$

Since $\circ$ is associative, this is equal to:

$$(\mathit{pp1}(K-1,i/2-1,ppX'(K,X)) \circ X_i) \circ X_{i+1} \quad = \quad \mathit{pp1}(K,i+1,X)$$

This is shown by unfolding the definition of *pp1* on the right-hand side and by the induction hypothesis for even i.

For even $i+1$, the claim is shown by the definition of *pp1* and the induction premise for odd i.

◂ Lemma 2.7

The parallel prefix circuit computes *pp1*.

$$\forall\, 0 \le k \le K,\ X \in v[2^k],\ 0 \le i \le 2^k : \quad ppY(k,i,X) = pp1(k,i,X)$$

PROOF

This is shown by induction on k. For $k=0$, the claim is obvious. For $k+1$, and after definition unfolding, the claim is:

$$ppY(k+1,i,X) \overset{!}{=} pp1(k+1,i,X)$$

For $i=0$, the claim is shown by definition unfolding. If i is odd, the claim is:

$$ppY(k,(i-1)/2,ppX'(k+1,X)) \overset{!}{=} pp1(k+1,i,X)$$

This is shown using the induction hypothesis and lemma 2.6.

If i is even, the claim is:

$$ppY(k,i/2-1,ppX'(k+1,X)) \circ X_i \overset{!}{=} pp1(k+1,i,X)$$

This is shown using the induction premise and lemma 2.6.

2.2.5 Adders

The definitions used in this section are taken from the PVS bit vector library. In order to define adders, the two functions *cout* and *sum* are used. Using both functions, one gets a fulladder.

The functions take three input bits a, b, and cin. The function *cout* calculates the carry-out bit of the adder, the function *sum* calculates the sum bit.

$$cout, sum : \mathsf{B} \times \mathsf{B} \times \mathsf{B} \to \mathsf{B}$$

The functions are defined using XOR, AND, and OR gates as follows:

$$\begin{aligned} cout(a,b,cin) &:= (a \wedge b) \vee ((a \oplus b) \wedge cin) \\ sum(a,b,cin) &:= a \oplus b \oplus cin \end{aligned}$$

Definition 2.7 ► *Carry Bits* Let x and y denote two n-bit bit vectors and cin a single bit. The carry bits $c(0)$ to $c(n-1)$ are defined as follows:

$$c(i) \quad := \quad \begin{cases} cout(x_0, y_0, cin) & : \quad i = 0 \\ cout(x_i, y_i, c(i-1)) & : \quad \text{otherwise} \end{cases}$$

Definition 2.8 ► *Adder* An n-bit adder implements the following function add on two n-bit bit vectors x, y: The function is defined using the addition on bit vectors as defined in section 2.1.5.

$$add : bvec[n] \times bvec[n] \longrightarrow bvec[n]$$

$$add(x,y) = x + y$$

Let $c(i)$ denote the i-th carry bit as in definition 2.7. An n-bit adder with carry-in and carry-out implements the following function $addc$ on two n-bit bit vectors x, y and the carry-in bit cin:

$$addc : bvec[n] \times bvec[n] \times \mathsf{B} \longrightarrow bvec[n] \times \mathsf{B}$$

$$\begin{aligned} addc(x,y,cin) &:= (result, cout) \\ \text{with } result &:= (x + y + \langle cin \rangle), \\ cout &:= c(n-1) \end{aligned}$$

The carry chain adder is implemented as follows:

$$cc : bvec[n] \times bvec[n] \times \mathsf{B} \longrightarrow bvec[n] \times \mathsf{B}$$

$$\begin{aligned} cc(x,y,cin) &:= (result, cout) \\ i \in \{0,\ldots,n-1\} : \ result(i) &:= \begin{cases} sum(x_0, y_0, cin) & : \quad i = 0 \\ sum(x_i, y_i, c(i-1)) & : \quad \text{otherwise} \end{cases} \\ cout &:= c(n-1) \end{aligned}$$

Lemma 2.8 ► The carry chain adder is correct according to definition 2.8.

The proof for this lemma is already in the PVS bit vector library.

2.2.6 Verification of a Carry Lookahead Adder

The *carry lookahead adder* provides both low hardware cost and low depth [KP95].

Let $c(0)$ to $c(n-1)$ denote the carry bits as defined in definition 2.7 for the addition of two n-bit bit vectors a and b and the carry-in bit cin. The idea is to use a parallel prefix calculation (definition 2.6) in order to calculate the carry bits $c(i)$. Using these bits, the carry lookahead adder is realized as follows:

$$cla(a,b,cin) = (result, cout)$$

$$\text{with } result(i) = a(i) \oplus b(i) \oplus \begin{cases} cin & : \; i=0 \\ c(i-1) & : \; \text{otherwise} \end{cases}$$

$$\text{and } cout = c(n-1)$$

The inputs (g_i, p_i) and the associative function $\circ$ used for the parallel prefix circuit are taken from [MP00]:

$$\begin{aligned} p_i &:= a(i) \oplus b(i) \\ g_i &:= \begin{cases} ((a(i) \oplus b(i)) \wedge cin) \vee (a(0) \wedge b(0)) & : \; i=0 \\ a(i) \wedge b(i) & : \; \text{otherwise} \end{cases} \end{aligned}$$

$$(g_1, p_1) \circ (g_2, p_2) \quad := \quad (g_2 \vee g_1 \wedge p_2,\; p_1 \wedge p_2)$$

The proof that $\circ$ is associative is trivial in PVS.

Let $G(i)$ and $P(i)$ denote the outputs of the parallel prefix circuit, i.e., according to theorem 2.4 (correctness of the parallel prefix circuit) this is:

$$\begin{aligned} G(i) &= ((g_0, p_0) \circ \ldots \circ (g_i, p_i)).g \\ P(i) &= ((g_0, p_0) \circ \ldots \circ (g_i, p_i)).p \end{aligned}$$

We will now show that we get the carry bits by calculating $G(i)$ as above.

◀ Lemma 2.9

The carry bits c are G.

$$c = G$$

The proof for this claim is already given in [MP00]. We verify it using PVS.

PROOF The proof proceeds by induction on i. For $i = 0$, the claim follows by definition unfolding.

For $i+1$, the claim after applying theorem 2.4 (correctness of the parallel prefix circuit) is:

$$c(i+1) \quad = \quad ((g_0, p_0) \circ \ldots \circ (g_{i+1}, p_{i+1})).g$$

By definition of $\circ$, this is equivalent to:

$$c(i+1) \quad = \quad g_{i+1} \vee ((g_0, p_0) \circ \ldots \circ (g_i, p_i)).g \wedge p_{i+1}$$

By the induction hypothesis, this is equivalent to:

$$c(i+1) = g_{i+1} \vee c(i) \wedge p_{i+1}$$

By definition of the carry bits, this is equivalent to:

$$\begin{aligned} & a(i+1) \wedge b(i+1) \vee ((a(i+1) \oplus b(i+1)) \wedge c(i)) \\ = \quad & g_{i+1} \vee c(i) \wedge p_{i+1} \end{aligned}$$

QED This is shown by definition of g_{i+1} and p_{i+1}.

2.3 Verification of an ALU

2.3.1 Specification

An ALU (arithmetic logical unit) performs operations such as addition, subtraction, comparisons, and bitwise operations such as AND, OR, and XOR.

The ALU takes two 32-bit bit vector operands a and b and additional five bits f. These bits f control the operation performed by the ALU. The ALU returns the result bit vector and an additional bit *ovf* that is set iff an overflow occurred during an addition or subtraction.

f[4]	f[3]	f[2]	f[1]	f[0]	Function
0	*	*	0	*	$a \ll b[4:0]$
0	*	*	1	0	$a \gg b[4:0]$
0	*	*	1	1	$a \gg_a b[4:0]$
1	0	0	0	0	$a+b$ with overflow test
1	0	0	0	1	$a+b$ without overflow test
1	0	0	1	0	$a-b$ with overflow test
1	0	0	1	1	$a-b$ without overflow test
1	0	1	0	0	$a \wedge b$
1	0	1	0	1	$a \vee b$
1	0	1	1	0	$a \oplus b$
1	0	1	1	1	$b[0:15]0^{16}$
1	1	0	0	0	return zero
1	1	0	0	1	$[a] > [b]$? 1 : 0
1	1	0	1	0	$a = b$? 1 : 0
1	1	0	1	1	$[a] \geq [b]$? 1 : 0
1	1	1	0	0	$a < b$? 1 : 0
1	1	1	0	1	$a \neq b$? 1 : 0
1	1	1	1	0	$[a] \leq [b]$? 1 : 0
1	1	1	1	1	return one

Table 2.2 ALU functions

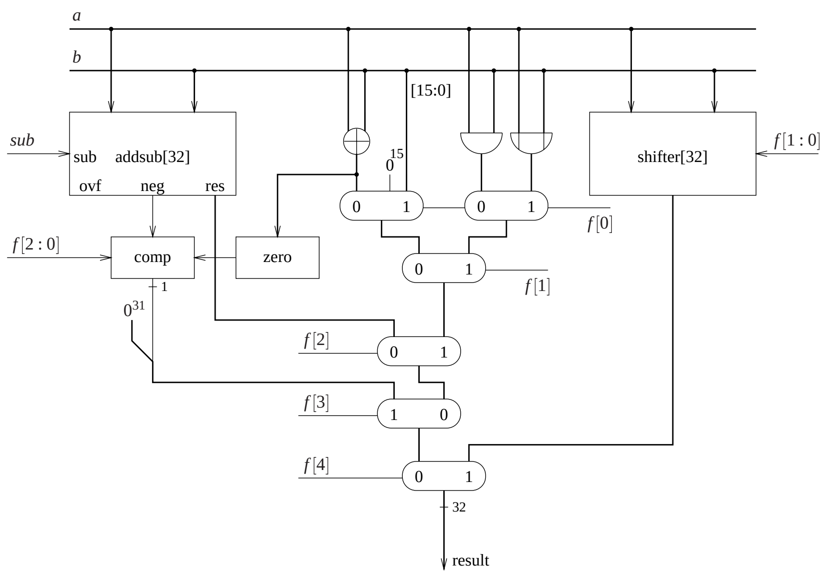


Figure 2.4 The ALU implementation

Table 2.2 lists the operations performed by the ALU. It is taken from [MP95] with small modifications. The notation $a \ll b$ is used to denote a left shift of a with shift distance b, $a \gg b$ denotes a logic right shift of a with shift distance b, $a \gg_a b$ denotes an arithmetic right shift of a with shift distance b.

Overflow Let $\circ$ be an addition or subtraction, i.e., $\circ \in \{+,-\}$. An overflow indicates that the result of $[a] \circ [b]$ is not in the range of the 32-bit two's complement numbers. Let $a \in T_n$ denote that a is in the range of the n-bit two's complement numbers.

Table 2.2 does not provide overflow test and comparisons for unsigned binary numbers in contrast to most microprocessors processors such as the MIPS CPUs or the Intel Pentiums [KH92, Int95b]. We do so in order to maintain the instruction set used in [MP00].

2.3.2 Implementation

Figure 2.4 [MP95] gives an overview of the ALU implementation. Depending on the signals f, the result from the appropriate unit is taken.

The addsub unit takes the operands a and b and one extra input bit *sub*, which indicates whether to do an addition or a subtraction. If *sub* is set, the unit performs a subtraction. The *sub* bit is calculated as follows:

$$sub \quad := \quad \overline{f_4 \wedge \overline{f_3} \wedge \overline{f_2} \wedge \overline{f_1}}$$

The unit returns the result bit vector, and the flag bits *ovf* and *neg*. The *ovf* bit is supposed to indicate the overflow condition described in the section above. The *neg* bit is used for the comparison operations and indicates that $[a] \circ [b]$ is below zero.

The addsub unit is realized as follows: Let $op1$ and $op2$ denote the operands. The second operand is inverted in case of a subtraction.

$$op1 := a$$

$$op2 := b \oplus (sub^{32})$$

This is justified by the following lemma:

◄ Lemma 2.10

For all bitvectors a, inverting and incrementing a implements the unary minus on bitvectors.

$$(a \oplus (1, \ldots, 1)) + 1 = -a$$

This is shown in the PVS bit vector library.

Using the operands and the sub bit the result is calculated by an adder. In the following, the carry lookahead adder (section 2.2.6) is used. However, there is also an implementation and proof of a compound adder, as described in [MP00], in the PVS tree in order to allow cycle time vs. hardware cost tradeoffs. The implementation and the proof are omitted here.

The *sub* bit is passed as carry-in bit to the adder. This realizes the incrementation in case of a subtraction.

$$addsub(a, b, sub) := (result, ovf, neg)$$

$$\text{with } result = cla(op1, op2, sub).result$$

The bits ovf and neg are calculated as follows:

$$\begin{aligned} neg &= cla(op1, op2, sub).cout \oplus op1[31] \oplus op2[31] \\ ovf &= neg \oplus cla(op1, op2, sub).result[31] \end{aligned}$$

Lemma 2.11 ▶ The calculation of *result* in the addsub unit is correct.

$$addsub(a, b, sub).result = a \circ b$$

This is shown using lemma 2.10 and 2.9.

Lemma 2.12 ▶ The calculation of the *ovf* signal in the addsub unit is correct.

$$addsub(a, b, sub).ovf = ([a] \circ [b]) \notin T_n$$

Lemma 2.13 ▶ The calculation of the *neg* signal in the addsub unit is correct.

$$addsub(a, b, sub).neg = ([a] \circ [b]) < 0$$

A proof for the lemmas 2.12 and 2.13 can be found in [MP00]. The full proof is also in the PVS tree.

An equality tester is realized by testing if $a \oplus b$ is zero. Using the output signal *eq* of the zero tester and the signals *ovf* and *neg* from the addsub unit, the comp unit makes the comparisons as follows:

$$comp : bvec[5] \times \mathsf{B} \times \mathsf{B} \longrightarrow \mathsf{B}$$

$$comp(f, neg, eq) = (f_2 \wedge neg) \vee (f_1 \wedge eq) \vee (\overline{eq} \wedge \overline{neg} \wedge f_0)$$

Using the lemmas 2.2, 2.3, 2.12, and 2.13, the correctness of the comp unit is shown.

Lemma 2.14 ▶ The ALU is correct.

This is shown by a case-split on the operation code f using the lemmas above. The correctness of the shifter is assumed.

2.4 Specifying the Reference Machine

2.4.1 DLX Architecture

The reference machine used for all designs in this thesis is the DLX [HP96, SK96]. However, the DLX architecture serves as an example only. The algorithms and proof method presented here does not depend on any properties of the DLX architecture.

The DLX architecture is a load/store architecture with support for integer and floating point arithmetic. The DLX instruction set (appendix B) is a RISC instruction set and is similar to the MIPS instruction set.

The DLX architecture provides three register files:

- The **general purpose register file** (GPR) consists of 32 integer registers (R_0,...,R_{31}), each of which is 32 bits wide. The register R_0 is defined to be always zero. The general purpose registers are used for all integer operations and memory addressing purposes.
- The **floating point register file** (FPR) consists of 32 single precision floating point registers (FGR_0,...,FGR_{31}), each of which is 32 bits wide. These registers can also be accessed as 16 double precision floating point registers (FPR_0, FPR_2,...,FPR_{30}), each of which is 64 bits wide. The register FPR_0 is mapped onto the single precision registers FGR_0 and FGR_1, and so on:

  $$FPR_0(i) \quad = \quad \begin{cases} FGR_0(i) & : \quad i < 32 \\ FGR_1(i-32) & : \quad i \geq 32 \end{cases}$$

  The floating point registers are used by FPU (floating point unit) instructions only.
- The **special purpose register file** (SPR) consists of several registers needed for special purposes such as flags and masks. An example is the IEEE floating point flags register.

2.4.2 Configuration of an Integer DLX with Delayed PC

The configuration set of the DLX specification machine consists of the visible registers (register files RF), the program counter (PC) registers, and the main memory (MEM) of the machine:

$$C_{DLX} = \mathtt{W}\ (RF) \times \mathtt{W}\ (RPC) \times \mathtt{W}\ (MEM)$$

The DLX implementation presented in chapter 3 implements integer operations only and no interrupts. The floating point and special purpose registers are not needed therefore. The machine is called DLX$_\sigma$.

$$RF = \{GPR[0], \ldots, GPR[31]\}$$
$$\mathtt{W}\ (GPR[i]) = \mathtt{B}^{32}$$

In order to implement pipelining at a high performance level without the need for a branch prediction mechanism, the DLX implemented in this thesis uses the concept of *delayed PCs* [MPK00, MP00]: all modifications to the PC register are delayed by one instruction, not just taken branches. This is realized by buffering the PC register in a register called DPC ("delayed PC"). The Delayed PC technique is provably equivalent to the delayed branch semantics. The delayed branch semantics is, for example, used in the MIPS [KH92], the SPARC [SPA92] and the PA-RISC [Hew94] instruction set.

In order to implement the Delayed PC technique, two PC registers are required: *DPC*, the delayed PC, and PC':

$$RPC = \{DPC, PC'\}$$
$$\mathtt{W}\ (DPC) = \mathtt{W}\ (PC') = \mathtt{B}^{32}$$

The main memory of the DLX specification machine consists of 2^{30} memory cells, each of which is 32 bits wide. That accounts for a total of four gigabytes RAM:

$$MEM = \{MEM[0], \ldots, MEM[2^{30} - 1]\}$$
$$\mathtt{W}\ (MEM[i]) = \mathtt{B}^{32}$$

2.4.3 Initial Configuration

The GPR registers and the main memory of the DLX$_\sigma$ machine are initialized with arbitrary but fixed values. The PC registers *DPC* and PC' are initialized as follows [MPK00]:

$$\begin{aligned} c_0.DPC &= 0 \\ c_0.PC' &= 4 \end{aligned}$$

2.4.4 Transition Function

The DLX$_\sigma$ machine provides control instructions (conditional branch and jump), ALU instructions such as add and compare, and the memory instructions load and store. The instruction that is to be executed is encoded in a 32-bit instruction word. This instruction word is fetched from the instruction memory *IM*, which is assumed to be constant in this thesis. The instruction memory is not part of the configuration therefore.

Let the signal *I* denote the instruction word fetched. The address used to fetch *I* is taken from the register DPC, as required by the Delayed PC technique [MPK00]:

$$I(c) = IM(c.DPC)$$

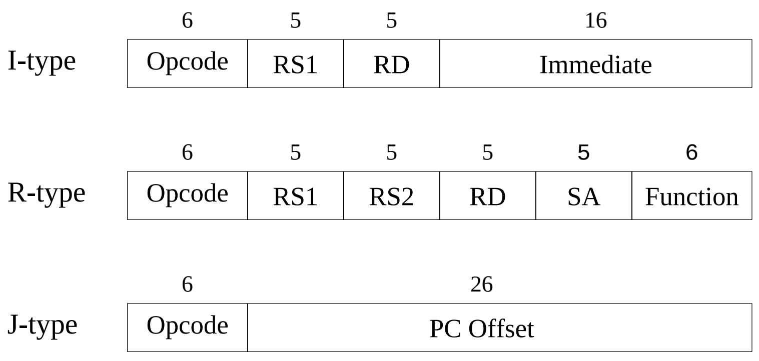


Figure 2.5 Integer instruction formats of the DLX

The DLX architecture provides three instruction formats for integer instructions (figure 2.5): the I-type format provides a 16-bit immediate constant and two register addresses, the R-type format provides three register addresses, a 5-bit immediate constant and an additional 6-bit function code. The J-type format provides a 26-bit immediate constant, which is used as PC offset.

The coding of the instructions is given in appendix B. In order to decode the instruction word I, the following functions are used: The functions *I_rtype*, *I_jtype*, *I_itype* indicate an R-type, J-type, and I-type instruction, respectively:

$$\begin{aligned} I\_rtype(I) &= (/I_{31} \wedge /I_{30} \wedge /I_{29} \wedge /I_{28} \wedge /I_{27} \wedge /I_{26}) \vee \\ &\quad (/I_{31} \wedge I_{30} \wedge /I_{29} \wedge /I_{28} \wedge /I_{27} \wedge I_{26}) \\[1em] I\_jtype(I) &= (/I_{31} \wedge /I_{30} \wedge /I_{29} \wedge /I_{28} \wedge I_{27}) \vee \\ &\quad (I_{31} \wedge I_{30} \wedge I_{29} \wedge I_{28} \wedge I_{27}) \\[1em] I\_itype(I) &= \overline{I\_jtype(I)} \wedge \overline{I\_rtype(I)} \end{aligned}$$

The function *I_ID* extracts the index of the destination register from the instruction word:

$$I\_RD(I) = \begin{cases} I[20:16] & : \; I\_itype(I) \\ I[15:11] & : \; I\_rtype(I) \\ 0^5 & : \; \text{otherwise} \end{cases}$$

The functions I_RS1 and I_RS2 extract the index of the first and second operand from the instruction word, respectively:

$$\begin{aligned} I\_RS1(I) &= I[25,21] \\ I\_RS2(I) &= I[20,16] \end{aligned}$$

The function *I_immediate* extracts the immediate constant from the instruction word:

$$I\_immediate(I) = \begin{cases} sign\_extend_{32}(I[15,0]) & : \; I\_itype(I) \\ zero\_extend_{32}(I[10,6]) & : \; I\_rtype(I) \\ sign\_extend_{32}(I[25,0]) & : \; I\_jtype(I) \\ 0 & : \; \text{otherwise} \end{cases}$$

This allows defining the values of the source operands: the integer DLX instructions can have up to two source operands. Let *op1* and *op2* denote the values of these operands. If the address of the operand is zero, the value of the operand is zero by convention:

$$op1(c) = \begin{cases} 0 & : \; I\_RS1(I) = 0 \\ c.GPR[I\_RS1(I)] & : \; \text{otherwise} \end{cases} \tag{2.1}$$

$$op2(c) = \begin{cases} 0 & : \; I\_RS2(I) = 0 \\ c.GPR[I\_RS2(I)] & : \; \text{otherwise} \end{cases} \tag{2.2}$$

Branch Mechanism The DLX architecture provides two instructions to modify the PC' register: the branch instructions test a given register for a condition and add the offset given as immediate constant if the condition holds; the jump instructions always set the PC' register to the given value.

In order to determine the instruction coded by an instruction word I, a boolean function is defined for each instruction. The equations for these functions are generated from the instruction set in appendix B and are in the PVS tree. A list of the functions is also in appendix B.

The functions $I\_j(I)$ and $I\_jr(I)$ return true iff the instruction is a jump instruction. In case of $I\_j(I)$, the immediate constant is used as offset to the PC, in case of $I\_jr(I)$ the jump target is the value of the first operand. The function $I\_branch(I)$ is used to detect a branch. If the instruction is a branch, $I\_branch\_eq(I)$ indicates that the branch is to be taken if the operand is zero. If $I\_branch\_eq(I)$ does not hold, the branch is to be taken if the operand is not zero.

Let $GPRa$ be the value of the operand. The function $bjtaken(I, GPRa)$ is true iff the given instruction I is a taken branch or jump:

$$bjtaken : bvec[32] \times bvec[32] \longrightarrow \mathsf{B}$$

$$\begin{aligned} bjtaken(I, GPRa) \quad = \quad & I\_j(I) \vee I\_jr(I) \vee (I\_branch(I) \wedge \\ & (I\_branch\_eq(I) \oplus (GPRa = 0))) \end{aligned}$$

The function $nextpc$ calculates the new value of PC' given the instruction word I, the value of the first operand $GPRa$ and the old value of PC':

$$nextpc(I, GPRa, PC') =$$

$$\begin{cases} GPRa & : \quad bjtaken(I, GPRa) \wedge I\_jr(I) \\ PC' + I\_immediate(I) & : \quad bjtaken(I, GPRa) \wedge \overline{I\_jr(I)} \\ PC' + 4 & : \quad \text{otherwise} \end{cases}$$

$$\delta.PC'(c) \quad = \quad nextpc(I, op1(c), c.PC') \tag{2.3}$$

According to the Delayed PC technique, the new value for DPC is the old value of PC':

$$\delta.DPC(c) \quad = \quad c.PC' \tag{2.4}$$

In case of a jump and link instruction, which is indicated by $I\_link(I)$, the old value of PC' plus four is stored in the destination register:

$$\delta.GPR[I\_RD(I)](c) \quad = \quad c.PC' + 4$$

ALU Instructions The function *ALUfunction(I)* extracts the ALU function code from the instruction word. The ALU function codes are given in table 2.2, page 23.

$$ALUfunction(I): \mathsf{B}^{32} \longrightarrow \mathsf{B}^{5}$$

$$ALUfunction(I) = \begin{cases} 1 \quad I_{30} \quad I[28:26] & : I\_itype(I) \\ I_5 \quad I_3 \quad (I_2 \wedge I_5) \quad I[1:0] & : I\_rtype(I) \\ 0^5 & : \text{otherwise} \end{cases}$$

The ALU performs the DLX ALU instructions such as addition and compare operations, which are indicated by I_ALU (two register operands) and I_ALUi (one register operand and one immediate constant operand). Furthermore, the shift operations are performed by the ALU. The shift operations are indicated by I_shift (two register operands) and I_shifti (one register operand and one immediate constant operand).

In case of an ALU or shift operation with two register operands, the transition function for the destination register is:

$$\delta.GPR[I\_RD(I)](c) = ALU(op1(c), op2(c), ALUfunction(I))$$

In case of an ALU or shift operation with one register operand and one immediate constant operand, the transition function for the destination register is:

$$\delta.GPR[I\_RD(I)](c) =$$
$$ALU(op1(c), I\_immediate(I), ALUfunction(I))$$

Memory Instructions In order to access off-chip memory, the DLX architecture provides load and store instructions. The load instructions copy a value of a memory cell into a register. The store instructions copy the value of a register into a memory cell.

As described in section 2.4.2, the DLX memory is organized in 32-bit words. The address that is to be accessed is computed as follows: the value of the first operand and the immediate constant provided in the instruction word are added. Let EA (effective address) denote this address. It is defined using the addition on bit vectors as defined in section 2.1.5:

$$EA := op1 + I\_immeditate(I)$$

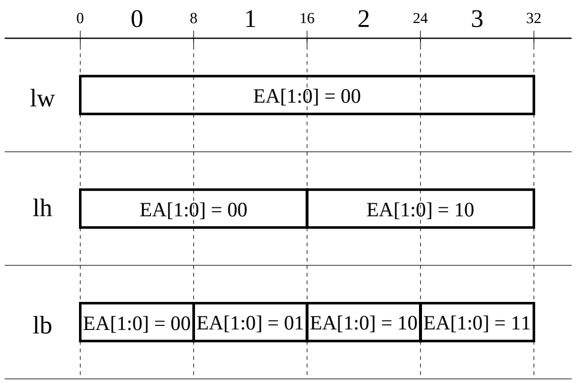


Figure 2.6 The possible alignments for memory instructions

The DLX architecture supports memory accesses with variable widths: byte (8 bits), half word (16 bits), and word (32 bits) accesses are allowed.

The bits $EA[31:2]$ are used to select the word that is to be accessed. The DLX architecture does not support non-aligned accesses, i.e., memory accesses must not cross a memory cell boundary. In case of a word access, this implies that $EA[1:0]$ must be zero. In case of a byte or half word access, $EA[1:0]$ is used to specify the bytes in the memory cell. Figure 2.6 shows the allowed positions of the memory operand within a memory cell.

In case of a load instruction, the 32 bits of the destination register are always written. In case of a byte or half word load instruction, the memory operand is stored in the register beginning with the least significant bits and either a zero or a sign extension is performed. In case of the lh and lb instructions, sign extension is performed, in case of the lhu and lbu instructions, zero extension is performed.

In case of a store instruction, the machines presented in the following chapters assume full word accesses. This restriction will be removed in chapter 6.

2.5 Literature

Besides the basic ciruits presented here, there are more advanced circuits, e.g., adders [LF80, Min95]. There are also HDL generators available for arithmetic circuits such as adders and multipliers [PA96]. Basic circuits with proofs in PVS language are covered by [BJK01].

Fully automated verification of combinational circuits such as adders has been reported using BDDs (binary decision diagrams) [Bry86, FFK88]. The BDDs of some circuits, such as multipliers, grow exponentially in the number of inputs bits. A lot of literature addresses this issue [Bur91, JNFSV97].

Barrett et.al. [BDL98] extend an equivalence-checker by decision procedures for bit vector arithmetic and verify components of a microprocessor such as an instruction fetch unit automatically. The decision procedures are similar to those used in PVS.

The specification of microprocessors as mathematical machine is a common technique [Gau95].

Chapter

3

A Sequential Implementation Machine

3.1 The Prepared Sequential Machine

IN THIS CHAPTER, an implementation machine is built that works as follows: the calculation of a configuration of the specification machine is split in n arbitrary phases, called *stages*. In each phase, the values of a subset of the registers of the configuration of the specification machine are calculated. The implementation machine performs the phases round-robin and needs one transition for each phase, i.e., the implementation machine needs n times as many transitions to do the same calculation as the specification machine.

The calculation is still done in a sequential way, at no time two configurations of M_S are calculated in parallel. However, the structure of the machine will match the structure of the pipelined machine described in the next chapter. The machine is called *prepared sequential* [MP00] machine or M_σ therefore.

Let M_S and M_I be mathematical machines. Let $M_S = (C_S, c_S^0, \delta_S)$ be a specification machine and let $M_I = (C_I, c_I^0, \delta_I)$ be the implementation machine. Let R_S be the registers of the specification machine and R_I be the registers of the implementation machine. The following sections describe how to build a prepared sequential machine that provably simulates the specification machine.

3.2 How Hardware is Specified

3.2.1 A Simple Hardware Description Language

The hardware of the implementation machine consists of the registers of the machine and of the data paths, which calculate the values of the registers. The registers are modeled by the configuration set of the mathematical machine, and the data paths are modeled by the transition function δ. The configuration set and the transition function δ are defined using a simple hardware description language that is similar to a register transfer language (RTL).

For example, in a register transfer language the new value for the *DPC* register is specified as follows:

$$DPC \quad := \quad PC'$$

In this example, the value of PC' is used in order to specify the new value of *DPC*. Formally, suppose the goal is to calculate the value *DPC* has in configuration c_S^i with $i > 0$. In this case, the value used for PC' is the value the register PC' has in configuration c_S^{i-1}. Thus, the following is supposed to hold for all $i > 0$:

$$c_S^i.DPC \quad = \quad c_S^{i-1}.PC'$$

A very similar definition is in [KP95].

In the example above, two things happen:

- The old value of PC' is read.
- The new value of *DPC* is written.

The hardware description language used in this thesis makes use of the following language elements:

1. The configuration set is defined using a list of registers and additional information on the registers such as their domain. This is described in the next section.
2. The transition function δ, i.e., the function computed by the gates between the registers, is defined using a set of functions. This is described in the sections 3.2.4 and 3.2.6.

3.2.2 The Register Set of the Implementation Machine

For all registers R of the implementation machine, let $R \in out(k)$ denote that the register R is updated by stage $k \in \{0,\ldots,n-1\}$.

◄ Definition 3.1 *Specification Register*

The registers of the implementation machine include all registers of the specification machine. These registers are called *specification registers*. The fact that $R \in \mathsf{R}_I$ is a specification register is denoted by $R \in spec$.

By convention, a specification register $R \in \mathsf{R}_I$ can be updated by exactly one stage only. Let the stage $k = stage(R)$ be the stage that updates R. In this case, the register R is also denoted by $R.(k+1)$. This convention and the notation is taken from [MP00].

◄ Definition 3.2 *Implementation Register*

In order to store temporary values used for the calculation, further registers are added to the machine. These registers are called *implementation registers*. The fact that R is an implementation register is denoted by $R \in impl$.

For example, if a processor fetches an instruction word from the instruction memory and stores it in the instruction word register, this instruction word is an intermediate result of the computation of the next state of the reference machine.

In contrast to specification registers, instances $R.k$ of implementation registers R can be present in multiple stages. The function $stage(R)$ is defined to be the first stage an instance of the implementation register is present in:

$$\forall R \in impl : stage(R) = \min\{\ k \mid R.(k+1) \in out(k)\ \}$$

The property of a register whether it is an implementation or specification register is called *class* of the register.

Thus, the configuration set is defined by listing the names of the registers, their types (i.e., domain), and their classes. Furthermore, for each register the stage(s) are given. In case of a specification register, only one stage is allowed. In case of an implementation register, multiple stages are allowed.

In addition to that, the command used in order to define a register is also used in order to specify the value the register has in the initial configuration.

	$T=0$	$T=1$	$T=2$	$T=3$	$T=4$	$T=5$	$T=6$
ue_0^T	1	0	0	0	1	0	0
ue_1^T	0	1	0	0	0	1	0
ue_2^T	0	0	1	0	0	0	1
ue_3^T	0	0	0	1	0	0	0

Table 3.1 The sequential scheduling of a four stage pipeline

3.2.3 Scheduling of the Prepared Sequential Machine

The next step is to define the transition function δ of the machine. The registers of the prepared sequential machine are updated round-robin. In each transition, the registers of only one stage are updated. The update of the registers in $out(k)$ of a stage is controlled by a signal ue_k (update enable). Iff ue_k is one, the registers in $out(k)$ are updated. Table 3.1 gives an example of the values of ue_k for a four stage pipeline. The same concept is used by [MP00].

The stage that is updated before stage k is calculated by the function $prev(k)$:

$$prev : \{0,\dots,n-1\} \to \{0,\dots,n-1\}$$

$$prev(k) = \begin{cases} k-1 & : \quad k \neq 0 \\ n-1 & : \quad k = 0 \end{cases}$$

Stage k is said to be updated in cycle T iff $ue^T_{prev(k)} = 1$ holds.

In analogy to the function $prev$, the function $next(k)$ calculates the stage that is updated after stage k:

$$next : \{0,\dots,n-1\} \to \{0,\dots,n-1\}$$

$$next(k) = \begin{cases} k+1 & : \quad k \neq n-1 \\ 0 & : \quad \text{otherwise} \end{cases}$$

In order to allow the machine M_I to keep track of the stage that is currently processed, an 1-bit register $full.j$ is added to each stage $j \in \{1,\dots,n\}$. If $full.j$ is set, the calculation of the registers $R.j$ is finished.

In the initial configuration, only the full bit of the last stage is set:

$$c^0.full.j \quad = \quad \begin{cases} 1 & : \quad j = n \\ 0 & : \quad \text{otherwise} \end{cases}$$

In addition to that, a *signal* $full_k$ is defined for each stage $k \in \{0,\dots,n-1\}$ as follows:

$$full_k(c) \quad = \quad \begin{cases} c.full.n & : \quad k = 0 \\ c.full.k & : \quad \text{otherwise} \end{cases}$$

If $full_k(c_I^T)$ holds, it is said that stage k is full during cycle T.

In general, the registers in $out(k)$ are updated iff $full_k$ is active. However, some operations on the registers might take more than one cycle, like an access to slow off-chip memory. This requires means to stall the machine. This is realized by a signal $stall_k$ for each stage. If active, the stage is stalled. The signal ue_k is active iff the stage is full and not stalled, therefore:

$$ue_k \quad = \quad full_k \wedge \overline{stall_k}$$

◄ Convention 3.1

By convention, the stall signal of a given stage k must not be active if the stage is not full:

$$\overline{full_k} \quad \Longrightarrow \quad \overline{stall_k}$$

The transition functions of the full bits are defined as follows: a full bit is set iff the stage was updated or the stage was full in the previous cycle and the stage was stalled:

$$\begin{aligned} 1 \le k < n: \quad \delta(c).full.k \quad &= \quad ue_{k-1}(c) \vee (c.full.k \wedge stall_k(c)) \\ k = n: \quad \delta(c).full.n \quad &= \quad ue_{n-1}(c) \vee (c.full.n \wedge stall_0(c)) \end{aligned}$$

Since $stall_k(c)$ implies $full_k(c)$ (convention 3.1), this definition can be simplified to:

$$\begin{aligned} 1 \le k < n: \quad \delta(c).full.k \quad &= \quad ue_{k-1}(c) \vee stall_k(c) \\ k = n: \quad \delta(c).full.n \quad &= \quad ue_{n-1}(c) \vee stall_0(c) \end{aligned}$$

◄ Lemma 3.2

A stage is full iff it is updated or stalled in the previous cycle.

$$full_k^{T+1} \quad = \quad ue_{prev(k)}^T \vee stall_k^T$$

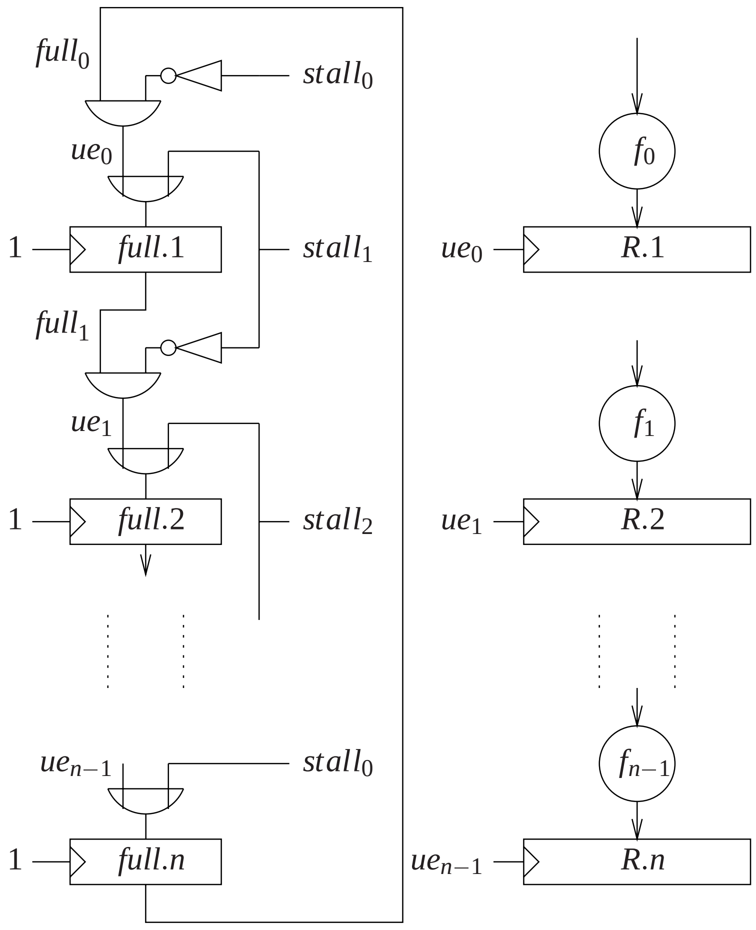


Figure 3.1 The prepared sequential machine

This is shown by unfolding the definition of the full signal $full_k$ and of $prev(k)$.

Figure 3.1 shows the registers of a prepared sequential machine and the clock enable signals that are used for them. As described in chapter 2, the circuits used in order to realize the calculation of the new values for the initial state c^0 are omitted.

3.2.4 The Transition Function

As described above, a register value is supposed to be written only if the update enable signal of its stage is active. The value of the register should remain unchanged otherwise. The overall transition function $\delta.R$ for a register $R \in out(k)$ therefore is generated as follows: if the update enable signal is not active, the old value is taken. If the update enable signal is active, the value provided by a function $\omega_k R(c)$ is taken, which is defined later.

$$\delta.R(c) \quad = \quad \begin{cases} \omega_k R(c) & : \quad ue_k(c) = 1 \\ c.R & : \quad \text{otherwise} \end{cases}$$

The functions $\omega_k R$ are mappings from the configuration of the implementation machine into the domain of the register R. These functions are generated from the hardware description language using the two simple elements: write accesses and read accesses. The accesses are kept in a list. In addition to the data of the read or write access, which is described below, the list contains a flag for each access that specifies whether the access is a read or write access.

A **write access without write address** is a five-tuple $(R.(k+1),\ f_k R,\ dep(R,k),\ f_k Rwe,\ dep\_we(R,k))$ (write accesses with write address are used in order to provide an address for memories or register files and will be described in the next section).

The first element specifies the instance of the register that is written to. We require that exactly one write access is given for each instance $R.(k+1)$ of each register.

The second element, the function $f_k R$, provides the value that is written into the register $R.(k+1)$, i.e., the range of the function is the domain of the register. The function is called *register transition function*. The register transition functions basically model the combinatorial circuits between the pipeline stages. As an example, this includes the ALU, FPU and so on.

A register transition function takes the values of several registers as arguments. These registers are listed in $dep(R,k)$. Let a register R' be in the list of a register R. In this case, it is said that the calculation of R depends on R'. Let $dep(R,k)$ denote the list of registers the calculation of R depends on:

$$dep(R,k) \quad = \quad (R'_1,\ldots,R'_j) \qquad \text{with } R'_l \in \mathsf{R}$$

This allows defining the domain and range of the functions f_kR:

$$f_kR : \mathsf{W}\ (R'_1) \times \ldots \times \mathsf{W}\ (R'_i) \longrightarrow \mathsf{W}\ (R)$$

Furthermore, a function f_kRwe may be provided as element four. The function f_kRwe is called *write enable* signal and can be used in order to realize updates of the given register instance that are only performed under a certain condition. As an example, consider that in case of a microprocessor most registers are only changed by certain instructions. The write enable function allows modeling this. This function may become non-trivial, for example if writing the register is to be suppressed in case of an interrupt.

The range of the function f_kRwe is B. If it returns one, the write access is to be done. If the value is zero, the write access is suppressed. The domain of the function is defined in analogy to the domain of f_kR using a list of input registers named $dep\_we(R,k)$:

$$\begin{aligned} dep\_we(R,k) \quad &= \quad (S'_1,\ldots,S'_o) \qquad \text{with } S'_l \in \mathsf{R} \\ f_kRwe : \mathsf{W}\ (S'_1) &\times \ldots \times \mathsf{W}\ (S'_m) \longrightarrow \mathsf{B} \end{aligned}$$

Let the functions γ_kR and γ_kRwe denote the *values* of the arguments of the functions f_kR and f_kRwe. These functions are defined later using the read accesses.

The effect of f_kRwe depends on whether R is an implementation or specification register. In case of a specification register, the following behaviour is used: If the function returns f_kRwe true, the updating of $R.(k+1)$ is performed. If the function returns false, the updating is suppressed and the value in the register does not change. Thus, if R is a specification register, ω_kR is defined as:

$$\omega_kR(c) \quad = \quad \begin{cases} f_kR(\gamma_kR(c)) & : \quad f_kRwe(\gamma_kRwe(c)) \\ c.R & : \quad \text{otherwise} \end{cases}$$

Note that we have actually two signals that are used in order to determine whether a specification register is to be clocked or not: both the update enable and the write enable signals must be active, i.e., the clock enable signal of a specification register $R \in out(k)$ is:

$$ue_k \wedge f_kRwe(\gamma_kRwe(c))$$

This method is taken from [MP00]. It allows us to specify the stall engine as a module as done in the previous section.

If the write enable signal is false and R is an implementation register, the following behavior is used: the value from the register in the previous stage is written into the register. If there is no instance of the implementation register in the previous stage, a pre-defined default value, e.g., zero, is taken. Thus, if R is a specification register, ω_kR is:

$$\omega_kR(c) \quad = \quad \begin{cases} f_kR(\gamma_kR(c)) & : \quad f_kRwe(\gamma_kRwe(c)) \\ c.R.k & : \quad R \in out(k-1) \\ 0 & : \quad \text{otherwise} \end{cases}$$

This is illustrated in figure 3.2: As an example, consider a processor with an ALU in stage 2. The results are stored in instances of implementation registers C. In case of an ALU instruction, f_2Cwe holds and we store the output of the ALU in the register $C.3$. If not so, the value in $C.2$ is taken. The ALU is modeled by the function f_2C. The multiplexer selecting the appropriate value is modeled by the function ω_2C.

If no function f_kRwe is provided, the constant value true is taken instead, i.e., the updating is performed unconditionally.

3.2.5 Inputs

The functions f_kR and f_kRwe above require certain inputs in order to provide a meaningful value, i.e., it is left to define the functions γ_kR and γ_kRwe. Formalizing the inputs of a register transition function is the most important concept of this thesis, since most of our arguments are used in order to justify how to get those inputs. In particular, we will realize forwarding in pipelined machines and speculation by adjusting these functions accordingly, i.e., the functions model the forwarding logic and speculation circuits.

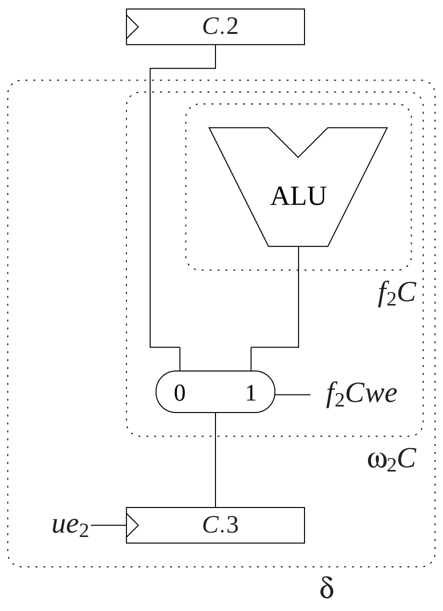


Figure 3.2 Example for f_kR, f_kRwe, and ω_kR

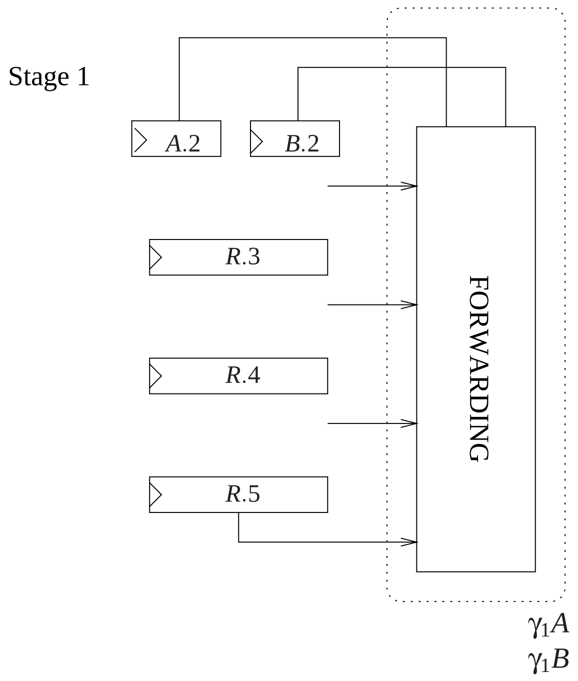


Figure 3.3 Example for the input generation functions

This is illustrated in figure 3.3: it depicts a pipeline that reads two values in stage 2 that require forwarding. The forwarding logic is modeled by the functions $\gamma_1 A$ and $\gamma_1 B$.

The register transition functions depend on a set of input registers. The functions $\gamma_k R$ and $\gamma_k Rwe$ provide the whole set. Let $g_k R'$ be a function that extract the value of a *single* input register R' from the configuration of the implementation macine.

$$g_k R' : C \longrightarrow \mathtt{W}\ (R)$$

We will later on define $g_k R'$. Using $g_k R'$, we define g_k, which takes a configuration and a list of registers. It returns the input values provided by $g_k R'$:

$$g_k(c,(R'_1,R'_2,\ldots,R'_j)) \quad = \quad (g_k R'_1(c), g_k R'_2(c),\ldots,g_k R'_j(c))$$

Let $f_k R$ be the register transition function and $dep(R,k)$ be the list of input registers $(R'_1,\ldots,R'_j)$, as above. Using g_k, we define $\gamma_k R$:

$$\gamma_k R : C \longrightarrow \mathtt{W}\ (R'_1) \times \ldots \times \mathtt{W}\ (R'_j)$$

$$\gamma_k R(c) \quad = \quad g_k(c, dep(R,k))$$

The functions $\gamma_k Rwe$ and $\gamma_k Rre$ are defined in analogy to this definition.

It is left to define the functions $g_k R'$, which calculate the actual input value. As described above, calculating such input values may be complex, for example in machines with forwarding or speculation. In the sequential prepared machine, we neither need forwarding nor speculation. Thus, we define rather simple functions $g_k R'$ for this machine.

In the hardware description language, the definition of $g_k R$ is done using read accesses. A **read access without read address** is a four-tuple (R', k, $f_k R're$, $dep\_re(R',k)$) (read accesses with read address will be described in the next section). For each stage and for each register that is input of the stage, exactly one read access must be defined. The first element is the register (not an instance thereof), the second element is the stage that depends on the register, the third element is a read enable function in analogy to the write enable for write accesses. The function $f_k R're$ also depends on registers:

$$dep\_re(R',k) \quad = \quad (U'_1, \ldots, U'_q) \qquad \text{with } U'_l \in \mathsf{R}$$

$$f_k R're : \mathtt{W}\ (U'_1) \times \ldots \times \mathtt{W}\ (U'_q) \longrightarrow \mathsf{B}$$

As above, $\gamma_k R're$ is used in order to denote the input arguments of $f_k R're$. In order to prevent this definition from becoming recursive, it is required that the read accesses to those registers in $dep\_re(R',k)$ or in $dep\_we(R,k)$ have no read enable signal.[1]

The read enable function has the following purpose: If the read enable signal $f_k R're$ is not active, a default value, e.g., zero, is used as input. This allows us to state whether we actually need an input or not. In case of a microprocessor, not all instructions have an equal number of input registers, some take one GPR operand, some two, and so on. The benefit of knowing when we do not need an input becomes obvious if one considers a machine with forwarding: in case forwarding fails because of data hazards, we do not have to stall if the input is not used anyway.

If no function $f_k R're$ is provided, the constant value true is taken instead, i.e., the read access is performed unconditionally.

[1]It is feasible to extend this definition in order to allow a recursion. However, no microprocessor design implemented for this thesis requires it.

If the read enable signal is active, the value provided by g_kR' is the value of the register. As described above, this simple definition only works in the prepared sequential machine. We will re-define g_kR' for faster machines later on.

The formal definition of g_kR' depends on whether R' is an implementation or specification register. Let w be $stage(R')$.

If R' is an implementation register, an instance of R' is expected to be in the previous stage. If there is no instance of $R' \in out(k-1)$, instances of the register R' are added to $out(w+1), \ldots, out(k-1)$ if not already present. These registers are called “buffer registers”. The transition function for these additional registers is:

$$\omega_p R'(c) = c.R'.p \quad \text{for } p \in \{w, \ldots, k-1\}$$

After this is done, an instance of R' is in $out(k-1)$. The value is read directly from the register $R'.k$.

$$g_kR'(c) \quad = \quad \begin{cases} c.R'.k & : \quad f_kR're(\gamma_kR're(c)) \\ 0 & : \quad \text{otherwise} \end{cases}$$

If R' is a specification register, there is only one instance of R, by definition. In this case, the value in this register is taken. It is required that $w \geq k$ holds (this limitation is removed in chapter 5).

Thus, g_kR' is defined as follows:

$$g_kR'(c) \quad = \quad \begin{cases} c.R'.(w+1) & : \quad f_kR're(\gamma_kR're(c)) \\ 0 & : \quad \text{otherwise} \end{cases}$$

Figure 3.4 shows an example how the functions f_kR and g_kR' are used in order to model hardware. It shows the hardware for an unconditional write access to a register $R.(k+1)$ that depends on two implementation registers R'_1 and R'_2. The read accesses to R'_1 and R'_2 are both unconditional.

3.2.6 Register Files and Memory

In hardware implementations of microprocessors, on-chip memory is used to realize register files. In addition to that, microprocessors provide an

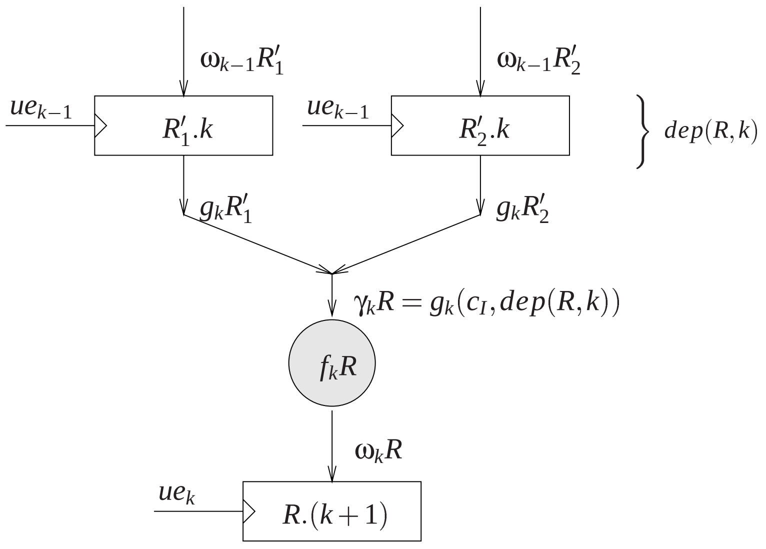


Figure 3.4 The input and output functions for an unconditional write access to a register $R.(k+1)$ that depends on two implementation registers R'_1 and R'_2. The read accesses to R'_1 and R'_2 are both unconditional.

interface to off-chip memory in order to store larger amounts of data. The microprocessor usually reads or writes only a small part of these memories in each cycle.

In theory, accesses to these memories could be modeled as follows: the complete contents are read, some parts are modified and re-written. However, in hardware the access to both memory and register files is limited. For write accesses to register files or memory, the transition functions are therefore expected to provide the value that is to be written, the address of the register or memory cell that is to be modified, and a write enable signal. For read accesses, the transition functions must provide the address and a read signal. The functions $\omega_k R$ defined above model the behavior of the hardware, but are not suited for synthesizing hardware as soon as memory or register files are involved.

The definition of the function $\omega_k R$ is therefore changed if a register file or memory is accessed. This is done by extending the hardware description language using read and write accesses with address. It is presumed that implementation registers are never in a register file or part of a memory.

A write access with write address is defined like a write access without write address but with additional elements $f_k Rwa$ and $dep\_wa(R,k)$, i.e., it is a seven-tuple. The write address function $f_k Rwa$ takes the registers in the list $dep\_wa(R,k)$ as arguments, as done with the arguments of $f_k R$. The function returns the address that is to be used. The range of the function therefore is the set of possible addresses of the access. Let $\mathsf{W}_a(R)$ denote this range.

$$dep\_wa(R,k) \;=\; (V'_1,\ldots,V'_r) \quad \text{with } V'_l \in \mathsf{R}$$
$$f_k Rwa : \mathsf{W}\;(V'_1) \times \ldots \times \mathsf{W}\;(V'_r) \longrightarrow \mathsf{W}_a(R)$$

The range of the function $f_k R$ has to be adjusted accordingly such that it matches the range of a single memory cell or register of the register file; e.g., if a 32x32 bit register file named $GPR.5$ is accessed, $f_4 GPR$ returns a 32-bit vector. Let $\mathsf{W}_r(R)$ denote this range. In the example, the function $f_4 GPRwa$ returns a five-bit vector.

$$dep(R,k) \;=\; (R'_1,\ldots,R'_j) \quad \text{with } R'_l \in \mathsf{R}$$
$$f_k R : \mathsf{W}\;(R'_1) \times \ldots \times \mathsf{W}\;(R'_i) \longrightarrow \mathsf{W}_r(R)$$

The value provided by the function $f_k Rwe$ enables (return value one) or disables the write access (return value zero) to the register or memory cell. This value is taken as write enable signal.

The behavior of the memory or register file is modeled by the function $\omega_k R$ as follows: let the brackets $[\,]$ denote a projection used in order to access a single memory cell or register of a register file and let the function $\gamma_k Rwa$ denote the function that calculates the arguments of the function $\gamma_k R' wa$ as described in section 3.2.4.

$$\forall\, x \in \mathtt{W}_a(R):$$

$$\omega_k R(c)[x] \quad = \quad \begin{cases} f_k R(\gamma_k R(c)) & : \quad f_k Rwe(\gamma_k Rwe(c)) \wedge \\ & : \quad x = f_k Rwa(\gamma_k Rwa(c)) \\ c.R[x] & : \quad \text{otherwise} \end{cases}$$

Furthermore, a read address can be supplied for each read access to a specification register that is a register file or memory. Such a read access is called read access with read address. In analogy to the write address functions $f_k Rwa$, this index is supplied by an additional function $f_k R' ra$, called read address. The function takes arguments as described above for $f_k R' re$. The list of registers is denoted by $dep\_ra(R,k)$. The range of the function is $\mathtt{W}_a(R)$, as described above.

Let the function $\gamma_k R' ra$ denote the function that calculates the arguments of $f_k R' ra$ as described in section 3.2.4. The function $g_k R$ for a conditional specification register read access with read address is:

$$g_k R' : C \longrightarrow \mathtt{W}_r(R)$$

$$g_k R'(c) \quad = \quad \begin{cases} c.R[f_k R' ra(\gamma_k R' ra(c))] & : \quad f_k R' re(\gamma_k R' re(c)) \\ 0 & : \quad \text{otherwise} \end{cases}$$

The generation of the hardware of the implementation machine can now be done automatically by a program that reads the following:

- The program reads the register list including the domain, type, class, and initial value of the register.
- The program reads the list of read and write accesses.

In the following section, the hardware description language above will be used in order to implement a sequential DLX. This is followed by a proof that this implementation simulates the specification as given in chapter 2.

3.2.7 Multiport Read Accesses

In case of a microprocessor, we can have multiple read accesses to the same register file in the same stage. For example, in a DLX implementation we have two read accesses to the general purpose register file. We support separate read enable and read address functions for these read access. Let R be the register that is read.

By convention, we name these functions as follows: the read enable function of the first access is named f_kRa_re, the read enable function of the second access is named f_kRb_re. In analogy to that, the read address function of the first access is named f_kRa_ra, the read address function of the second access is named f_kRb_ra. The list of inputs these functions depend on is denoted by $dep\_re(Ra,k)$, $dep\_re(Rb,k)$, and so on. In analogy to that, the function that provides the inputs to f_kRa_re is named γ_kRa_re, and so on.

Since we have separate read enable and read address functions, we also get different input values. We denote the value generated for the first read access by g_1Ra and the value generated for the second read access by g_1Rb.

As an example, consider two read accesses in stage 1 to the GPR register file. The read enable functions are named f_1GPRa_re and f_1GPRb_re. The input generation functions are named g_1GPRa and g_1GPRb.

3.2.8 Notation

For sake of simplicity, we introduce the following shorthand for formulas that will be used very often in the rest of this thesis: Consider two functions f_kQ and γ_kQ. Let x be a tuple of arguments of γ_kQ. In this thesis, we will often need the value f_kQ of γ_kQ of x:

$$f_kQ(\gamma_kQ(x))$$

We will denote this by $f\gamma_kQ(x)$:

$$f\gamma_kQ(x) \quad := \quad f_kQ(\gamma_kQ(x))$$

For example, the function compositions used in the previous section will be shortened as follows:

$$
\begin{aligned}
f\gamma_k R(c) &:= f_k R(\gamma_k R(c)) && (3.1)\\
f\gamma_k Rre(c) &:= f_k Rre(\gamma_k Rre(c)) && (3.2)\\
f\gamma_k Rwe(c) &:= f_k Rwe(\gamma_k Rre(c)) && (3.3)\\
f\gamma_k Rra(c) &:= f_k Rra(\gamma_k Rra(c)) && (3.4)\\
f\gamma_k Rwa(c) &:= f_k Rwa(\gamma_k Rwa(c)) && (3.5)
\end{aligned}
$$

3.3 Precomputed Control

In the sections above, the signals f_kRwa, f_kRwe, f_kRra, and f_kRre are used in order to specify which register is read or written. The functions that calculate these signals can take an arbitrary number of registers as input just as the functions f_kR.

Consider a write enable signal of stage 4 in a five stage pipeline. Let this write enable signal depend on an instruction word that is calculated by stage 0. In order to read this instruction word in stage 4, one has to add buffer registers for the stages 1 to 4. These registers are quite expensive. In order to save hardware cost, one can calculate the value of the write enable signal already in stage 0 or 1, thus saving the buffer registers.

In order to get the value of the write enable signal, registers for the write enable signal are added instead. However, this requires only a one-bit register for each stage. This is less expensive than the registers for the full instruction word. This can be also done for other signals such as the read/write address.

This method is called *precomputed control* [PH94, MP00]. If the value of a control signal is calculated as described above, it is said that the signal is precomputed.

Naming Convention Let s be the name of a precomputed control signal, e.g., f_4GPRwe. The registers added in order to store the value of the signal will be named $s.k$ with k being the number of the stage the register is an input of, i.e., $s.k \in out(k-1)$. All registers containing precomputed control are summarized by the register $P.k$.

For example, the registers containing the precomputed versions of the write enable signal f_4GPRwe are called $f_4GPRwe.1$, $f_4GPRwe.2$, and so on. Note that these registers are treated like implementation registers. In particular, there are corresponding functions $f_k R$ for each precomputed signal R.

3.4 Implementing the Prepared Sequential DLX

3.4.1 Structure

The prepared sequential machine DLX_σ is the first approach to implement the DLX defined in chapter 2. The execution of an instruction in the DLX_σ is done in five stages. The organization of the stages is similar to the pipeline of a MIPS R2000/R3000 [KH92] and also used in [HP96, MP00]:

- In stage 0 (IF), the instruction fetch is done.
- In stage 1 (ID), the instruction word is decoded and the operands of the instruction are fetched.
- In stage 2 (EX), the ALU calculation is done.
- In stage 3 (M), the memory access for load and store instructions is done.
- In stage 4 (WB), the result of the instruction is written into the register file.

Figure 3.5 shows all registers of the machine DLX_σ and the stage they belong to. As described above, the signals used for precomputed control are summarized as register P. Furthermore, the main components such as ALU and memory are depicted. Table 3.2 lists the stages and summarizes the registers that are written in and read in a given stage, respectively.

Initial Configuration and Transition Function In the initial configuration, the values of specification registers of the DLX_σ machine are identical to the values of the corresponding registers of the specification machine. The implementation registers are initialized with zero.

The transition function of the DLX_σ machine is defined using register transition functions as described in section 3.2.4.

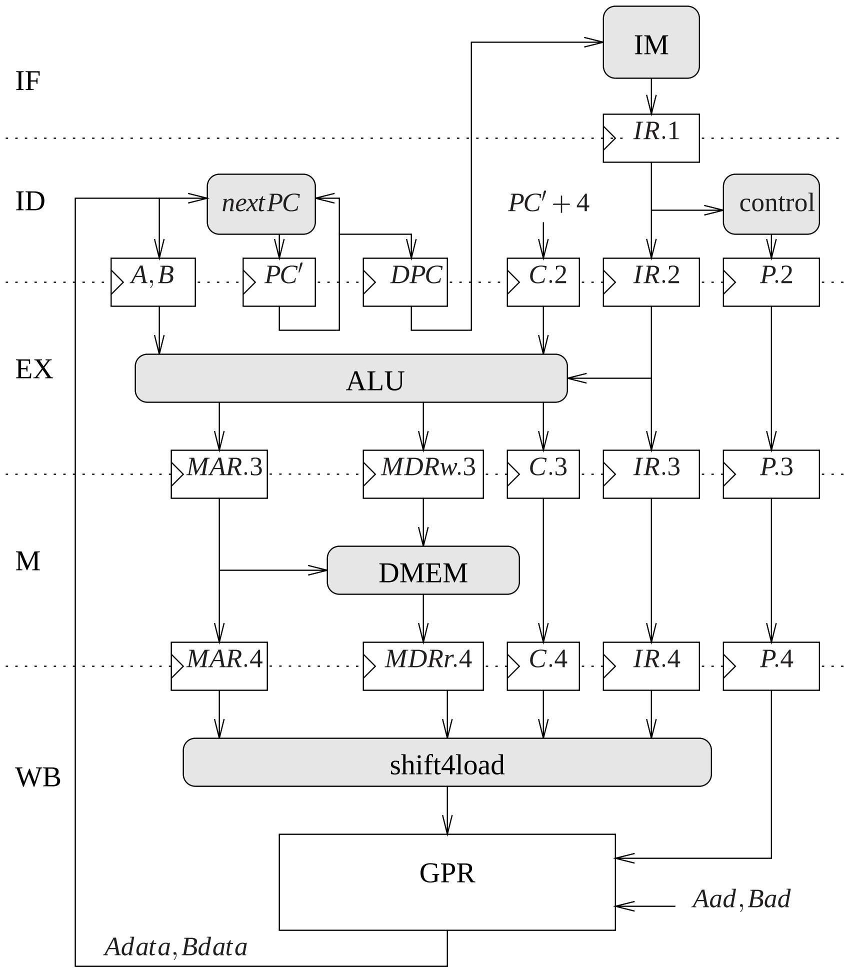


Figure 3.5 The prepared sequential DLX. The registers $P.k$ summarize the registers used for precomputed control.

Stage	Reads	Writes
0	DPC	IR
1	IR, PC', $GPR[Aad] = GPRa$, $GPR[Bad] = GPRb$	DPC, PC', Aad, Bad, A, B, C
2	IR, A, B	$C, MAR, MDRw$
3	$IR, MAR, MDRw, C$, $DM[MAR[31:2]]$	$C, MARh, DM[MAR[31:2]]$, $MDRr$
4	$MDRr, MAR, IR$	GPR

Table 3.2 The registers the stages of the prepared sequential machine read and write, without precomputed control

3.4.2 The Instruction Fetch Stage

The instruction fetch stage IF reads the delayed PC register *DPC* unconditionally and fetches the instruction memory cell that *DPC* points to. This value is stored in the only output register of the stage, the IR implementation register, unconditionally. The register transition function for IR.1 therefore is:

$$f_0 IR(DPC) \quad = \quad IM[DPC] \tag{3.6}$$

3.4.3 The Instruction Decode Stage

The instruction decode stage ID reads the instruction word in the register IR and decodes it.

The operand registers of the instruction are read and stored in two implementation registers A and B. This is realized by means of two conditional read accesses with read address to *GPR*. The naming conventions for such multiport accesses is described in section 3.2.7.

The read enable functions for these read accesses depend on the instruction word and on the source address of the operand: we need to test the instruction word in order to determine whether the instruction requires the operand or not. This is determined by testing the instruction word read from *IR* using the functions defined in chapter 2.

In addition to that, we test the address. If the address is zero, the read access is not necessary, since the *GPR* register with address zero has the constant value zero, as required by the DLX specification. If the read enable function is not active, the value zero is passed to the register transition function by convention. This is exactly the value required by the specification. Thus, we omit an extra multiplexer in order to get zero in case of a read access to $GPR[0]$.

The first operand is required by loads, stores, ALU instructions, branch instructions, and the jump register instructions. Thus, the read access is performed if the following condition holds:

$$\begin{aligned} f_1GPRa\_re(IR) &= (I\_load(IR) \vee I\_store(IR) \vee \\ &\quad I\_ALUi(IR) \vee I\_branch(IR) \vee \\ &\quad I\_jr(IR) \vee I\_shift(IR) \vee \\ &\quad I\_ALU(IR) \vee I\_shifti(IR)) \wedge \\ &\quad (I\_RS1(IR) \neq 0) \end{aligned} \tag{3.7}$$

The second operand is required by ALU instructions that do not use the immediate constant as second argument and by store instructions. In case of store instructions, the address of the second operand is stored in the RS2 location of the instruction word and not in the RD location (appendix B).

$$\begin{aligned} f_1GPRb\_re(IR) &= (I\_shift(IR) \vee I\_ALU(IR) \vee I\_store(IR)) \wedge \\ &\quad (I\_store(IR) ? I\_RD(IR) : I\_RS2(IR)) \neq 0) \end{aligned} \tag{3.8}$$

The indices of the read accesses are calculated from the instruction word in *IR*: In case of the first operand, I_RS1 provides the address. In case of the second operand, it is necessary in order to distinguish store instructions from ALU instructions:

$$\begin{aligned} f_1GPRa\_ra(IR) &= I\_RS1(IR) \\ f_1GPRb\_ra(IR) &= \begin{cases} I\_RD(IR) & : I\_store(IR) \\ I\_RS2(IR) & : \text{otherwise} \end{cases} \end{aligned} \tag{3.9}$$

The result of the instruction is buffered in the implementation register C. In the decode stage, only the result of jump and link instructions is known already, which is $PC' + 4$. This value is stored in C therefore if the instruction is a jump and link instruction. This is realized using a conditional write access to $C.2$.

$$f_1C(PC') = PC' + 4 \tag{3.10}$$

$$f_1Cwe(IR) = (I\_jr(IR) \vee I\_j(IR)) \wedge I\_link(IR) \tag{3.11}$$

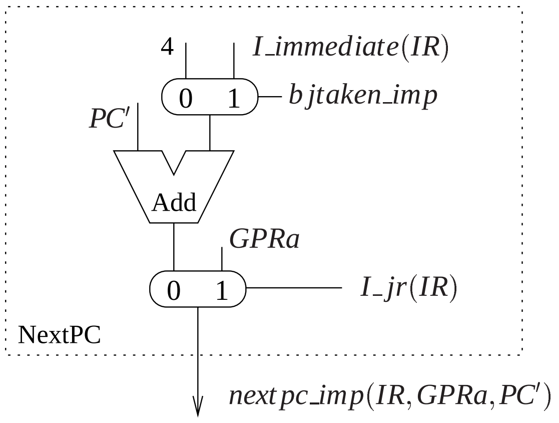


Figure 3.6 The implementation of *nextpc*

Furthermore, the decode stage calculates the new values for the PC registers *DPC* and PC' according to the Delayed PC technique. Let *GPRa* denote the value of the first operand, as calculated by g_1GPRa.

$$f_1DPC(PC') = PC' \tag{3.12}$$
$$f_1PC'(IR, GPRa, PC') = nextpc\_imp(IR, GPRa, PC') \tag{3.13}$$

The function *nextpc_imp* is implements the *nextpc* calculation as defined in chapter 2. It is defined in the obvious way using a zero tester in order to calculate the *bjtaken* signal, as defined in section 2.4.4:

$$\begin{aligned} & bjtaken\_imp(IR, GPRa) \\ := \; & I\_j(IR) \vee I\_jr(IR) \vee (I\_branch(IR) \wedge \\ & (I\_branch\_eq(Iw) \oplus zerotester\_imp(GPRa))) \end{aligned} \tag{3.14}$$

By using the correctness of the zero tester (lemma 2.2), one easily shows the correctness of the circuit that calculates *bjtaken*:

◀ Lemma 3.3

The calculation of the *bjtaken* signal is correct:

$$bjtaken\_imp = bjtaken$$

Using the *bjtaken_imp* signal and a carry lookahead adder as described in section 2.2.6, we calculate the new PC (figure 3.6). In case of a jump register instruction, we take the value of the GPR operand. In case of a taken branch, we add the old PC and the PC offset from the instruction word. In any other case, we take the old PC incremented by four.

Lemma 3.4 ▶ The calculation of the new PC is correct:

$$nextpc\_impl = nextpc$$

This is shown easily using the correctness of the adder circuit.

Precomputed Control In addition to that, the decode stage also does the precomputation of several control signals. The following signals are precomputed in the decode stage:

- The write enable signal and write address used in the write back stage,
- the write enable signals of all C registers.

The formulae for these signals are given in the sections of the stages the registers belong to for sake of simplicity. Note that the circuits calculating the signal values actually belong to the decode stage. In the later stages, the signal is just taken from the register holding the precomputed signal and no calculation is performed.

3.4.4 The Execute Stage

In the execute stage, the result of all ALU instructions is computed. This includes the integer instructions such as addition and subtraction, the shifting instructions, and the compare instructions. Furthermore, the address computation for memory instructions is performed.

The stage reads the values of the operands from implementation registers $A.2$ and $B.2$. However, both the memory instructions and the ALU instructions with immediate constant (e.g., addi) take the immediate constant from the instruction word as second operand. Let $aluop2$ denote the value of the second operand:

$$aluop2(IR,B) = \begin{cases} B & : I\_ALU(IR) \vee \\ & \quad I\_shift(IR) \\ I\_immediate(IR) & : \text{otherwise} \end{cases} \tag{3.15}$$

The function $aluop2$ is used as a shorthand for this text only; in the PVS tree, the expanded form is always used.

In case of ALU instructions, the operation that is to be performed is provided by the function $ALUfunction$. This function is defined in section 2.4.4. In case of memory instructions, as indicated by I_load and I_store, an addition is performed in order to compute the effective address.

$$aluf(IR) = \begin{cases} (1,0,0,0,0) & : I\_store(IR) \vee I\_load(IR) \\ ALUfunction(IR) & : \text{otherwise} \end{cases}$$

The implementation register $C.3$ holds the result provided by the ALU. It is only written on ALU instructions.

$$\begin{aligned} f_2C(IR,A,B) &= ALU(A, aluop2(IR,B), aluf(IR)).result \\ f_2Cwe(IR) &= I\_ALU(IR) \vee I\_ALUi(IR) \vee \\ & \quad I\_shifti(IR) \vee I\_shift(IR) \end{aligned}$$

In case of a memory instruction, the result (i.e., the address of the memory operand) is stored in the register $MAR.3$.

$$f_2MAR(IR,A,B) = ALU(A, aluop2(IR,B), aluf(IR)).result$$

In the register $MDRw.3$, the second operand is stored, which is the value to be stored in memory in case of a store instruction.

$$f_2MDRw(B) = B$$

3.4.5 The Memory Stage

In the memory stage, the memory access for load and store instructions is performed. In order to realize load instructions, a conditional read access with read address to *DM* is performed. The read access is performed iff the instruction is a load instruction. The read address is the high-order 30 bits of the effective memory address stored in *MAR*.

$$\begin{aligned} f_3DMre(IR) &= I\_load(IR) \\ f_3DMra(MAR) &= MAR[31:2] \end{aligned}$$

This result is stored in the register *MDRr*. The result of the read access, as provided by g_3DM, is named *DMemout*.

$$f_3MDRr(DMemout) = DMemout$$

The register *C* is passed to the next stage without modification. The write enable function of the write access to $C.4$ is constant false therefore (compare the definition of conditional write accesses on page 43).

$$f_3Cwe(IR) = 0$$

In order to realize store instructions, a conditional write access to *DM* is performed. The value read from the register *MDRw* is written iff the instruction is a store instruction. The address of the write access is the upper 30 bits of the effective memory address, as above.

$$\begin{aligned} f_3DMwe(IR) &= I\_store(IR) \\ f_3DMwa(MAR) &= MAR[31:2] \\ f_3DM(MDRw) &= MDRw \end{aligned}$$

3.4.6 The Write Back Stage

In the write back stage, the result of the instruction is stored in the register file. In case of a load instruction, the data word fetched from the data memory present in MDRr is shifted and masked prior to write back. This is done using the function $shift4load$, which is defined in section 2.4.4. In case of any other instruction, the result is read from the implementation register C.

$$f_4GPR(C, IR, MAR, MDRr) =$$
$$\begin{cases} shift4load(MAR, MDRr, IR) & : I\_load(IR) \\ C & : \text{otherwise} \end{cases}$$

The write access to the register file is conditional; the condition is that the instruction has a GPR destination operand. This is true for ALU/shift instructions, loads, and jump and link instructions. Thus, the write enable signal is:

$$\begin{aligned} f_4GPRwe(IR) &= I\_ALU(IR) \lor I\_ALUi(IR) \lor I\_load(IR) \lor \\ & \quad I\_shifti(IR) \lor I\_shift(IR) \lor \\ & \quad ((I\_j(IR) \lor I\_jr(IR)) \land I\_link(IR)) \end{aligned}$$

Furthermore, the write access has a write address. As defined in chapter 2, the function $I\_RD(IR)$ determines the address destination register.

$$f_4GPRwa(IR) = I\_RD(IR)$$

Both the write enable and the write address signals are precomputed in the decode stage as described in section 3.3, i.e., the calculation is done in the decode stage and the result is buffered using additional registers. In the write back stage, one just takes the values from the registers.

Note that the cost savings of precomputing these signals are low in the prepared sequential machine. However, in the pipelined machine presented in the next chapter, we will need the signals in multiple stages for forwarding. In this case, precomputing the signals saves a significant amount of hardware since the computation has to be done only once. In order to prevent that we need to make changes to the machine due to pipelining, we already introduce the precomputed control in this chapter.

3.5 Data Consistency Proof

3.5.1 Properties of the Full Bits

Using the equations for the full bits and update enable signals, it is easy to conclude the following properties:

◄ Lemma 3.5

If a stage is full, either the same or the previous stage was full in the previous cycle.

$$full_k^{T+1} \implies full_k^T \lor full_{prev(k)}^T$$

PROOF By lemma 3.2 (page 39), one can conclude from $full_k^{T+1}$ that $ue_{prev(k)}^T$ or $stall_k^T$ holds. If $ue_{prev(k)}^T$ holds, $full_{prev(k)}^T$ holds by definition of the *ue* signals. If $stall_k^T$ holds, $full_k^T$ holds by convention 3.1.

Lemma 3.6 ▶ If a stage is full, either the same or the next stage is full in the next cycle.

$$full_k^T \quad \Longrightarrow \quad full_k^{T+1} \vee full_{next(k)}^{T+1}$$

PROOF Assume neither $full_k^{T+1}$ nor $full_{next(k)}^{T+1}$ holds. Using lemma 3.2 for stages k and $next(k)$, one concludes that neither $ue_{prev(k)}$, nor $ue_{prev(next(k))}$, nor $stall_k^T$, nor $stall_{next(k)}^T$ holds.

By definition of ue_k^T, it is concluded that ue_k^T holds. An easy proof shows that $prev(next(k)) = k$. This allows concluding that $full_{next(k)}^{T+1}$ holds, which is a contradiction to the assumption.

Lemma 3.7 ▶ If a stage k becomes full in cycle $T+1$, the previous stage $prev(k)$ was full in the previous cycle and the output registers of the stage $prev(k)$ were updated.

$$\overline{full_k^T} \wedge full_k^{T+1} \quad \Longrightarrow \quad full_{prev(k)}^T \wedge ue_{prev(k)}^T$$

PROOF By lemma 3.2, $ue_{prev(k)}^T$ or $stall_k^T$ holds. By convention 3.1, $ue_{prev(k)}^T$ is concluded. The first claim, $full_{prev(k)}^T$, is concluded by definition of *ue*.

Lemma 3.8 ▶ In every cycle, exactly one stage is full.

$$\exists! k : \quad full_k^T$$

PROOF The proof proceeds by induction on T. For $T = 0$, the claim obviously holds. For $T+1$, one has to prove that at least one full bit is set and that this full bit is unique.

It is easy to show that at least one full but is set by a case split on the stall signal of the stage with the full bit set. Let stage k be this stage. If the stage is stalled, in cycle $T+1$ the full bit of the same stage is set. If the stage is not stalled, the full bit of stage $next(k)$ is set in cycle $T+1$.

This full bit is unique, i.e., $full_x^{T+1} \wedge full_y^{T+1}$ implies that x is equal y. Assume that $x \neq y$ holds. According to lemma 3.5, there are four cases:

1. $full_x^T \wedge full_y^T$
2. $full_{prev(x)}^T \wedge full_y^T$
3. $full_x^T \wedge full_{prev(y)}^T$
4. $full_{prev(x)}^T \wedge full_{prev(y)}^T$

The cases one and four are disproved by the induction premise. Let case 2 hold (otherwise, swap x and y). According to the induction premise, $y = prev(x)$ must hold. Using lemma 3.7, ue_y^T is concluded, which is equal to $full_y^T \wedge \overline{stall_y^T}$ by definition.

Since $prev(y) \neq y$, and because of the induction premise, $\overline{full_{prev(y)}^T}$ holds. This allows concluding that $\overline{ue_{prev(y)}^T}$ holds. Since $full_y^{T+1}$ is active, this is a contradiction to $\overline{stall_y^T}$ according to lemma 3.2.

3.5.2 Scheduling Functions

Unless stalled, the implementation machine calculates parts of the configurations $c_S^0, c_S^1, \ldots$ of the specification machine. A *scheduling function* [MP00] specifies which configuration is being calculated by the machine in a given stage and cycle. If stage k is full during cycle T, let

$$sI(k,T) \quad = \quad i$$

denote that the implementation machine is performing a part of the computation of configuration c_S^{i+1} in stage k during cycle T.[2]

In case of a microprocessor, let

$$I_0,\ I_1,\ I_2,\ \ldots$$

denote an instruction sequence. In this case, the configuration c_S^{i+1} of the specification machine provides the values of the registers *after* executing instruction I_i, i.e., instruction I_i transforms configuration c_S^i into c_S^{i+1}:

$$c_S^0 \xrightarrow{I_0} c_S^1 \xrightarrow{I_1} c_S^2 \ldots c_S^i \xrightarrow{I_i} c_S^{i+1} \ldots$$

[2]The function is named sI and not I, as in [MP00], because I is used as the identity in PVS.

This is different from the notation used in [MP00]. In [MP00], c^{i+1} denotes a value before the execution of I_{i+1}.

If $sI(k,T) = i$ holds and if stage k is full during cycle T, it is said that instruction i is in stage k during cycle T [MP00].

For this thesis, the domain of the function above is extended to cycles T in that the stage k is not full in order to simplify some proofs. If the stage k was never full before cycle T, $sI(k,T)$ is supposed to be zero. If the stage k was full before cycle T, the supposed value of the function $sI(k,T)$ is defined using the value the function had in the last cycle $T' < T$ such that $full_k^{T'}$ holds. In this case, $sI(k,T)$ is supposed to be $sI(k,T') + 1$ in anticipation of the next instruction in the stage. In contrast to the definition of the scheduling function in [MP00], such a scheduling function sI is total.

A scheduling function of the prepared sequential machine is constructed as follows: The following properties of the scheduling function should hold obviously:

1. During cycle 0, all stages are in the initial configuration:

$$\forall k : sI(k,0) = 0$$

2. If the output registers of a stage k are not updated during cycle $T-1$ (i.e., $ue_k^{T-1} = 0$), the stage was either not full or stalled. The stage was inactive; the value of the scheduling function should not change either.

$$ue_k^{T-1} = 0 \quad \Rightarrow \quad sI(k,T) = sI(k,T-1)$$

3. If the output registers of a stage k are updated during cycle $T-1$ (i.e., $ue_k^{T-1} = 1$), the registers are updated with values of the same configuration that is in the previous stage, i.e., stage $k-1$. The scheduling function must reflect this.

$$k \geq 1 \wedge ue_k^{T-1} = 1 \quad \Rightarrow \quad sI(k,T) = sI(k-1,T-1)$$

   In case of the first stage ($k = 0$), the computation of the next configuration of the specification machine is started:

$$ue_0^{T-1} = 1 \quad \Rightarrow \quad sI(0,T) = sI(0,T-1) + 1$$

	$T=0$	$T=1$	$T=2$	$T=3$	$T=4$	$T=5$	$T=6$
$sI(0,T)$	0	1	1	1	1	2	2
$sI(1,T)$	0	0	1	1	1	1	2
$sI(2,T)$	0	0	0	1	1	1	1
$si(3,T)$	0	0	0	0	1	1	1

Table 3.3 The values of sI in a four stage sequential machine in the absence of stalls

This allows for a recursive definition of the scheduling function of the prepared sequential machine:

$$sI(k,T) \quad = \begin{cases} 0 & : \quad T=0 \\ sI(k,T-1) & : \quad T \neq 0 \wedge \overline{ue_k^{T-1}} \\ sI(0,T-1)+1 & : \quad T \neq 0 \wedge ue_k^{T-1} \wedge k=0 \\ sI(k-1,T-1) & : \quad T \neq 0 \wedge ue_k^{T-1} \wedge k \neq 0 \end{cases}$$

Table 3.3 illustrates the values of $sI(k,T)$ for the first seven cycles assuming four stages and that the stall signals are never active.

3.5.3 Properties of the Scheduling Function

◄ Invariant 3.1

If the update enable signal of a stage is active in cycle $T-1$, the value of the scheduling function for that stage increases by one. If the update enable signal of a stage is not active, the value does not change. For $T>0$:

$$sI(k,T) \quad = \begin{cases} sI(k,T-1) & \text{if } ue_k^{T-1}=0 \\ sI(k,T-1)+1 & \text{if } ue_k^{T-1}=1 \end{cases}$$

◄ Invariant 3.2

Given a cycle T, the values of the scheduling functions of two adjacent stages are either equal or the value of the scheduling function of the earlier stage is greater by one.

◄ Invariant 3.3

The value of the scheduling function of the earlier stage is greater by one iff the full bit of the later stage is set. For $k>0$:

$$full_k^T = 1 \Leftrightarrow sI(k-1,T) = sI(k,T)+1$$

Negating both sides of the last equation and applying invariant 3.2 results in:

$$full_k^T = 0 \Leftrightarrow sI(k-1,T) = sI(k,T)$$

PROOF The proof of the invariants proceeds by induction. Let $P_i(T)$ denote that invariant i holds for cycle T. The claim is concluded as follows:

Invariant 3.1 for cycle T is shown using invariant 3.3 for cycle $T-1$. Invariant 3.2 for cycle T is shown using invariant 3.1 in cycle T and invariant 3.3 in cycle $T-1$. Invariant 3.3 is shown using invariant 3.1 in cycle T and invariant 3.2 in cycle $T-1$.

$$\begin{aligned} P_3(T-1) &\implies P_1(T) \\ P_1(T) \wedge P_2(T-1) \wedge P_3(T-1) &\implies P_2(T) \\ P_1(T) \wedge P_2(T-1) \wedge P_3(T-1) &\implies P_3(T) \end{aligned}$$

Proof of Invariant 3.1 The claim for the case $ue_k^{T-1} = 0$ holds by definition of sI. Let $ue_k^{T-1} = 1$ hold. For the case $k = 0$, the claim follows from the definition of sI. For $k > 0$, the claim is:

$$sI(k,T) = sI(k,T-1) + 1$$

According to the definition of $sI(k,T)$, this is equivalent to:

$$sI(k-1,T-1) = sI(k,T-1) + 1$$

According to invariant 3.3 for cycle $T-1$, this is equivalent to $full_k^{T-1} = 1$. This is true because of the definition of ue_k^{T-1}.

Proof of Invariant 3.2 For cycle $T = 0$, the claim holds by definition of $sI(k,0)$.

For $T > 0$, let us consider the stages $k-1$ and k with $k > 0$. There are four cases regarding the update enable signals ue_k^{T-1} and ue_{k-1}^{T-1} of these stages:

1. Let both update enable signals be active. According to the definition of the update enable signals, this is a contradiction to the fact that at most one full bit is active in a given cycle (lemma 3.8).

2. Let both update enable signals be not active. According to invariant 3.1, the values of the scheduling function do not change and the claim follows from invariant 3.2 for cycle $T-1$ therefore.

3. Let the update enable signal of stage k be active and the update enable signal of stage $k-1$ be not active. Let the first case given by invariant 3.2 for cycle $T-1$ hold:

   $$sI(k-1, T-1) \quad = \quad sI(k, T-1)$$

   Using lemma 3.1 for stage k on the right-hand side, one concludes:

   $$sI(k-1, T-1) \quad = \quad sI(k, T) - 1$$

   According to the definition of $sI(k, T)$, this is equal to:

   $$sI(k-1, T-1) \quad = \quad sI(k-1, T-1) - 1$$

   This is a contradiction. The case above therefore never happens.

   Let the second case given by invariant 3.2 for cycle $T-1$ hold, i.e.,

   $$sI(k-1, T-1) \quad = \quad sI(k, T-1) + 1$$

   holds. Using invariant 3.1 for both stages k and $k-1$, $sI(k-1, T) = sI(k, T)$ is concluded.

4. Let the update enable signal of stage k be not active and the update enable signal of stage $k-1$ be active. Let the first case given by invariant 3.2 for cycle $T-1$ hold, i.e., $sI(k-1, T-1)$ is equal to $sI(k, T-1)$. Using invariant 3.1, $sI(k-1, T) = sI(k, T) + 1$ is concluded.

   Let the second case given by invariant 3.2 for cycle $T-1$ hold, i.e., $sI(k-1, T-1) = sI(k, T-1) + 1$ holds. According to invariant 3.3, $full_k^{T-1}$ holds. According to the definition of the update enable signals, $full_{k-1}^{T-1}$ also holds. This is a contradiction to lemma 3.8.

Proof of Invariant 3.3 For $T = 0$, the claim is shown using the definition of sI. For $T > 0$, according to lemma 3.2, the claim is equivalent to:

$$ue_{prev(k)}^{T-1} \vee stall_k^{T-1} \quad \Longleftrightarrow \quad sI(k-1, T) = sI(k, T) + 1$$

Since $prev(k) = k-1$ for all $k > 0$, this is equivalent to:

$$ue_{k-1}^{T-1} \vee stall_k^{T-1} \quad \Longleftrightarrow \quad sI(k-1, T) = sI(k, T) + 1$$

The proof proceeds by a full case split on the values of the update enable bits ue_{k-1}^{T-1} and ue_k^{T-1}, as done in the proof of invariant 3.2. There are four cases:

1. If both update enable signals are on, this is a contradiction to the fact that at most one full bit is on (lemma 3.8).

2. If ue_{k-1}^{T-1} is on and ue_k^{T-1} is off, the left side of the equivalence evaluates to true and the claim is equal to:

   $$sI(k-1,T) \quad = \quad sI(k,T)+1$$

   Invariant 3.1 for cycle T and stages $k-1$ and k is used to show that the claim is equal to:

   $$sI(k-1,T-1)+1 \quad = \quad sI(k,T-1)+1$$

   Obviously, this claim is equal to:

   $$sI(k-1,T-1) \quad = \quad sI(k,T-1)$$

   Assume this claim does not hold. In this case, invariant 3.2 states that

   $$sI(k-1,T-1) \quad = \quad sI(k,T-1)+1$$

   holds. According to invariant 3.3 for cycle $T-1$, this implies that $full_k^{T-1}$ holds. Since $full_{k-1}^{T-1}$ also holds because of the definition of ue_{k-1}^{T-1}, this is a contradiction to the fact that at most one full bit is on (lemma 3.8).

3. If ue_{k-1}^{T-1} is off and ue_k^{T-1} is on, it is left to show that

   $$stall_k^{T-1} \quad \Longleftrightarrow \quad sI(k-1,T) = sI(k,T)+1$$

   holds. Using invariant 3.1 for stages $k-1$ and k and cycle T one shows that this is equal to:

   $$stall_k^{T-1} \quad \Longleftrightarrow \quad sI(k-1,T-1) = sI(k,T-1)+2$$

   According to the definition of the update enable signal ue_k^{T-1}, the stall signal $stall_k^{T-1}$ cannot be active. Invariant 3.2 shows that $sI(k-1,T-1) = sI(k,T-1)+2$ never holds. Thus, both sides of the equivalence are false.

4. If both update enable signals are off, invariant 3.1 shows that the claim is equal to:

$$stall_k^{T-1} \quad \Longleftrightarrow \quad sI(k-1,T-1) = sI(k,T-1)+1$$

Using invariant 3.3 for cycle $T-1$, one shows:

$$sI(k-1,T-1) = sI(k,T-1)+1 \quad \Longleftrightarrow \quad full_k^{T-1}$$

Thus, the claim is equivalent to:

$$stall_k^{T-1} \quad \Longleftrightarrow \quad full_k^{T-1}$$

By definition of ue_k^{T-1}, $full_k^{T-1}$ implies $stall_k^{T-1}$ if ue_k^{T-1} is off. The opposite direction is given by convention 3.1.

QED

Invariant 3.3 can be extended to multiple stages inductively, which results in the following claim:

◄ Lemma 3.9

Let k and l be stage numbers and $l > k$. If the full bit of all stages between k and l (including stage l, not including stage k) is not set, the scheduling functions for stage k and l are equal:

$$(\forall m \mid m > k \wedge m \leq l : \overline{full_m^T}) \quad \Longrightarrow \quad sI(k,T) = sI(l,T)$$

PROOF

The claim is shown by induction on l using invariant 3.3.

◄ Lemma 3.10

Let stage k be full in cycle T. In this case, stages after stage k contain the values of the same configuration and stages prior to stage k contain the values of the next configuration.

$$T > 0 \wedge full_k^T \quad \Longrightarrow \quad sI(l,T) = \begin{cases} sI(k,T)+1 & l < k \\ sI(k,T) & \text{otherwise} \end{cases}$$

This lemma is the central lemma for showing the correctness of the operands read. The lemma is almost identical to the dateline lemma presented in [MP00].

Lemma 3.10 is illustrated by figure 3.7: Let $full_2^T$ hold and $sI(2,T)$ be i. In this case, the output registers of the stages 0 and 1 already contain the

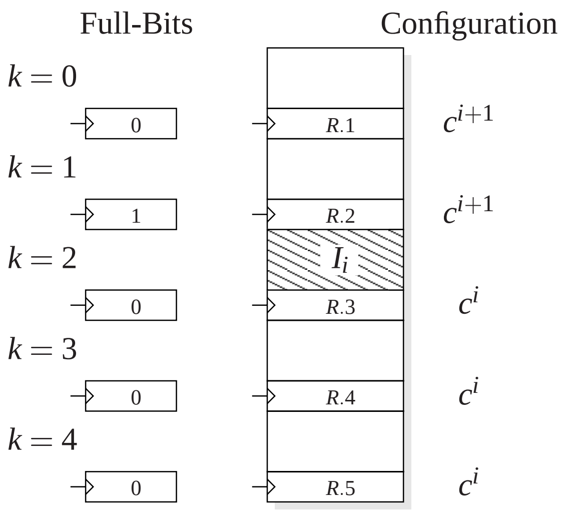


Figure 3.7 Calculation of the configurations in the sequential prepared machine. In the current cycle, instruction I_i is in stage 2.

values of configuration c^{i+1}. The stages 2, 3, and so on still contain the values of configuration c^i.

PROOF For $l = k$, the claim is obvious. For $l < k$, the claim is

$$sI(l,T) \quad = \quad sI(k,T)+1$$

According to invariant 3.3, $sI(k-1,T) = sI(k,T)+1$ holds, which shows the claim for $l = k-1$. For $l < k-1$, lemma 3.8 states that the full bits are not set. Thus, lemma 3.9 can be used in order to show the claim.

For $l > k$, lemma 3.8 states that the full bits of these stages are not set either. Lemma 3.9 shows the claim. QED

◄ Lemma 3.11

Stage k is full at the earliest in cycle k.

$$full_k^T \quad \Longrightarrow \quad T \geq k$$

PROOF

The proof proceeds by induction over T. For $T = 0$, the claim is concluded by the fact that during cycle 0, only full signal $full_0$ is active.

Assuming the lemma for cycle T, the claim for $T+1$ is shown as follows: For $k = 0$, the claim is obvious. Thus, the claim is shown for $k > 0$.

If $full_k^T$ or $full_{k-1}^T$ holds, one simply uses the induction premise. If $full_k^T$ and $full_{k-1}^T$ do not hold, one shows that $full_k^{T+1}$ cannot hold using lemma 3.2:

$$full_k^{T+1} \quad = \quad ue_{k-1}^T \vee stall_k^T$$

Applying the definition of ue_{k-1}^T, this results in:

$$\begin{aligned} full_k^{T+1} \quad &= \quad (full_{k-1}^T \wedge \overline{stall_{k-1}^T}) \vee stall_k^T \\ &= \quad stall_k^T \end{aligned}$$

According to convention 4.2, $stall_k^T$ cannot be active. QED

3.5.4 Data Consistency Proof Strategy

The correctness criterion for the machines presented in this thesis is based on the scheduling function: the values of the specification registers of the implementation machine must match the values of the corresponding registers in the specification machine. Given a stage k and a cycle T, the scheduling function provides the configuration of the specification machine to compare with.

Thus, the correctness of the complete machine is asserted in the following theorem:

Theorem 3.12 ► The value of a given specification register $R \in out(k)$ during cycle T in the implementation machine must match the value of the same register in the specification machine in the configuration c_S^i with $i = sI(k,T)$.

$$R_I^T \quad = \quad R_S^i$$

This data consistency criterion is taken literally from [MP00] **but with index shift**. This index shift arises from a notational difference: in [MP00], R_i denotes the value of R *after* the execution of I_i. In this thesis, R_S^i denotes $c_S^i.R$, which is the value of R *before* the execution of I_i. This difference can be adjusted by taking R_S^{i+1}. Thus, the correctness criterion of [MP00] in the notation of this thesis is:

$$R_I^{T+1} \quad = \quad R_S^{i+1}$$

Furthermore, in [MP00], the criterion is shown for cycles T with ue_k^T only. Using invariant 3.1, one can conclude that for this case $sI(k,T+1) = i+1$ holds. Inserting this into the equation above results in:

$$R_I^{T+1} \quad = \quad R_S^{sI(k,T+1)}$$

This is exactly the correctness criterion as given above despite that the criterion in [MP00] does not cover the values of the registers during cycle 0 (initial configuration).

The proof of the correctness criterion proceeds by induction on T. For the PVS tree, an automated tool developped by the autor generates this proof. In the following, the generic algorithm used in order to generate the proof is described.

Step 1 For all implementation registers R of the implementation machine, a function $\Omega_k R(c)$ is defined. This function maps a configuration

c of the *specification* machine on the domain $\mathtt{W}(R)$ of the register R and provides the "correct" value of R. It is not necessary to define this function for specification registers, since the correct value of a specification register is defined by the specification machine.

For intuition, take the prepared sequential machine and remove all implementation registers. The inputs of the registers are connected to the outputs (figure 3.8). The remaining specification registers share a common clock. This machine processes one configuration of the specification machine with each cycle unless stalled. The configuration set of this machine exactly matches the configuration set of the specification machine. Let c be such a configuration. In this machine, one can get the value of $\Omega_{k-1}R'(c)$ right at the point where the register $R'.k$ formerly was.

Formally, the functions $\Omega_k R$ are defined recursively: in analogy to g and γ (section 3.2.4), functions G and Γ are defined, which provide the correct input values for a register transition function f. In analogy to the function ω, the function Ω is defined. The definition of Ω is identical to the definition of ω except for that Γ is used instead of γ.

$$G_k R : \, C_S \longrightarrow \mathtt{W}(R)$$

Let $G_k(c,(R'_1,R'_2,\ldots,R'_j))$ denote a j-tuple of values calculated as follows:

$$G_k(c,(R'_1,R'_2,\ldots,R'_i)) = (G_k R'_1(c), G_k R'_2(c),\ldots,G_k R'_i(c))$$

Let Γ be a function that maps a configuration of the specification machine to the correct input values of a register transition function. Let $dep(R,k)$ be $(R'_1,\ldots,R'_i)$.

$$\Gamma : C_S \longrightarrow \mathtt{W}(R'_1)\times\ldots\times\mathtt{W}(R'_i)$$

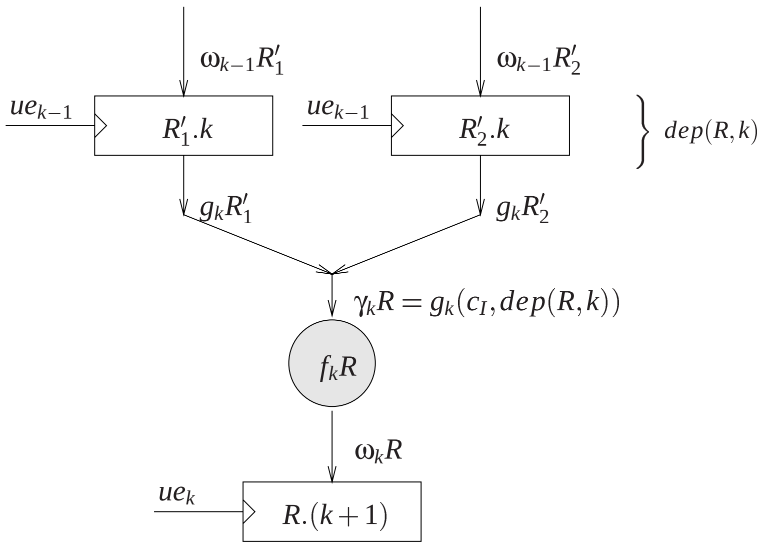
$$\Gamma_k R(c) \quad = \quad G_k(c, dep(R,k))$$

The functions $G_k R$ can now be specified recursively: If R is a specification register, $G_k R$ is:

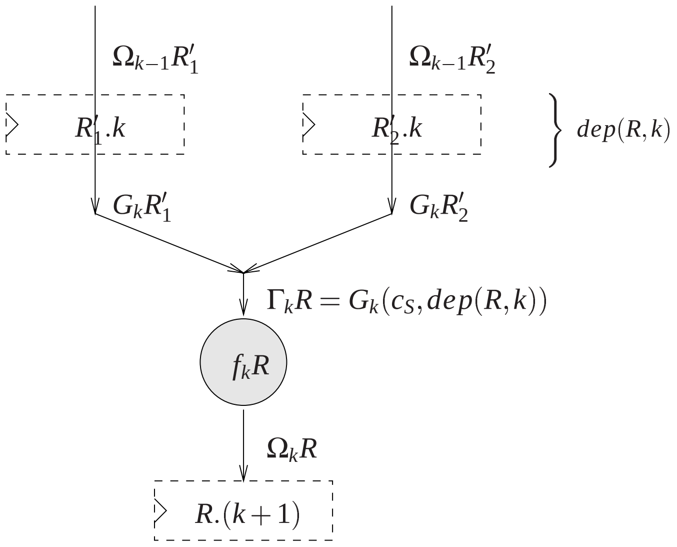
$$G_k R(c) \quad = \quad c.R$$

This allows a straightforward definition of $G_k R$ if R is an implementation register. Since an instance of R must be in the previous stage, the correct value of $R.k$ is used, i.e., $\Omega_{k-1}R(c)$:

$$G_k R(c) \quad = \quad \Omega_{k-1}R(c)$$



Sequential prepared machine



Machine without implementation registers

Figure 3.8 Relationship between ω_kR, g_kR' and Ω_kR, G_kR', depicted for two stages k and $k+1$. Let R'_1 and R'_2 be implementation registers and let R be an implementation register that depends on R'_1 and R'_2. The read accesses to R'_1 and R'_2 are unconditional.

Remember that $f\Gamma_k Rre(c)$ is just a shorthand for

$$f_k Rre(\Gamma_k Rre(c))$$

as described in section 3.2.8 (page 51).

For a conditional read access to a specification register, $G_k R$ is defined as follows:

$$G_k R(c) \quad = \quad \begin{cases} c.R & : \quad f\Gamma_k Rre(c) \\ 0 & : \quad \text{otherwise} \end{cases}$$

For a read access with read address to a specification register, $G_k R$ is defined as follows:

$$G_k R(c) \quad = \quad \begin{cases} c.R[f\Gamma_k Rra(c)] & : \quad f\Gamma_k Rre(c) \\ 0 & : \quad \text{otherwise} \end{cases}$$

Step 2 A set of lemmas is claimed and asserted later. For each specification register $R \in out(k)$, one lemma is used:

> *Using the correct input values, the register transition function $f_k R$ provides the correct output value.*

Let the inputs be calculated using values from configuration c_S^i. In this case, the output values can be found in configuration c_S^{i+1} of the specification machine.

These lemmas assert the correctness of the non-scheduled implementation described above, i.e., the sequential machine that performs the calculation of a configuration of the specification machine within one transition without implementation registers. The lemmas are therefore called *register transition function correctness lemmas*.

◀ Lemma 3.13

If the write access to the register is neither conditional nor has a write address, the claim is:

$$c_S^{i+1}.R \quad = \quad f\Gamma_k R(c_S^i)$$

If the write access to the register is conditional, the value of the transition function is used only if the write enable signal is active. If the write enable signal is not active, one takes the value from the previous configuration. The claim therefore is:

$$c_S^{i+1}.R \quad = \quad \begin{cases} f\Gamma_k R(c_S^i) & : \quad f\Gamma_k Rwe(c_S^i) \\ c_S^i.R & : \quad \text{otherwise} \end{cases}$$

In case of a write access with write address, the claim is for all addresses possible write addresses x:

$$c_S^{i+1}.R[x] \quad = \quad \begin{cases} f\Gamma_k R(c_S^i) & : \quad f\Gamma_k Rwe(c_S^i) \wedge \\ & \quad\;\; f\Gamma_k Rwa(c_S^i) = x \\ c_S^i.R[x] & : \quad \text{otherwise} \end{cases}$$

Step 3 Let T be a cycle. A **stage correctness predicate** $\mathsf{P}_k(T)$ is defined for each stage. It will be used later on in the proofs of all central claims.

The predicate $\mathsf{P}_k(T)$ holds iff the values of the registers of stage k are correct cycle T. This comprises both the implementation and the specification registers. Let $s\mathsf{P}_k(T)$ denote the stage correctness predicate for the specification registers and let $i\mathsf{P}_k(T)$ denote the stage correctness predicate for the implementation registers:

$$\mathsf{P}_k(T) \quad \Longleftrightarrow \quad s\mathsf{P}_k(T) \wedge i\mathsf{P}_k(T)$$

The stage correctness predicate $s\mathsf{P}_k(T)$ for the specification registers is given in analogy to the data consistency criterion in theorem 3.12: the values of the specification registers must match the values of the corresponding registers in the configuration of the specification machine indicated by the scheduling function. Thus, for all specification registers $R \in out(k)$ the following condition must hold:

$$R_I^T \quad = \quad R_S^{sI(k,T)}$$

The stage correctness predicate for the implementation registers is given using the notion of a correct implementation register as defined in step 1. For all implementation registers $R \in out(k)$ the following condition must hold:

$$R_I^T.(k+1) = \begin{cases} 0 & : \quad sI(k,T) = 0 \\ \Omega_k R(c_S^{sI(k,T)-1}) & : \quad \text{otherwise} \end{cases}$$

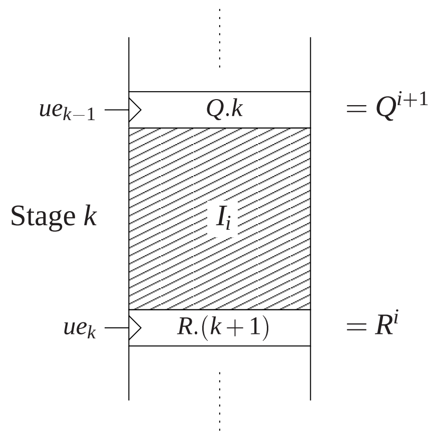


Figure 3.9 Illustration of the values in the registers if instruction I_i is in stage k (i.e., $sI(k,T) = i$). Q is a specification register in $out(k-1)$, R is a specification register in $out(k)$.

The stage correctness predicate for the implementation registers is motivated as follows: The stage correctness predicate for the implementation registers is supposed to provide information about the value of an implementation register $R \in out(k)$ during cycle T, i.e., about $R_I^T.(k+1)$. In case of $sI(k,T) = 0$, the register has never been written before, i.e., it has still the initial value, which is zero by definition:

$$R_I^T.(k+1) \quad = \quad 0$$

If $i = sI(k,T) > 0$, the last time the register was written was on calculating a part of configuration c^i (figure 3.9). Suppose this was done during cycle T'. By definition of the transition function, the following value was written:

$$\omega_k R(c_I^{T'})$$

This value was not changed since cycle T', thus, it is still in the register during cycle T:

$$R_I^T.(k+1) \quad = \quad \omega_k R(c_I^{T'})$$

In case of correct calculations, the inputs used by $\omega_k R$ for the transition function $f_k R$ during cycle T' while calculating configuration c^i were taken from configuration c^{i-1}. Thus, the right-hand side is:

$$R_I^T.(k+1) \quad = \quad \Omega_k R(c_S^{i-1})$$

This motivates the stage correctness predicate for implementation registers.

Lemma 3.14 ▶ All stage correctness predicates hold for the initial cycle, i.e., $\mathsf{P}_k(0)$ holds for all stages k.

PROOF In the initial cycle, the value of all stage scheduling functions is zero. One therefore has to show that the values of the specifications registers in the implementation machine during cycle 0 match the values of the corresponding registers in the specification. Since this is exactly the definition of c_I^0, the claim follows immediately. QED

3.5.5 Correctness of the Transition Functions

Lemma 3.15 ▶ The register transition function lemmas, as defined in step 2, hold.

PROOF The stages IF and EX do not write any specification register, thus, there is nothing to show.

Note that the following proofs are given here for illustration only. In PVS, the proofs are much simpler, since PVS is able to expand the definitions of the functions f_kR, Γ_kR, and G_kR automatically. Furthermore, the lemmas that show the correctness of circuits such as the ALU can be applied automatically. The proofs rely on definition expansion and trivial use of lemmas only, thus, the proofs below have just a few lines in PVS and require almost no manual interaction.

Stage ID Stage ID writes the specification registers DPC and PC'. The claim of the register transition function lemma for register DPC is:

$$f_1DPC(\Gamma_1DPC(c_S^i)) \quad \overset{!}{=} \quad c_S^{i+1}.DPC$$

Expanding the function Γ_1DPC on the left hand side, this is equal to:

$$f_1DPC(G_1(c_S^i, PC')) \quad \overset{!}{=} \quad c_S^{i+1}.DPC$$

Expanding the function G_1 on the left hand side, this is equal to:

$$f_1DPC(G_1PC'(c_S^i)) \quad \stackrel{!}{=} \quad c_S^{i+1}.DPC$$

Since PC' is a specification register, by definition of G_1PC' this is equal to:

$$f_1DPC(c_S^i.PC') \quad \stackrel{!}{=} \quad c_S^{i+1}.DPC$$

Since f_1DPC is just the identity (equation 3.12 page 57), this claim simplifies to:

$$c_S^i.PC' \quad \stackrel{!}{=} \quad c_S^{i+1}.DPC$$

This holds because of the definition of $c_S^{i+1}.DPC$ (equation 2.3 page 31).

The claim of the register transition function lemma for register PC' is:

$$f_1PC'(\Gamma_1PC'(c_S^i)) \quad \stackrel{!}{=} \quad c_S^{i+1}.PC'$$

The calculation of PC' depends on the first GPR operand. The functions for this operand use *GPRa* as register and not GPR in order to distinguish them from the functions of the second GPR operand.

Repeatedly expanding definitions as above, the claim is equal to:

$$f_1PC'(G_1IR(c_S^i), G_1GPRa(c_S^i), G_1PC'(c_S^i)) \quad \stackrel{!}{=} \quad c_S^{i+1}.PC'$$

By definition of f_1PC' (equation 3.13 page 57), this is equal to:

$$nextpc\_imp(G_1IR(c_S^i), G_1GPRa(c_S^i), G_1PC'(c_S^i)) \quad \stackrel{!}{=} \quad c_S^{i+1}.PC'$$

By lemma 3.4 (correctness of *nextpc_impl*), this is equal to:

$$nextpc(G_1IR(c_S^i), G_1GPRa(c_S^i), G_1PC'(c_S^i)) \quad \stackrel{!}{=} \quad c_S^{i+1}.PC'$$

There are three cases:

- If the instruction is a jump register or branch instruction, the register transition function $f_1 PC'$ reads IR, the first GPR operand, and the old value of the PC' register. These values are passed to the function *next pc*.

  One easily shows the correctness of the IR argument by expanding definitions, since IR is an implementation register:

  $$\begin{aligned} G_1 IR(c_S^i) &= \Omega_0 IR(c_S^i) \\ &= f_0 IR(c_S^i.DPC) \\ &= IM[c_S^i.DPC] \end{aligned}$$

  The correctness of the PC' argument is shown easily:

  $$G_1 PC'(c_S^i) = c_S^i.PC'$$

  The correctness of the GPR operand is assured as follows: the first GPR operand is read using a conditional read access with address. In case the read enable signal holds (equation 3.7), the GPR register with address $I\_RS1(IR)$ is read (equation 3.9):

  $$G_1 GPRa(c_S^i) = c_S^i.GPR[I\_RS1(IM[c_S^i.DPC])]$$

  This is exactly $op1$, as required by the specification (equation 2.1 page 30).

  If the condition does not hold, zero is returned by the G_1 function:

  $$G_1 GPRa(c_S^i) = 0$$

  In case of a jump register or branch instruction this happens only if $I\_RS1(IR)$ is zero. In this case, register GPR_0 is read, which is always zero when read, as required by equation 2.1.

- If the instruction is neither a jump nor branch instruction, the value of IR, zero, and the old value of PC' is passed to *next pc*. In this case, *next pc* ignores the value of the second argument and returns the correct result therefore.

- If the instruction is a jump instruction, *next pc* does not use the second argument, the GPR operand. The offset to the PC is provided by the immediate constant.

Stage M The memory stage writes a data word into the main memory in case of a store instruction. The write access to the data memory is a conditional write access with write address.

Most transition functions depend on the implementation register IR. Exemplary, it is shown how to assert the correctness of the IR arguments. By definition of G_3IR, one shows that $\Omega_2IR(c_S^i)$ is the value read. Repeatedly expanding the Ω functions and proceeding as above, one shows the correctness of the IR arguments:

$$\begin{aligned} G_3IR(c_S^i) &= \Omega_2IR(c_S^i) \\ &= \Omega_1IR(c_S^i) \\ &= \Omega_0IR(c_S^i) \\ &= f_0IR(c_S^i.DPC) \\ &= IM[c_S^i.DPC] \end{aligned}$$

As described above, the write access to DM is conditional and has a write address. In case the write enable signal $f_3DMwe(G_3IR(c_S^i))$ does not hold, nothing has to be shown. In the case that $f_3DMwe(G_3IR(c_S^i))$ holds, one has to show that the data value written is correct and that the write address is correct.

The data value written is read from the $MDRw$ register from the previous stage. By definition, the correct value of the $MDRw$ register is the correct value of the B register written by the decode stage. The decode stage places the second operand here. The correctness of this operand is asserted as described in the section above.

The address used for the write access to DM is taken from the function f_3DMwa. The function reads MAR from the previous stage and strips the two least significant bits. The correct value of the MAR register is by definition the output of the ALU. The ALU performs an addition, which is shown easily using that $f_3DMwe(G_3IR(c_S^i))$ holds and using the correctness of the ALU (lemma 2.14). Furthermore, it is shown easily that the second operand of the ALU is the immediate constant. The first operand of the ALU is generated by the decode stage. The correctness of this operand is shown as in the proof for the stage ID. Thus, exactly the effective address, as required by the specification, is used as write address for the write access.

Stage WB The write back stage writes the GPR destination operand of the instruction. The proof is similar to the proof used for the memory stage.

A conditional write access with address to GPR is performed. In the case that the write enable signal is active, one has to show that the data value written is correct and that the write address is correct.

In case of a load instruction, the data value is taken from the $shift4load$ circuit, which takes MAR, $MDRr$, and IR as inputs. The correctness of the value in the MAR register is asserted as described above. The correctness of the value in the $MDRr$ register is asserted as follows: the register is written by the memory stage using a conditional read access with address to DM. It is easy to show that the read enable signal of this read access is active using that the write enable signal holds. The correctness of the address of the memory access is shown as described above.

If the instruction is not a load instruction, the data value is taken from the C register. The C register is passed unmodified by the memory stage from the execute stage. In case of an ALU/shift instruction, the correctness of this value is asserted as follows: the correctness of the input operands is asserted as described above; using the correctness of the ALU (lemma 2.14), the correctness of the result is shown.

In case of a jump and link instruction, the execute stage passes the value of the C register from stage ID. This is the correct value of the PC' register, as required by the specification.

The correctness of the index used for the write access to GPR is shown easily by definition unfolding. The index is written into an implementation register by stage ID and not changed in any subsequent stage. QED

Correctness of the Functions $\mathbf{g}_k R$ In the proofs above, the correctness of the inputs of each stage is assumed. In real hardware, the implementations of the functions g_kR are used in order to generate the input operands. It is therefore left to show that the values generated by the functions g_kR actually match the correct values.

Lemma 3.16 ► Let $sI(k,T) = i$ and $full_k^T$ hold. Assuming that the stage correctness predicates P_j hold in all cycles up to cycle T, the inputs generated by the functions g_kR during cycle T are correct:

$$g_kR(c_I^T) \quad = \quad G_kR(c_S^i)$$

PROOF In case of an implementation register, the correct value on the right-hand side is defined by the function $\Omega_k R$:

$$g_k R(c_I^T) \overset{!}{=} \Omega_{k-1} R(c_S^i)$$

In case of a specification register, the correct value is given in the configuration of the specification machine. If the read access is neither indexed nor conditional:

$$g_k R(c_I^T) \overset{!}{=} R_S^i$$

In case of a conditional read access, the correct value is zero if the read enable signal is not active. If the signal is active, the correct value is the same as in the case above.

$$g_k R(c_I^T) \overset{!}{=} \begin{cases} R_S^i & : \ f\Gamma_k Rre(c_S^i) \\ 0 & : \ \text{otherwise} \end{cases}$$

In case of an indexed read access, the correct value is defined using the correct value of the address. Let x denote this value:

$$\begin{aligned} x &:= f\Gamma_k Rra(c_S^i) \\ g_k R(c_I^T) &\overset{!}{=} \begin{cases} R_S^i[x] & : \ f\Gamma_k Rre(c_S^i) \\ 0 & : \ \text{otherwise} \end{cases} \end{aligned}$$

The proof depends on the type of the register that is read and in which stage the register is. The first thing is to show the correctness for the case that neither a condition nor a read address is used.

1. Let the register that is to be read be an implementation register. By the definition of $g_k R$, the register from the previous stage is taken:

$$g_k R(c_I^T) = c_I^T.R.k \tag{3.16}$$

   An implementation register is never read in stage $k = 0$, and one therefore can use lemma 3.11 and the fact that the full signal $full_k^T$ is active in order to conclude that $T \geq 1$ holds. For this case, invariant 3.3 (page 65) states:

$$sI(k-1, T) = i + 1$$

The stage correctness predicate for $R \in impl$, cycle T, and stage $k-1$ states:

$$R_I^T.k = \begin{cases} 0 & : \quad sI(k-1,T) = 0 \\ \Omega_{k-1}R(c_S^{sI(k-1,T)-1}) & : \quad \text{otherwise} \end{cases}$$

Since $sI(k-1,T) = i+1$ is never zero, this simplifies to:

$$R_I^T.k \quad = \quad \Omega_{k-1}R(c_S^i)$$

Remember that $R_I^T.k$ just denotes $c_I^T.R.k$. Thus, one can insert this into equation 3.16. This changes equation 3.16 into:

$$g_kR(c_I^T) \quad = \quad \Omega_{k-1}R(c_S^i) \tag{3.17}$$

This is exactly the claim.

2. Let the register that is to be read be a specification register that is in the same stage in which it is read. By definition of g_kR, the value of R is read:

$$g_kR(c_I^T) \quad = \quad c_I^T.R.(k+1) \tag{3.18}$$

By using the stage correctness predicate for specification register R, stage k, cycle T, this is transformed into:

$$c_I^T.R.(k+1) \quad = \quad R_S^{sI(k,T)} \tag{3.19}$$

Since $i = sI(k,T)$, this is the claim.

3. Let the register that is to be read be a specification register that is in a later stage than the stage it is read in. Let w be $stage(R)$. By definition of g_kR, the value of R is read:

$$g_kR(c_I^T) \quad = \quad c_I^T.R.(w+1) \tag{3.20}$$

By applying the stage correctness predicate for stage w and cycle T, this transforms into:

$$g_kR(c_I^T) \quad = \quad R_S^{sI(w,T)} \tag{3.21}$$

For cycle $T=0$, both $sI(w,0)$ and $sI(k,0)$ are zero by definition of sI. Thus, the claim holds.

For cycles $T > 0$, one uses lemma 3.10 for cycle T and stages k and w. Because of $full_k^T$ and $w > k$, lemma 3.10 shows that

$$\begin{aligned} sI(w,T) &= sI(k,T) \\ &= i \end{aligned}$$

holds. This concludes the claim.

This shows the claim for inputs without index and condition. The claim for inputs with index or condition is shown as follows: Since the inputs for the functions f_kRre and f_kRra never use a condition or an index, the correctness of the inputs of these functions can be shown as above. If the condition does not hold, the claim obviously holds. If the condition holds, the proof proceeds as above. In case of an indexed access, the claim is shown using the arguments above and that the index is correct. QED

◀ Lemma 3.17

Let T' be greater than zero. Assuming all stage correctness predicates for the cycle $T'-1$, the predicate for stage k holds for cycle T'.

$$(\forall l : \mathsf{P}_l(T'-1)) \implies \mathsf{P}_k(T')$$

PROOF

Let the update enable signal ue_k^{T-1} be active. In this case, one uses invariant 3.1 in order to conclude that $sI(k,T-1) = i-1$. This allows using lemma 3.16 for cycle $T-1$ and configuration $i-1$. The lemma shows that the inputs of the stage transition functions are correct. In case of a specification register, lemma 3.15 is used to show that the output written in the register is correct. In case of an implementation register, the output value of the stage matches the correct value by definition of the correct value of an implementation register.

If the update enable signal ue_k^{T-1} is not active, invariant 3.1 is used to show that the value of the stage scheduling function does not change from cycle $T-1$ to cycle T. Since the update enable signal is not active, the values in the registers do not change from cycle $T-1$ to cycle T, which shows the claim. QED

◀ Theorem 3.18

All stage predicates hold for all cycles.

PROOF

This is shown by induction on T. The case $T = 0$ is subsumed by lemma 3.14, the induction step is shown using lemma 3.17.

Theorem 3.18 obviously implies the data consistency criterion as proposed in theorem 3.12.

3.6 Liveness

3.6.1 Introduction

The liveness criterion used in this thesis is that the implementation machine actually calculates any desired configuration of the specification machine within a finite amount of time. In order to prove the liveness criterion for the prepared sequential machine, a formal notion of "will happen in finite time" is required.

Definition 3.3 ▶ *Time Predicate* A time predicate is a mapping from $\mathbb{N}_0$ to B.

The constant time predicates *always* and *never* are defined as follows:

$$always(T) = true \tag{3.22}$$
$$never(T) = false \tag{3.23}$$

Let *pred* be a time predicate. The following notation is used:

$$\exists pred \quad :\Longleftrightarrow \quad \exists T \in \mathbb{N}_0 : \ pred(T) \tag{3.24}$$

The operator $\exists^{\geq T'}$ on a time predicate holds iff the predicate is true for a time $T \geq T'$.

$$\exists^{\geq T'} pred \quad :\Longleftrightarrow \quad \exists T \in \mathbb{N}_0 : (pred(T) \wedge T \geq T') \tag{3.25}$$

Lemma 3.19 ▶ If there exists a time $T \geq T'$ with $pred(T)$, also a time T'' exists that is the smallest $T'' \geq T'$ satisfying the predicate.

PROOF Let S be the set of natural numbers that are greater or equal T' and satisfy the predicate. The set is non-empty and has a lower bound. The minimum $\min(S)$ exists therefore and is T''.

Definition 3.4 ▶ *Finite False* Let *pred* be a time predicate. The predicate is called *finite false* iff for all T $\exists^{\geq T} pred$ holds. This implies that if $pred(T)$ does not hold for a given T, there is a finite $T' \geq T$ such that $pred(T')$ holds. In analogy to that, a predicate is called *finite true*, iff $\overline{pred}$ is finite false.

3.6.2 Liveness Criterion

Let c_S^i be any desired configuration of the specification machine. The implementation machine is said to be alive iff for all stages k there exists a time $T \in \mathbb{N}_0$ with $sI(k,T) = i$:

$$\exists T \in \mathbb{N}_0 : sI(k,T) = i$$

3.6.3 Liveness Properties of the Scheduling Logic

Let ue_k denote the time predicate of the update enable signal of stage k. Let $stall_k$ denote the time predicate of the stall signal of stage k.

◄ Lemma 3.20

Let $full_k^T$ hold for a stage k and a cycle T. Let the stall signal $stall_k$ be finite true (thus, it becomes false within a finite amount of time). This implies that ue_k becomes true within a finite amount of time:

$$full_k^T \implies \exists^{\geq T} ue_k$$

PROOF

Since $stall_k$ is finite true, there exists a cycle $T' \geq T$ such that the stall signal is not active. Let T' be the smallest value with this property. If $T' = T$, $full_k^{T'}$ holds by premise.

If $T' > T$, assume that $full_k^{T'}$ does not hold. According to lemma 3.2, this implies that $stall_k^{T'-1}$ does not hold. This is a contradiction to the assumption that T' is the smallest value.

Since $full_k^{T'} = 1$ and $stall_k^{T'} = 0$, $ue_k^{T'}$ holds by definition.

QED

◄ Lemma 3.21

Assuming that all stall signals are finite true, and that the update enable signal of stage 0 will be active within finite time after cycle T, the update enable signal of stage k will be active within finite time after cycle T.

$$\exists^{\geq T} ue_0 \implies \exists^{\geq T} ue_k$$

PROOF

This is shown by induction on k. For $k = 0$, the claim is subsumed by the premise.

For $k+1$, the induction premise states that there is a cycle $T' \geq T$ with $ue_k^{T'}$. By the transition function of the full bits, $full_{k+1}^{T'+1}$ holds. Lemma 3.20 concludes the claim.

Lemma 3.22 ▶ Assuming that all stall signals are finite true, the update enable signals are finite false.

PROOF The claim for ue_0 is that for all T there is a $T' \geq T$ such that $ue_0^{T'}$ holds. This is shown by induction on T. For $T = 0$, one uses lemma 3.20 and the fact that $full_0^0$ holds by definition.

For $T+1$, lemma 3.21 is used to argue that there exists a $T' \geq T$ such that $ue_{n-1}^{T'}$ holds. According to the transition function of the full bits, $full_0^{T'+1}$ holds. Lemma 3.20 is used to show the claim.

The claim for ue_k with $k \geq 1$ is shown by induction on k. For $k = 0$, the claim is shown already. For $k+1$, the claim is shown as in lemma 3.21. QED

3.6.4 Liveness Proof for the Sequential DLX

Lemma 3.23 ▶ Let the update enable signal ue_k of a stage k be off during the cycles T'' with $T' > T'' \geq T$. The value of the scheduling function does not change from cycle T to T'.

$$\forall T'' | T' > T'' \geq T : \overline{ue_k^{T''}} \quad \Longrightarrow \quad sI(k,T) = sI(k,T')$$

PROOF The proof proceeds by induction on T' and by definition unfolding.

Theorem 3.24 ▶ Assuming that all stall singnals are finite true, the machine is alive.

PROOF This is shown by induction on i. For $i = 0$, the claim is that there is a T such that $sI(k,T) = 0$ holds. By definition of sI, $T = 0$ satisfies this.

For $i+1$, the induction premise states that there is a cycle T such that $sI(k,T) = i$ holds. According to lemma 3.22), the update enable signal ue_k is finite false,

Thus, there is a cycle $T' \geq T$ such that $ue_k^{T'}$ holds. Let T' be the smallest value that satisfies this. If T' is equal to T, the claim holds by invariant 3.1.

If $T' > T$, lemma 3.23 states that $sI(k,T)$ is equal to $sI(k,T')$. Invariant 3.1 shows that $sI(k,T'+1)$ is $i+1$.

3.7 Literature

The concept of the prepared sequential machine and the DLX implementation is taken from [MP00]. There are many publications on the verification of sequential machines, e.g., Cohn verified the VIPER [Coh87], Joyce verified the Tamarack [Joy88a, Joy88b], Hunt verified the FM8501 [Hun94], and Windley verified the AVM-1 [Win95].

There is not much literature on the verification of liveness properties of microprocessors. However, liveness verification is critical. In [MP96], deadlocks in the original version of the well known scoreboard scheduler are described. Furthermore, a corrected version is presented and its liveness is proven.

Chapter 4

Pipelined Machines

4.1 Scheduling the Pipelined Machine

4.1.1 Introduction

THE PREPARED SEQUENTIAL MACHINE, as described in the previous chapter, calculates a configuration of the specification machine within *n* transitions if no external stall condition arises, with *n* being the number of stages. In each transition of the prepared sequential machine, only one stage is in use. The data paths of the remaining stages are left idle.

In this chapter, the prepared sequential machine M_σ is transformed into a pipelined machine M_π, which allows running all stages in parallel. This concept is taken from [MPK00, MP00]. In contrast to the cited literature, an automated tool is used in order to do the transformation including the generation of stalling and forwarding logic. Furthermore, the transformation is not limited to microprocessors. Any prepared sequential machine as specified in the previous chapter can be transformed into a pipelined design.

The goal of the transformation is to use the formerly idle data paths in order to speed up the calculation of the desired configurations of the specification machine. As before, the registers of all stages are initialized

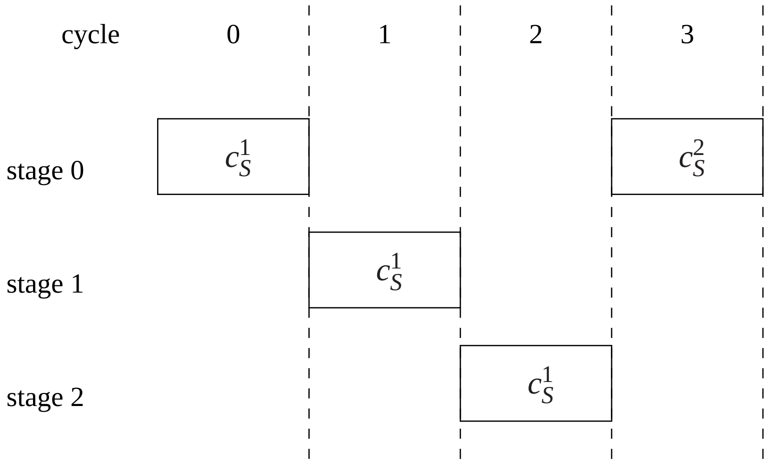


Figure 4.1 Scheduling of the prepared sequential machine with $n = 3$ stages in the absence of external stalls.

with the values of c_S^0 and the first stage starts with the calculation of c_S^1. In the next cycle, the second stage starts with the calculation of c_S^1, as before. In contrast to the prepared sequential machine, the first stage does not idle but starts the calculation of c_S^2.

In particular, the calculation of the configuration c_S^2 starts before the calculation of configuration c_S^1 is finished since stage 0 calculates only some parts of the configuration. Figure 4.1 shows how the prepared sequential machine calculates the configurations, and figure 4.2 shows how the pipelined machine uses the formerly idle stages to speed up this calculation.

The calculation of configuration c_S^1 is finished in cycle 2, as in the sequential machine. In contrast to the sequential machine, the calculation is of configuration c_S^2 is finished already in cycle 3.

A stage ”runs” if the corresponding update enable signal is active. A stage is updated if it is full and not stalled. The first step of the transformation therefore is to modify the machine such that there are as many full stages as possible.

In the new initial configuration, no full bit is set in contrast to the initial configuration of the prepared sequential machine. The definition of the signal $full_0$ is changed: the first stage is defined to be always full, since one can start with the calculation of the next configuration any time the

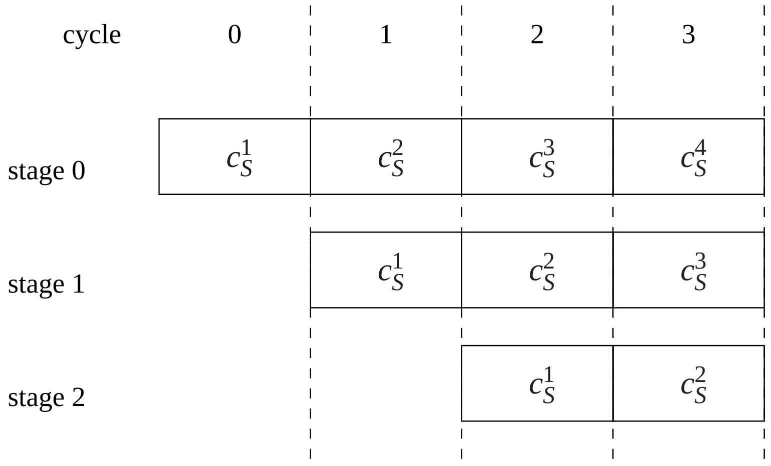


Figure 4.2 Running all stages in parallel.

	$T=0$	$T=1$	$T=2$	$T=3$	$T=4$	$T=5$	$T=6$
ue_0^T	1	1	1	1	1	1	1
ue_1^T	0	1	1	1	1	1	1
ue_2^T	0	0	1	1	1	1	1
ue_3^T	0	0	0	1	1	1	1

Table 4.1 The update enable signals of a four stage pipeline in the absence of stalls

stage would be empty otherwise.

$$full_0(c) \quad := \quad 1$$

This is the only change required in order to get a pipelined schedule. This full bit is propagated to the next stage in each transition just as in the machine M_σ. Thus, if there is no stall signal, the full bits are never cleared after they are set. Table 4.1 illustrates the values of the update enable signals for a four stage pipeline and after applying this modification and assuming that no stall signal is active.

After n transitions, all stages work in parallel therefore. Every stage calculates a part of a different configuration of the reference machine. For example, let stage 2 calculate parts of c_S^i. In this case, stage 1 calculates parts of c_S^{i+1} and stage 3 calculates parts of c_S^{i-1}. This is depicted in figure 4.3.

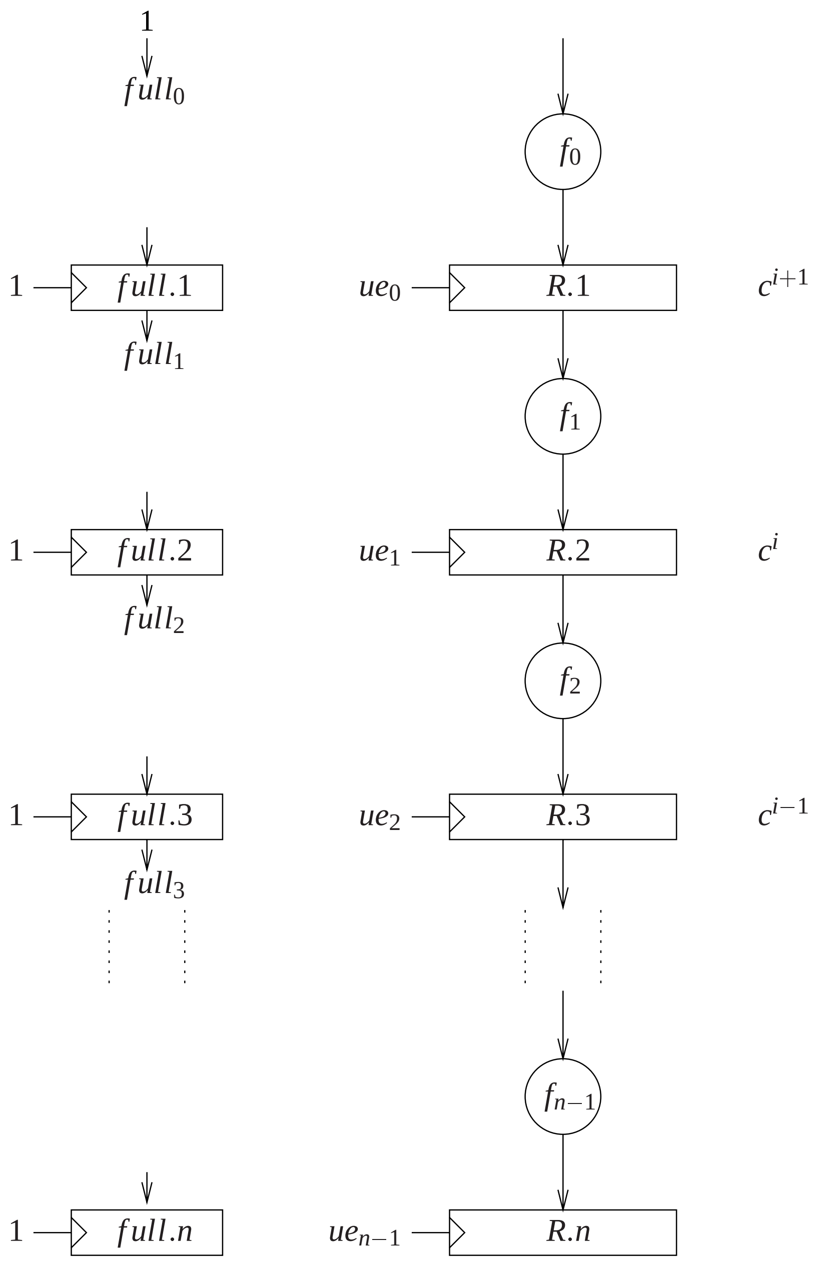


Figure 4.3 The structure of the pipelined machine

	$T=0$	$T=1$	$T=2$	$T=3$	$T=4$	$T=5$	$T=6$
$sI(0,T)$	0	1	2	3	4	5	6
$sI(1,T)$	0	0	1	2	3	4	5
$sI(2,T)$	0	0	0	1	2	3	4
$sI(3,T)$	0	0	0	0	1	2	3

Table 4.2 The values of sI in a four stage pipelined machine in the absence of stalls

The new values of the update enable signals affect the values of the scheduling function also since the scheduling function is defined using the update enable signals. Table 4.2 illustrates the values of $sI(k,T)$ for the first seven cycles.

4.1.2 Scheduling Lemmas

The following simple lemmas are concluded from the new definition of the full signals:

◄ Lemma 4.1

A stage is full iff it was updated or stalled in the previous cycle:

$$\forall k \geq 1 : \quad full_k^{T+1} = ue_{k-1}^T \vee stall_k^T$$

The signal $full_0$ is always active:

$$full_0^T = 1$$

All other signals $full_k$ are not active during cycle 0:

$$\forall k \geq 1 : \quad full_k^0 = 0$$

This lemma is a counterpart of lemma 3.2 of the sequential machine.

PROOF

This lemma is an implication of the transition function of the full bits and of the definition of the full signals.

◄ Convention 4.2

In analogy to convention 3.1, it is required that if a stage is not full, it must not be stalled:

$$\overline{full_k^T} \quad \Longrightarrow \quad \overline{stall_k^T}$$

In addition to that, it is required that if a stage is stalled and the previous stage is full, the previous stage must be stalled also:

$$\forall k \geq 1: \quad full_{k-1}^T \wedge stall_k^T \quad \Longrightarrow \quad stall_{k-1}^T$$

Using the lemma and the convention above, it is easy to show that the pipeline has the same properties like a simple queue: no entry in the queue is lost and no entry in the queue is duplicated. These properties are subsumed by three trivial lemmas.

The equations for the stall signals of the prepared sequential machine in chapter 3 also comply with this extended convention, which is shown easily using lemma 3.8. Thus, all properties of the pipelined machine concluded using this convention also hold in the sequential machine.

Lemma 4.3 ► If a stage is full and is updated, the next stage is updated, too.

$$\forall k \geq 1: \quad full_k^T \wedge ue_{k-1}^T \quad \Longrightarrow \quad ue_k^T$$

This ensures that the contents of a stage are never overwritten without moving into the next stage.

PROOF According to the definition of the update enable signals, it is sufficient to show that $full_k^T$ and $\overline{stall_k^T}$ holds. According to the premise of the lemma, $full_k^T$ holds. By the definition of the update enable signal ue_{k-1}^T, $\overline{stall_{k-1}^T}$ and $full_{k-1}^T$ holds. The claim is concluded by convention 4.2.

Lemma 4.4 ► If a stage is full and if its output registers are not updated, the full bit is preserved.

$$\forall k \geq 1: \quad full_k^T \wedge \overline{ue_k^T} \quad \Longrightarrow \quad full_k^{T+1}$$

PROOF By the definition of the update enable signals, one concludes that $stall_k^T$ holds. The claim is concluded using lemma 4.1.

Lemma 4.3 and lemma 4.4 guarantee that no configuration in a given stage is ever lost.

◄ Lemma 4.5

If a configuration in a stage moves into the next stage (i.e., the output registers of a stage are updated), and if the next configuration is not clocked into the stage, the full bit is cleared:

$$\forall k \geq 1: \quad full_k^T \wedge ue_k^T \wedge \overline{ue_{k-1}^T} \quad \Longrightarrow \quad \overline{full_k^{T+1}}$$

This lemma guarantees that no configuration is duplicated.

PROOF

By the definition of the update enable signals, one concludes $\overline{stall_k^T}$. The claim is concluded by lemma 4.1.

The following lemma is the counterpart of lemma 3.11 in the sequential machine.

◄ Lemma 4.6

Stage k is full at the earliest in cycle k.

$$full_k^T \quad \Longrightarrow \quad T \geq k$$

PROOF

The proof proceeds by induction over T. For $T = 0$, the claim is concluded from lemma 4.1.

Assuming the lemma for cycle T, the claim for $T+1$ is shown as follows: For $k = 0$, the claim is obvious. Thus, the claim is shown for $k > 0$.

If $full_k^T$ or $full_{k-1}^T$ holds, one simply uses the induction premise. If $full_k^T$ and $full_{k-1}^T$ do not hold, one shows that $full_k^{T+1}$ cannot hold using lemma 4.1:

$$full_k^{T+1} \quad = \quad ue_{k-1}^T \vee stall_k^T$$

The rest of the proof proceeds as the proof of lemma 3.11 (page 71). QED

4.1.3 The Scheduling Invariants

In order to prove the data consistency of the pipelined machine, the three scheduling invariants presented for the prepared sequential machine in chapter 3 (page 65) will be used. We will therefore show that they also hold for the pipelined machine.

PROOF The proof of the invariants proceeds as in chapter 3: Let $P_i(T)$ denote that invariant i holds for the pipelined machine for the cycle T. The claim is concluded as in chapter 3:

$$\begin{array}{rcl} P_3(T-1) & \Longrightarrow & P_1(T) \\ P_1(T) \wedge P_2(T-1) \wedge P_3(T-1) & \Longrightarrow & P_2(T) \\ P_1(T) \wedge P_2(T-1) \wedge P_3(T-1) & \Longrightarrow & P_3(T) \end{array}$$

The proof of invariant 3.1 is identical to the proof presented in chapter 3. The proof depends on the definition of sI and invariant 3.3 only.

Proof of Invariant 3.2 The proof of invariant 3.2 presented in chapter 3 depends on lemma 3.8 ("exactly one stage full"), which no longer holds in the pipelined machine.

Let us consider the stages $k-1$ and k with $k > 0$. There are four cases regarding the update enable signals ue_k^{T-1} and ue_{k-1}^{T-1} of these stages:

1. Let both update enable signals be active. According to invariant 3.2, the values of the scheduling function of the stages in cycle $T-1$ are either equal or the value of the scheduling function of stage $k-1$ is greater by one. According to invariant 3.1, the values of both scheduling functions increase by one within the step to cycle T. The claim therefore holds by invariant 3.2 for cycle $T-1$.

2. Let both update enable signals be not active. According to invariant 3.1, the values of the scheduling function do not change and the claim follows from invariant 3.2 for cycle $T-1$ therefore.

3. Let the update enable signal of stage k be active and the update enable signal of stage $k-1$ be not active. This case is argued as in chapter 3.

4. Let the update enable signal of stage k be not active and the update enable signal of stage $k-1$ be active. Let the first case given by invariant 3.2 for cycle $T-1$ hold, i.e., $sI(k-1,T-1)$ is equal to $sI(k,T-1)$. Using invariant 3.1, $sI(k-1,T) = sI(k,T)+1$ is concluded.

   Let the second case given by invariant 3.2 for cycle $T-1$ hold, i.e., $sI(k-1,T-1) = sI(k,T-1)+1$ holds. According to invariant 3.3, $full_k^{T-1}$ holds. According to lemma 4.3, one can conclude ue_k^{T-1} from $full_k^{T-1} \wedge ue_{k-1}^{T-1}$. This is a contradiction since $\overline{ue_k^{T-1}}$ was assumed.

Proof of Invariant 3.3 The proof of invariant 3.3 presented in chapter 3 also depends on lemma 3.8, which no longer holds in the pipelined machine.

For $T = 0$, the claim can be shown by definition unfolding and using lemma 4.6. For $T > 0$, according to lemma 4.1, the claim is equivalent to:

$$ue_{k-1}^{T-1} \vee stall_k^{T-1} \iff sI(k-1,T) = sI(k,T)+1$$

The proof proceeds by a full case split on the values of the update enable bits ue_{k-1}^{T-1} and ue_k^{T-1}, as done in the proof of invariant 3.2. There are four cases:

1. If both update enable signals are on, it is left to show that

   $$sI(k-1,T) = sI(k,T)+1$$

   holds. According to invariant 3.1, this is equivalent to:

   $$sI(k-1,T-1)+1 = sI(k,T-1)+2$$

   It is sufficient to show that $full_k^{T-1}$ holds because of invariant 3.3 for the previous cycle. This is done using the definition of ue_k^{T-1}.

2. If ue_{k-1}^{T-1} is on and ue_k^{T-1} is off, one assumes the claim does not hold. Using the same arguments as in the proof of invariant 3.3 in chapter 3, one can conclude that $full_k^{T-1}$ holds.

   Using lemma 4.3, ue_k^{T-1} is concluded. This is a contradiction to the assumption above.

3. If ue_{k-1}^{T-1} is off and ue_{k}^{T-1} is on, the rest of the argumentation is identical to the proof in chapter 3.

4. If both update enable signals are off, the arguments in chapter 3 can be repeated using convention 4.2.

QED This concludes the claim.

4.2 Forwarding

4.2.1 Introduction

The new scheduling has impact on the calculation of the input values of the stages. Let stage k read an implementation register R. Read access to implementation registers is not affected by the changes to the scheduler, since according to the requirements of the hardware description language (section 3.2.4), an instance of the register R must be in $out(k-1)$. Thus, the value of the implementation register has been calculated by the previous stage. A formal proof for that claim uses the same arguments as given for the sequential machine.

However, the access to specification registers is affected. For the sequential machine, lemma 3.16 (page 82) shows that the value the register has in the previous configuration of the specification machine (as given by the scheduling function) is passed as input.

The goal is to modify the functions g_kR such that the same proposition can be made for the pipelined machine. This allows concluding that the values the pipelined machine writes into the registers match those written by the sequential machine. This proof method is taken from [MPK00, MP00].

Let stage k read specification register $R \in out(w)$. There are two cases, as $k > w$ is not allowed so far (section 3.2.5, page 43):

1. If the read access is done in the stage that writes the register, i.e., $k = w$, it is sure that the register still contains the value from the previous configuration as required. Nothing has to be changed in this case. The formal proof for this proposition is identical to the corresponding case in lemma 3.16.

An example is the read access to PC' in the decode stage of the DLX as implemented in chapter 3.

2. If the read access is done in a stage before the stage that writes the register, i.e., $k < w$, the access cannot be done, since the desired value is not in the register yet.

There are two methods to overcome the limitation in the last case: *forwarding* and, if this fails, *stalling*. These methods will be described in the next sections.

4.2.2 Forwarding from the Next Stage

Forwarding makes use of the parallelism of the calculations of the configurations. In the literature, forwarding is often also called bypassing [Fly95]. Let R be the specification register to be read. The technique used here is presented in [KPM00]. Let $stage(R) = w = k+1$, i.e., the register R is an output register of the next stage and do not let the read access have an address. There are two cases (figure 4.4):

- If $full.w$ is set, the stage w contains the configuration that the desired value is part of. Since the full bit is set, the stage is still busy and the desired value is not yet stored in the register. However, the register transition function of R provides the final value. Since $R \in out(w)$, $\omega_w R$ is this value.
- If $full.w$ is not set, there is either no previous configuration (the stage was never used after reset), or the previous configuration is already in the next stage. In the first case, the stage still contains the initial values, i.e., the values of c_S^0. In the second case, the calculation is already done and the result is stored in the register. In both cases, $R.(w+1)$ contains the desired value.

Thus, in order to realize forwarding from the next stage, it is sufficient to select between $\omega_w R$ and $R.(w+1)$ depending on the signal $full.w$.

This is formalized as follows: The function $g_k R$ is used in order to generate the input values for the register transition functions. The method described above allows defining $g_k R$ for the pipelined machine for the case

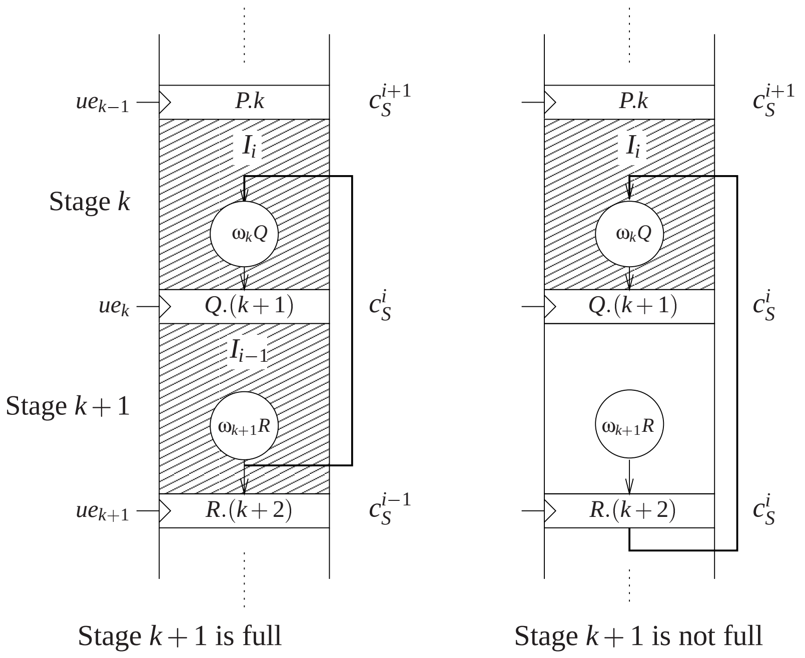


Figure 4.4 How forwarding is done from the next stage: the calculation of $Q.(k+1)$ depends on $R.(k+2)$. If stage $k+1$ is full, the output of stage $k+1$ is taken. If not, the value from the register is taken.

$w = k+1$ and $R \in spec$. If R is an implementation register, no forwarding is necessary and we use g_kR from the prepared sequential machine.

If R is a specification register with $w = k+1$ and the read access does not have an address, g_kR is:

$$g_kR(c) \quad = \quad \begin{cases} R.(w+1) & : \quad \overline{c.full.w} \wedge f\gamma_kRre(c) \\ \omega_wR(c) & : \quad c.full.w \wedge f\gamma_kRre(c) \\ 0 & : \quad \text{otherwise} \end{cases}$$

The following lemma asserts that the input generation function g_kR defined above provides the correct value. This lemma corresponds to lemma 3.16 (page 82) as used in the sequential machine.

◀ Lemma 4.7

Let $sI(k,T) = i$ and $full_k^T$ hold. Let R be a specification register and $w = k+1$. Assuming that the stage correctness predicates P_j hold in all cycles up to cycle T, the inputs generated by the function g_kR during cycle T are correct:

$$g_kR(c_I^T) \quad = \quad G_kR(c_S^i)$$

PROOF

Since R is a specification register, the correct value G_kR on the right-hand side of the claim is given in the configuration of the specification machine. Since the read access does not have an address, this is:

$$g_kR(c_I^T) \quad \overset{!}{=} \quad \begin{cases} R_S^i & : \quad f\Gamma_kRre(c_S^i) \\ 0 & : \quad \text{otherwise} \end{cases}$$

The correctness of the read enable signal is asserted as in the proof of lemma 3.16. After that, the claim is shown for the last stage, which is stage $n-1$, then for stage $n-2$, and so on until the claim is shown for all stages. In case of the last stage, there is nothing to show since there is no next stage to forward from. Assuming the claim holds for stage $k+1$, the claim is shown for stage k as follows:

There are two cases regarding the full bit $full.w^T$:

1. The full bit $full.w^T$ is set. Thus, invariant 3.3 states:

$$sI(k+1,T) \quad = \quad i-1$$

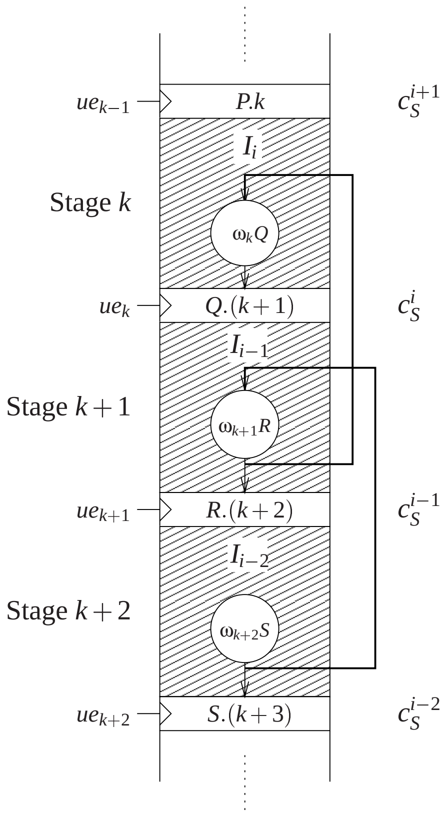


Figure 4.5 Forwarding multiple times: register Q depends on R, which depends on S.

The next thing to show is that the inputs of $f_w R$ are correct:

$$\gamma_w R(c_I^T) \quad \stackrel{!}{=} \quad \Gamma_w R(c_S^{sI(w,T)}) \tag{4.1}$$

This is done as follows: for implementation registers, one applies lemma 3.16. For specification registers that are in the same stage, one also uses lemma 3.16. For specification registers that are in stage $w+1$, forwarding from the next stage is done. For this case, the correctness is shown using the induction premise.

This situation is depicted in figure 4.5: register Q depends on register R, which depends on register S. The proof covers this situation by using induction as described. However, we do not recommend it since it results in a bad cycle time.

The correctness of the inputs implies that the outputs of the stage given by $\omega_w R(c^T)$ are also correct. Formally, lemma 3.13 is used. Let the write enable signal $f_w Rwe$ be active:

$$\begin{array}{rcll} g_k R(c_I^T) & = & \omega_w R(c_I^T) & \text{(by def. of } g_k R\text{)} \\ & = & f\gamma_w R(c_I^T) & \text{(by def. of } \omega\text{)} \\ & = & f\Gamma_w R(c_S^{sI(w,T)}) & \text{(eq. 4.1)} \\ & = & f\Gamma_w R(c_S^{i-1}) & \text{(inv. 3.3)} \\ & = & R_S^i & \text{(lemma 3.13)} \end{array}$$

If the write enable signal $f_w Rwe$ is not active, the function $\omega_w R$ takes the value from the register, i.e., $c_I^T.R$. According to the stage correctness predicate for stage $k+1$ and cycle T, this is equal to $R_S^{sI(w,T)}$. This is equal R_S^{i-1} because of $sI(w,T) = i-1$. This is equal to R_S^i because the register R is not changed since the write enable signal is not active. Formally, one uses lemma 3.13.

$$\begin{array}{rcll} g_k R(c_I^T) & = & \omega_w R(c_I^T) & \text{(by def. of } g_k R\text{)} \\ & = & c_I^T.R & \text{(by def. of } \omega\text{)} \\ & = & R_S^{sI(w,T)} & \text{(stage correctness)} \\ & = & R_S^{i-1} & \text{(inv. 3.3)} \\ & = & R_S^i & \text{(lemma 3.13)} \end{array}$$

This concludes the claim.

2. The full bit $full.w^T$ is not set. In this case, $g_k R$ is by definition:

$$g_k R(c_I^T) \quad = \quad R_I^T \tag{4.2}$$

Using the stage correctness for the value on the right-hand side, this is transformed into:

$$g_k R(c_I^T) \quad = \quad R_S^{sI(w,T)} \tag{4.3}$$

Invariant 3.2 shows that $sI(w,T)$ is either i or $i-1$. Invariant 3.3 shows that it is not $i-1$, thus, $sI(w,T)$ is i. This concludes the claim.

This concludes the correctness of the forwarding from the next stage. QED

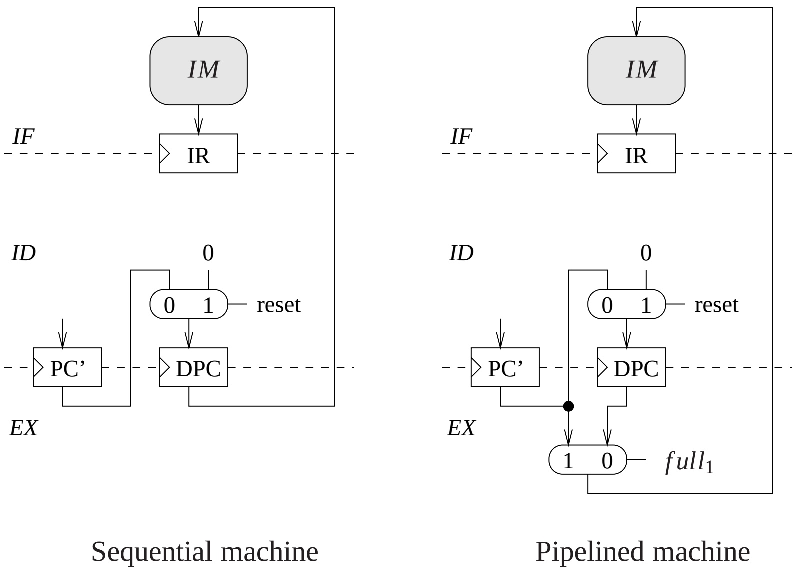


Figure 4.6 Transformation of the PC environment

Example: Forwarding of DPC in the DLX The DLX implementation in the previous chapter reads the register DPC in the instruction fetch stage (section 3.4.2 page 55). The register DPC is an output register of the decode stage. Thus, the read access to DPC in stage 0 can be realized using the function g_0DPC as defined above.

According to the definition above, the value read depends on the full bit of the stage that writes DPC, i.e., it depends on $full_1$. If $full_1$ is not set, the value from the register DPC.2 is taken. If $full_1$ is set, the value provided by the transition function of DPC is taken. The transition function of DPC reads PC' and outputs this value unmodified. If $full_1$ is set, one therefore takes the value of PC' as input in stage 0.

$$g_0DPC \quad = \quad \begin{cases} PC'.2 & : \quad full_1 = 1 \\ DPC.2 & : \quad full_1 = 0 \end{cases}$$

The PC environment before and after the transformation is depicted in figure 4.6.

However, this does not disprove the correctness of the implementation in [MP00]: in the pipelined implementation in [MP00], the value of DPC is always taken from the register PC'. This is correct since the stall engine in [MP00] ensures that the stages 0 and 1 are always clocked simultaneously.

This implies that the full bit $full_1$ is always active iff stage 0 reads *DPC* except for the first time after reset. The calculation of the first *DPC* is compensated using the signal *reset*.

4.2.3 Result Forwarding

In the general case, i.e., if $w > k+1$, g_kR is still undefined. The method used for the case $w = k+1$ cannot be used with reasonable effort since this would require combining the transition functions of two or even more stages. Besides the extra hardware cost, these combined transition functions would be too deep and would lengthen the cycle time.

However, forwarding over multiple stages is reasonable in one special case: Microprocessor instruction sets usually offer different kinds of instructions, such as ALU and memory instructions. The value that is to be forwarded is the result of these operations. The different instructions are processed by different stages, e.g., by an execute stage and by a memory stage as described in the previous chapter. The result is available in an early stage therefore. The later stages just pass this result unmodified. The transition functions that are left to be applied are very simple in this case: they are just the identity.

In a sequential machine, it is possible to write the result in the register file as soon as it is calculated. However, as shown by the example in figure 4.7, it is not possible to do so in the pipelined machine. Consider two instructions I_1 and I_2:

$$I_1: \quad R1 := DM[0]$$

$$I_2: \quad R1 := R2 + R3$$

If each instruction writes its result into the register file as soon as it is available, the register R_1 would contain the result of the memory instruction I_1 instead of the ALU instruction I_2. The result is written in the last stage (WB) therefore and not as soon as available. This is shown in figure 4.8.

In a pipelined implementation, the result therefore must be buffered in an implementation register. The value of this implementation register is written into the register file in the last stage.

In order to realize forwarding of results that are available in an early

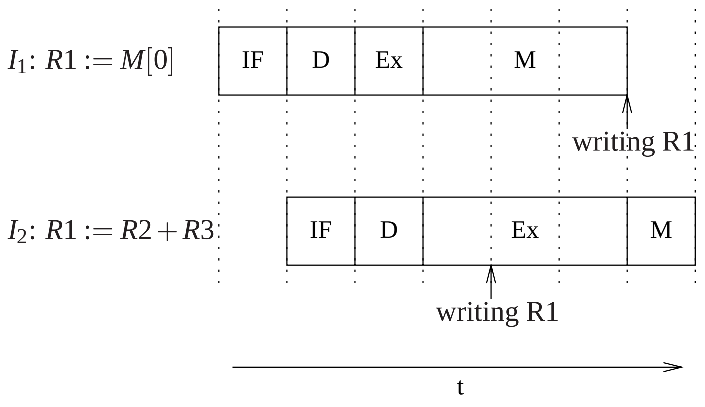


Figure 4.7 Write back as soon as possible.

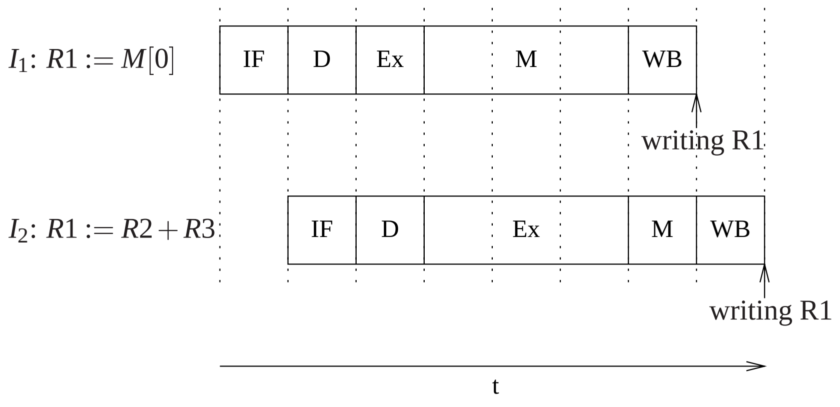


Figure 4.8 Write back in program order.

Stage	Alias
2 (EX)	$C.3 =^a GPR$
3 (M)	$C.4 =^a GPR$

Table 4.3 Write aliases for the DLX

stage, it is necessary to specify which implementation register holds the intermediate result of a specification register.

Definition 4.1
Write Alias

Let R denote a specification register and Q with $Q \in out(k)$ denote an implementation output register of stage k. In this case, let $Q =^a R$ denote that Q is used in order to buffer the final value of R. The register Q is called a *write alias* for R.

The list of write aliases is added to the hardware description file.

Example: Result Forwarding in the DLX Consider the prepared sequential DLX as defined in chapter 3. Consider an ALU instruction, e.g., addi. The final result written into GPR, i.e., the sum, is known already in stage 2. The result of the ALU instruction is written into the register $C.3$ (figure 3.5, page 54). Thus, one can define $C.3 =^a GPR$.

Table 4.3 shows the list of all write aliases defined for the DLX design.

Definition 4.2
Valid Values

As soon as a value is written into a write alias register Q with $Q =^a R$, it is assumed that this value matches the final value of R in the configuration that is being calculated. Such a value is called *valid* value.

A register $Q.(l+1)$ is written iff the write enable signal of the register is active. The write enable signal of $Q.(l+1)$ is $f_l Qwe$. Thus, a value in a register $Q.(k+1)$ is valid iff any write enable signal $f_l Qwe$ of the alias register Q of stage $l \leq k$ is active.

Definition 4.3
Valid Signals

The hardware transformation program uses the list of write aliases in order to generate a set of additional signals $Q_k valid(c_I)$ for each $Q =^a R$.

In the following, it is assumed that precomputed versions of all write enable signals $f_l Qwe$ exist (see section 3.3, page 52). These precomputed values are stored in implementation registers. Remember that these registers are named just like the signal. We will use these registers in order to calculate the valid signals in an obvious way:

$$Q_k valid : C_I \longrightarrow \mathsf{B}$$

$$Q_k valid(c_I) \quad := \quad \bigvee_{l=stage(Q)}^{k} c_I . f_l Qwe.k$$

This definition is slightly different from the definition of the valid signals in [MP00]. The valid signals from [MP00] are on even if the instruction does not write *GPR*, e.g., in case of a store instruction, which writes to the data memory only. However, this does not disprove the correctness of the hardware in [MP00], since special care is taken for instructions that do not write *GPR*.

However, there is no guarantee that a value written into an alias register matches the value finally written into the register that is to be forwarded. Thus, this assumption must be proved for each $Q =^a R$.

This is formalized as follows: Let c_S be a configuration of the specification machine. In analogy to the valid signals above, we define a *correct valid signal*. The predicate $Q_k Valid(c_S)$ holds iff a correct write enable signals $f_l Qwe$ of the alias register Q of stage $l \leq k$ holds.

As described in section 3.3, the registers holding the precomputed versions of control signals are treated just like implementation registers. Thus, the definition of a correct value of a register, as given in section 3.5.4 (page 71), also applies for these registers. Thus, one can use the correct value of the registers holding precomputed signals in order to define the correct value of the valid signals.

The register holding the precomputed version of a signal is named just like the signal, i.e., if the name of the write enable signal is $f_l Qwe$, so is the name of the register. The correct value of a register $R.k$ is $\Omega_{k-1} R$. Thus, the predicate $Q_k Valid$ is defined as follows:

$$Q_k Valid : C_S \longrightarrow \mathsf{B}$$

$$Q_k Valid(c_S) \quad := \quad \bigvee_{l=stage(Q)}^{k} \Omega_{k-1} f_l Qwe(c_S)$$

Example: Result Forwarding in the DLX Consider the prepared sequential DLX as defined in chapter 3. Let

$$I_0,\, I_1,\, I_2,\, \ldots$$

denote an instruction sequence, as in section 3.5.2 (page 63). As described above, if I_i is an ALU instruction, the final result written into GPR is known already in stage 2. The result of the ALU instruction is written into the register $C.3$. In this case, $C_2Valid(c_S^i)$ holds because the write enable signal f_2Cwe is active.

Let I_i be a load instruction. In this case, $C_2Valid(c_S^i)$ does not hold because neither f_2Cwe nor f_1Cwe are active.

◄ Lemma 4.8

Let $w = stage(R)$ hold. The following statement is shown for each $Q =^a R$: if the valid predicate of a register Q holds, the correct value written in this implementation register has to match the final value generated by the register transition function of R. The correct value written into $Q.(k+1)$ is given by $\Omega_k Q(c_S^i)$ (section 3.5.4 page 71).

If the write access to R does not have an address:

$$Q_kValid(c_S^i) \quad\Longrightarrow\quad \Omega_k Q(c_S^i) = c_S^{i+1}.R$$

If the write access to R has an address, we assume that the control signals for the write address f_wRwa are also precomputed. The correct value of this address is just the correct value of implementation register holding the precomputed write address.

$$Q_kValid(c_S^i) \quad\Longrightarrow\quad \Omega_k Q(c_S^i) = c_S^{i+1}.R[\Omega_{k-1}\, f_wRwa(c_S^i)]$$

This is illustrated in figure 4.9 for stage $k = 2$ (EX). In this stage, the result, which is to be forwarded, is provided by the ALU. This is $\Omega_2 C$. This result is calculated using the values in the registers in $out(1)$ as inputs. Thus, the address is also taken from a register in $out(1)$, which is $f_4GPRwa.2$. The correct value of $f_4GPRwa.2$ is given by $\Omega_1\, f_4GPRwa$.

The same method is used in [MP00] with a different notation: the result provided by the ALU is denoted by $C'.2$. The address is taken from $f_4GPRwa.2$, which is a precomputed version of the write address used for writing GPR. In correspondence to lemma 4.8, [MP00] provides a lemma in order to argue that the value of $C'.2$ matches the final value assuming the valid signal is active.

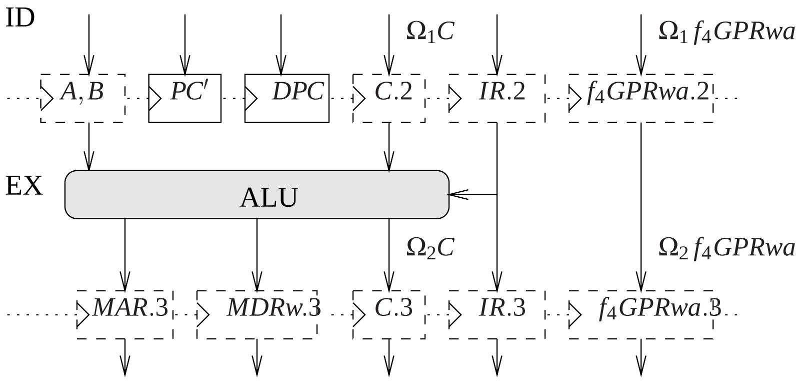


Figure 4.9 ALU results and the register address in the machine without implementation registers

PROOF Lemma 4.8 is shown for the write aliases for the pipelined DLX as defined in table 4.3.

For $C.3 =^a GPR$, the claim is:

$$C_2 Valid(c_S^i) \implies \Omega_2 C(c_S^i) = c_S^{i+1}.GPR[\Omega_1 f_4 GPRwa(c_S^i)] \quad (4.4)$$

The first step is to conclude that the write enable signal $f_4 GPRwe$ is active using that $C_2 Valid(c_S^i)$ holds, i.e., one shows that an instruction in stage EX that is valid actually writes a GPR. Remember that the write enable signal of $GPR.4$ is precomputed in the decode stage (section 3.3, page 52), i.e., $f_4 GPRwe$ just takes the value from the register $f_4 GPRwe.4$. The correct value of this register is $\Omega_3 f_4 GPRwe(c_S^i)$:

$$f_4 GPRwe(\Omega_3 f_4 GPRwe(c_S^i)) \quad = \quad \Omega_3 f_4 GPRwe(c_S^i)$$

By repeatedly expanding the definition the function on the right-hand side, one gets:

$$\begin{aligned} f_4 GPRwe(\Omega_3 f_4 GPRwe(c_S^i)) &= \Omega_3 f_4 GPRwe(c_S^i) \\ &= \Omega_2 f_4 GPRwe(c_S^i) \\ &= \Omega_1 f_4 GPRwe(c_S^i) \\ &= f_1 f_4 GPRwe(\Omega_0 IR(c_S^i)) \end{aligned}$$

One proves this by expanding the definition of the correct valid signal:

$$\begin{aligned} C_2 Valid(c_S^i) &= \Omega_1 f_1 Cwe(c_S^i) \vee \Omega_1 f_2 Cwe(c_S^i) \\ &= f_1 f_1 Cwe(\Omega_0 IR(c_S^i)) \vee f_1 f_2 Cwe(\Omega_0 IR(c_S^i)) \end{aligned}$$

As defined in section 3.4.3 (page 55), $f_1 f_1 Cwe(\Omega_0 IR(c_S^i))$ holds if the instruction I_i is a jump and link instruction. The write enable function $f_1 f_2 Cwe(\Omega_0 IR(c_S^i))$ holds if I_i is an ALU/shift instruction (section 3.4.4). This allows concluding that $f_1 f_4 GPRwe(\Omega_0 IR(c_S^i))$ holds (section 3.4.6, page 60). Thus, $\Omega_3 f_4 GPRwe(c_S^i)$ holds.

This allows concluding that the write enable signal with correct input $f_4 GPRwe(\Omega_3 f_4 GPRwe(c_S^i))$ is active, which is equivalent to $f\Gamma_4 GPRwe$. We conclude the claim using lemma 3.15: Since the write enable signal $f\Gamma_4 GPRwe(c_S^i)$ is active, lemma 3.15 (page 78, the generic claim is in lemma 3.13 page 75) states for all addresses x:

$$c_S^{i+1}.GPR[x] \quad = \quad \begin{cases} f\Gamma_4 GPR(c_S^i) & : \quad x = f\Gamma_4 GPRwa(c_S^i) \\ c_S^i.GPR[x] & : \quad \text{otherwise} \end{cases} \tag{4.5}$$

By expanding definitions, one shows that $f\Gamma_4 GPRwa(c_S^i)$ is equal to the precomputed version, which is $\Omega_1 f_4 GPRwa(c_S^i)$. Using that equality, equation 4.5 with $x = \Omega_1 f_4 GPRwa(c_S^i)$ is:

$$c_S^{i+1}.GPR[\Omega_1 f_4 GPRwa(c_S^i)] \quad = \quad f\Gamma_4 GPR(c_S^i) \tag{4.6}$$

By inserting equation 4.6 into the claim (equation 4.4), the claim is transformed into:

$$\Omega_2 C(c_S^i) \quad \overset{!}{=} \quad f\Gamma_4 GPR(c_S^i) \tag{4.7}$$

This new claim is shown as follows: the first step is to show that the instruction is not a load instruction, as indicated by $I\_load(\Omega_1 IR(c_S^i))$. This is done using that $C_2 Valid(c_S^i)$ holds. One shows that the instruction coded by $\Omega_1 IR(c_S^i)$ is either a jump and link or ALU/Shift instruction. Thus, it cannot be a load instruction. One easily shows that $\Omega_1 IR$ is equal to $\Omega_3 IR$, thus, $I\_load(\Omega_3 IR(c_S^i))$ is also not active.

The proof proceeds by expanding the definition of $\Gamma_4 GPR$ on the right-hand side of the claim (equation 4.7):

$$\begin{aligned} \Omega_2 C(c_S^i) \quad &\overset{!}{=} \quad f_4 GPR(\Gamma_4 GPR(c_S^i)) \\ &= \quad f_4 GPR(\Omega_3 C(c_S^i), \Omega_3 IR(c_S^i), \\ &\qquad \Omega_3 MAR(c_S^i), \Omega_3 MDRr(c_S^i)) \end{aligned} \tag{4.8}$$

One then expands the definition of $f_4 GPR$ (section 3.4.6). Since the instruction is not a load instruction ($I\_load(\Omega_3 IR(c_S^i))$ does not hold), the function f_4GPR returns the value of the C register. This transforms the claim into:

$$\Omega_2 C(c_S^i) \overset{!}{=} \Omega_3 C(c_S^i) \tag{4.9}$$

This is shown by expanding the definition of $\Omega_3 C(c_S^i)$ on the right-hand side.

Very similar arguments are used in order to show the claim for $C.4 =^a GPR$. QED

Implementing Result Forwarding Thus, if such a $Q_k Valid$ predicate holds, it is possible to take the result written into the $Q.(k+1)$ register as the value for a read access. This is done only if the instruction in a given stage actually writes the desired register. A signal is defined that is active if this holds. This signal is called *hit signal*.

Definition 4.4 ► *Hit Signals* Let stage k depend on a specification register R that is an output register of stage w with $w \geq (k+1)$. In addition to the valid signals, a set of hit signals is defined as follows: if the write access to R does not have an address:

$$\begin{aligned} &\forall j \in \{k+1,\dots,w\}: \\ &\quad R_k hit[j](c_I) \;:=\; full_j(c_I) \wedge f_w Rwe.j \end{aligned}$$

We use the precomputed version of the write enable signal of R in order to determine if the instruction in the given stage writes R. If the write access has an address, it is necessary to check the address of the write access in addition to the conditions above. As above, we use the precomputed version of the write address.

$$\begin{aligned} &\forall j \in \{k+1,\dots,w-1\}: \\ &\quad R_k hit[j](c_I) \;:=\; full_j(c_I) \wedge f_w Rwe.j \wedge \\ &\qquad\qquad\qquad\quad (f\gamma_k Rra(c_I) = f_w Rwa.j) \end{aligned}$$

In case of stage w, the address and write enable signals are taken from the write access directly:

$$\begin{aligned} R_k hit[w](c_I) \;:=\; & full_w(c_I) \wedge f\gamma_w Rwe(c_I) \wedge \\ & (f\gamma_k Rra(c_I) = f\gamma_w Rwa(c_I)) \end{aligned}$$

A very similar definition is in [MP00]. If any hit signal of a stage j is active, let top denote the smallest such j, i.e., the topmost stage with active hit signal:

$$top \quad := \quad \min\{j \in \{k+1,\ldots,w\} \mid R_k hit[j](c_I)\}$$

This is undefined if no signal $R_k hit[j]$ is active. The signal

$$R_k hit[top]$$

is called *topmost hit signal*.

Using this definition, one can now define the forwarding function $g_k R$. For sake of simplicity, let us assume that the read enable signal $f\gamma_k Rre(c_I)$ is active. If not, $g_k R$ returns zero and no forwarding is necessary.

If the topmost hit signal is in stage w, i.e., the stage that actually outputs R, one just takes the value written into R, which is provided by $f_w R$. If the topmost hit signal is in a stage $j < w$, one takes the value written into the alias register, i.e., $\omega_j Q(c_I)$. If no hit signal is active, one takes the value from R.

If the write access to R does not have an address, $g_k R$ is:

$$g_k R(c) \quad = \quad \begin{cases} f\gamma_w R(c) & : \quad R_k hit[w](c) \wedge w = top \\ \omega_j Q(c) & : \quad j \in \{k+1,\ldots,w-1\} \wedge \\ & \quad\;\; R_k hit[j](c) \wedge j = top \\ R.(w+1) & : \quad \text{otherwise} \end{cases} \tag{4.10}$$

If the read access has an address, $g_k R$ is defined using the read address. Let $x := f\gamma_k Rra(c)$ be the address.

$$g_k R(c) \quad = \quad \begin{cases} f\gamma_w R(c) & : \quad R_k hit[w](c) \wedge w = top \\ \omega_j Q(c) & : \quad j \in \{k+1,\ldots,w-1\} \wedge \\ & \quad\;\; R_k hit[j](c) \wedge j = top \\ R.(w+1)[x] & : \quad \text{otherwise} \end{cases} \tag{4.11}$$

The same forwarding method is used in [MP00]. As in [MP00], the comparison $j = top$ is realized using a chain of multiplexers (in PVS, IF ... THEN ... ELSIF ... ENDIF is used). This is depicted exemplary in figure 4.10. [1]

[1] In larger pipelines, the delay of this circuit grows linear with the pipeline size. For large pipelines, a find first one circuit is faster.

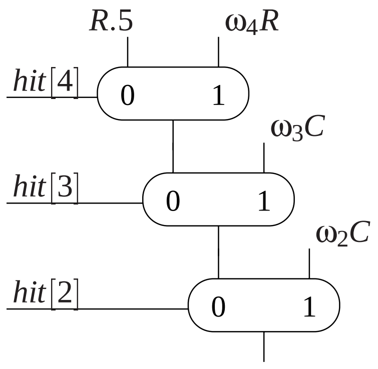


Figure 4.10 Implementation of 3-stage forwarding

Observe that the forwarding from the next stage, as described in section 4.2.2, is just a special instantiation of the more general forwarding method described in this section.

In case of a hit in a stage, lemma 4.8 will be used in order to argue about the value read from the given stage. However, if the hit signal is not active, one has to argue that one can safely ignore the contents of the stage in order to do forwarding. This is asserted by the following lemma. In terms of microprocessors, the lemma asserts that the instruction in a given stage does not update the register that is to be forwarded if the hit signal is off.

Lemma 4.9 ▶ Let $Q =^a R$ hold and let $Q \in out(j)$ and $R \in out(w)$ hold. Consider the correct value of the precomputed write enable signal of R.

If the write access to R does not have an address, the register R is not modified if the write enable signal is not active:

$$\overline{\Omega_{j-1} f_w Rwe} \quad \Longrightarrow \quad c_S^i.R = c_S^{i+1}.R$$

Observe that $\Omega_{j-1} f_w Rwe$ is just the correct version of the hit signal (definition 4.4). Thus, it is called *correct hit signal* (the full signal is ommited because the configuration c_S does not have full bits; it processes one instruction in one cycle).

If the write access to R has an address, in analogy to definition 4.4, the address x is compared with the correct precomputed address, as defined by $\Omega_{j-1} f_w Rwa(c_S)$.

Thus, in terms of microprocessors, if the addresses are not equal or if the write enable function returns false, the instruction in stage j does not update the desired register.

$$\begin{array}{rl} & \forall x \in \mathtt{W}_a(R):\ x \neq \Omega_{j-1} f_w Rwa(c_S^i) \vee \overline{\Omega_{j-1} f_w Rwe} \\ \Longrightarrow & c_S^i.R[x] = c_S^{i+1}.R[x] \end{array}$$

The lemmas 4.8 and 4.9 are called *write alias correctness lemmas*.

PROOF

Lemma 4.2.3 is shown for the write aliases for the pipelined DLX as defined in table 4.3.

For $C.3 =^a GPR$, the claim is:

$$\begin{array}{rl} & \forall x \in \mathtt{W}_a(GPR):\ x \neq \Omega_1 f_4 GPRwa(c_S^i) \vee \overline{\Omega_1 f_4 GPRwe(c_S^i)} \\ \overset{!}{\Longrightarrow} & c_S^i.GPR[x] = c_S^{i+1}.GPR[x] \end{array}$$

There are two cases regarding the value of $f\Gamma_4 GPRwe(c_S^i)$. If it is off, the claim directly follows from lemma 3.15 (page 78).

Thus, let $f\Gamma_4 GPRwe(c_S^i)$ hold. From this, we easily show by expanding definitions that $\Omega_1 f_4 GPRwe(c_S^i)$ holds:

$$\begin{array}{rcl} f\Gamma_4 GPRwe(c_S^i) & = & f_4 GPRwe(\Gamma_4 GPRwe(c_S^i)) \\ & = & f_4 GPRwe(\Omega_3 f_4 GPRwe(c_S^i)) \\ & = & \Omega_3 f_4 GPRwe(c_S^i) \\ & = & \Omega_2 f_4 GPRwe(c_S^i) \\ & = & \Omega_1 f_4 GPRwe(c_S^i) \end{array}$$

This transforms the claim into:

$$\begin{array}{rl} & \forall x \in \mathtt{W}_a(GPR):\ (x \neq \Omega_1 f_4 GPRwa(c_S^i)) \\ \overset{!}{\Longrightarrow} & c_S^i.GPR[x] = c_S^{i+1}.GPR[x] \end{array}$$

Let *ind* be a shorthand for $f\Gamma_4 GPRwa(c_S^i)$. Since the write enable signal is active, lemma 3.15 states for all addresses x:

$$c_S^{i+1}.GPR[x] \quad = \quad \begin{cases} f_4 GPR(\Gamma_4 GPR(c_S^i)) & : \ x = ind \\ c_S^i.GPR[x] & : \ \text{otherwise} \end{cases} \qquad (4.12)$$

By inserting this into the claim, the claim is transformed into:

$$\begin{aligned} & \forall x \in \mathtt{W}_a(GPR) : \ (x \neq \Omega_1 f_4GPRwa(c_S^i)) \\ \overset{!}{\Longrightarrow} \ & c_S^i.GPR[x] = \begin{cases} f_4GPR(\Gamma_4GPR(c_S^i)) & : \ x = ind \\ c_S^i.GPR[x] & : \ \text{otherwise} \end{cases} \end{aligned}$$

We will now conclude $x \neq ind$ using $x \neq \Omega_1 f_4GPRwa(c_S^i)$:

$$\begin{aligned} ind &= f\Gamma_4GPRwa(c_S^i) \\ &= f_4GPRwa(\Gamma_4GPRwa(c_S^i)) \\ &= f_4GPRwa(\Omega_3 f_4GPRwa(c_S^i)) \\ &= \Omega_3 f_4GPRwa(c_S^i) \\ &= \Omega_2 f_4GPRwa(c_S^i) \\ &= \Omega_1 f_4GPRwa(c_S^i) \\ &\neq x \end{aligned}$$

This concludes the claim. Very similar arguments are used in order to
QED show the claim for $C.4 =^a GPR$.

Consider a hit in stage top and that the valid signal of that stage, as given by definition 4.3, does not hold. In this case, one cannot use lemma 4.8 to argue that the values in the stage are valid. Lemma 4.9 cannot be used, either, in order to argue that the stage can be ignored, since there is a hit in the stage. In this case, forwarding, as described above, fails completely.

4.3 Stalling

If forwarding fails, the calculation of the input values of the stage is not possible. It is necessary to delay the calculation in a stage until the data is available. Since the result of prior stages has to be stored somewhere, these stages have to wait also. In contrast to that, later stages must not be stalled since these stages calculate the desired inputs. The mechanism used to realize this is called *stall engine* and is introduced in [MP00]. In contrast to the stall engine in [MP00], the stall engine presented here supports independent stall signals for each stage.

The stall engine is implemented by re-defining the signals $stall_k$. In the prepared sequential machine the stall signals are only used in order to obey external stall conditions such as caused by slow memory. In the pipelined machine, *internal stall conditions* are added.

In order to calculate the signals $stall_k$, a signal is required that indicates whether a given stage has to wait for an input value. The signal $dhaz_k$ (data hazard) is active iff stage k is waiting for an input operand. The stage k must be stalled if $dhaz_k$ is active and if the stage is full. This is called a *data hazard stall*.

Furthermore, the stage must be stalled if the next stage (stage $k+1$) is stalled because the necessary data paths and registers are not available in this case. This case is called a *structural hazard stall*.

Let ext_k denote the disjunction of the external stall signals of stage k, e.g., used for memory. Let int_k denote the disjunction of the internal stall signals. Since the last stage (stage $n-1$) has no next stage, the definition of the signal int_k depends on the stage number:

$$\begin{aligned} k \neq n-1: \quad int_k \quad &:= \quad dhaz_k \vee stall_{k+1} \\ int_{n-1} \quad &:= \quad dhaz_{n-1} \end{aligned} \tag{4.13}$$

This allows defining the stall signal $stall_k$:

$$stall_k \quad := \quad full_k \wedge (ext_k \vee int_k) \tag{4.14}$$

This definition of the stall signal obviously conforms to the stall signal convention 4.2 (page 95)[2].

As described above, forwarding fails if there is a hit in stage top and the valid signal of the stage does not hold. For each input that requires result forwarding, a separate data hazard signal is defined. Let $R \in out(w)$ be a specification register that is read by stage k. The data hazard signal for this input is then called $R_k dhaz$. The data hazard signal of stage k, which is $dhaz_k$, is the disjunction of these data hazard signals.

In case of the DLX, we have two read accesses to the general purpose register file. In analogy to the naming convention described in section 3.2.7 (page 51), *GPRa* and *GPRb* are used for the GPR operands:

$$dhaz_1 \quad := \quad GPRa_1 dhaz \vee GPRb_1 dhaz$$

[2]In the PVS tree, this is shown in form of a TCC (type-correctness condition).

The data hazard signals $R_k dhaz$ are defined as follows: if there is no hit signal active, no data hazard is indicated. If there is a hit in any stage, the stage given by top is examined. As in [MP00], a data hazard is indicated if the stage is not valid. In case of stage $w = stage(R)$, there is no valid signal. In stage w, the result is written into the register and the result is therefore known in stage w at the latest. Thus, there is no need for a valid signal in stage w.

In addition to that, a data hazard is also indicated if the data hazard signal of the stage top is active:

$$R_k dhaz(c_I) \quad := \quad \begin{cases} dhaz_w(c_I) & : \quad R_k hit[w](c_I) \wedge w = top \\ dhaz_j(c_I) \vee & : \quad j \in \{k+1, \ldots, w-1\} \wedge \\ \overline{Q_j valid(c_I)} & \quad\;\; R_k hit[j](c_I) \wedge j = top \\ 0 & : \quad \text{otherwise} \end{cases}$$

This addition to [MP00] is motivated as follows: If the valid signal of the stage is active, one uses lemma 4.8 in order to show that the output of the stage matches the value finally written in the register that is read. However, the output value of the stage is only correct if the inputs of the stage are correct. Assume the stage uses forwarding in order to get one or more inputs. These inputs are only correct if the forwarding does not fail.

It will turn out that forwarding fails iff the data hazard signal is active. Thus, the data hazard signal is checked.

This does not disprove the correctness of the implementation of the forwarding logic in [MP00]: in [MP00], data is forwarded from stages 2 to 4. For the calculation of GPR results, these stages never use forwarding in order to get inputs.

This also applies to the pipelined DLX presented in this chapter: stages 2 to 4 never use result forwarding, the data hazard signals of stages 2 to 4 are always false therefore.

In hardware, the comparison of the stage number with top is done with multiplexers as described in section 4.2.3.

4.4 Implementing the DLX$_\pi$

The implementation of the DLX$_\pi$ is completely identical to the implementation of the machine DLX$_\sigma$ as described in the previous chapter except for the following changes:

1. The definition of $full_0$ and the stall signals are changed as described in sections 4.1.1 and 4.3.
2. Forwarding logic is added for the stages IF and ID as described in section 4.2.

The complete process of introducing the new stalling and forwarding logic is completely automated.

4.5 Data Consistency

The following lemmas assert the correctness of the result forwarding mechanism as presented in the previous sections. These lemmas correspond to lemma 3.16 (page 82) of the sequential machine. They assert that the inputs generated during cycle $T-1$ are correct. They will be used in order to show that the values of the registers during cycle T are correct.

◄ Lemma 4.10

Let $sI(k,T)=i$ and $full_k^T$ hold. Let $R \in out(w)$ be a specification register with $w>k$ and let the stage correctness predicates P_j hold in all cycles up to cycle T. If there is no hit signal active, register R is not modified from configuration $c_S^{sI(w,T)}$ to configuration c_S^i:

$$R_S^{sI(w,T)} \quad = \quad R_S^i$$

If the read access has an address, the claim is that the register with the given address is not modified. Let x denote the address.

$$\begin{array}{rcl} x & := & f\Gamma_k Rra(c_S^i) \\ R_S^{sI(w,T)}[x] & = & R_S^i[x] \end{array}$$

PROOF

The first step is to assert the correctness of an address value, if present:

$$f\Gamma_k Rra(c_S^i) \quad \stackrel{!}{=} \quad f\gamma_k Rra(c_I^T)$$

Thus, one has to show:

$$\Gamma_k Rra(c_S^i) \quad \stackrel{!}{=} \quad \gamma_k Rra(c_I^T)$$

This is done as in the proof of lemma 3.16 (page 82). By definition, the inputs required in order to calculate the address do not require forwarding.

The claim is then shown easily by a full case split on the full bits of the stages $k+1$ to w. As soon as fixed values for the full bits are given, the scheduling invariants can be used in order to determine the value of $sI(w,T)$ relative to $sI(k,T)$ ($T > 0$ is shown easily using lemma 4.6).

For example, if all full bits are off, one easily shows that $sI(w,T) = sI(k,T) = i$ holds. In this case, the claim above obviously holds.

If the full bit of one or more stages is set, let

$$diff \quad := \quad sI(k,T) - sI(w,T)$$

denote the difference between the values of the scheduling function. Using the scheduling invariants 3.2 and 3.3, one easily shows that there are as many active full bits as given by *diff*.

For each active full bit $full_l^T$, one argues that

$$R_S^{sI(l,T)} \quad = \quad R_S^{sI(l,T)+1}$$

holds. If the read access has an address, it is argued that

$$R_S^{sI(l,T)}[x] \quad = \quad R_S^{sI(l,T)+1}[x]$$

holds. This is done using the fact that the hit signal $R_k hit[l]$ is off and by lemma 4.9 if $l \neq w$ and by lemma 3.13 if $l = w$.

QED This can be done *diff*-times and concludes the claim.

Lemma 4.11 ▶ Let $sI(k,T) = i$ and $full_k^T$ hold. Let $R \in out(w)$ be a specification register with $w > k$ and let the stage correctness predicates P_j hold in all cycles up to cycle T. If there is an active hit signal, register R is not modified from configuration $c_S^{sI(top,T)+1}$ to configuration c_S^i:

$$R_S^{sI(top,T)+1} \quad = \quad R_S^i$$

If the read access has an address:

$$\begin{aligned} x \quad &:= \quad f\Gamma_k Rra(c_S^i) \\ R_S^{sI(top,T)+1}[x] \quad &= \quad R_S^i[x] \end{aligned}$$

It is not surprising that one argues about $R_S^{sI(top,T)+\mathbf{1}}$. In case of an active hit signal, the forwarding hardware takes the *output* of the stage top. If instruction $I_{sI(top,T)}$ is in stage top, the outputs of the stage are part of configuration $c_S^{sI(top,T)+1}$. Let $j = sI(top,T)$ hold:

$$c_S^j \xrightarrow{I_j} c_S^{j+1}$$

PROOF

This is shown with the same method as used in the proof of lemma 4.10.

For example, if $top = k+1$, i.e., the hit is in the next stage, one easily shows that $sI(top,T)+1 = i$ holds using invariant 3.3 and that $full_{top}^T$ is active.

Let top be $k+2$ and $full_{k+1}^T$ not hold. In this case, one uses invariants 3.2 and 3.3 in order to show that $sI(top,T)+1 = i$. If a full bit is active, one uses lemma 4.9, as above.

QED

◀ Lemma 4.12

Let $sI(k,T) = i$ and $full_k^T$ hold and let the data hazard signal $dhaz_k^T$ be not active. Let $R \in out(w)$ be a specification register with $w > k$ and let the stage correctness predicates P_j hold in all cycles up to cycle T. Let there be no hit signal active.

The claim is that the inputs generated by the function g_kR during cycle T are correct:

$$g_kR(c_I^T) \quad = \quad G_kR(c_S^i)$$

PROOF

Since R is a specification register, the correct value on the right-hand side of the claim is given in the configuration of the specification machine. If the read access does not have an address, this transforms the claim into:

$$g_kR(c_I^T) \quad \stackrel{!}{=} \quad \begin{cases} R_S^i & : \quad f\Gamma_kRre(c_S^i) \\ 0 & : \quad \text{otherwise} \end{cases}$$

In case of a read access with address, the correct value is defined using the correct value of the address, as in lemma 3.16 (page 82):

$$\begin{aligned} x \quad &:= \quad f\Gamma_kRra(c_S^i) \\ g_kR(c_I^T) \quad &\stackrel{!}{=} \quad \begin{cases} R_S^i[x] & : \quad f\Gamma_kRre(c_S^i) \\ 0 & : \quad \text{otherwise} \end{cases} \end{aligned}$$

The first step is to assert the correctness of an address value, if present, as done in the proof of lemma 4.10:

$$f\Gamma_k Rra(c_S^i) \quad = \quad f\gamma_k Rra(c_I^T)$$

By the same arguments, one shows the correctness of the inputs of the read enable function $f_k Rre$. Let the read enable signal $f_k Rre$ be active. Otherwise, the claim is trivial since zero is returned and no forwarding is required. This transforms the claim into:

$$\begin{array}{rrcl} \text{no address:} & g_k R(c_I^T) & \overset{!}{=} & R_S^i \\ \text{with address:} & g_k R(c_I^T) & \overset{!}{=} & R_S^i[x] \end{array} \qquad (4.15)$$

Since no hit signal is active, by definition of $g_k R$ (equation 4.10), R_I^T is read:

$$g_k R(c_I^T) \quad = \quad R_I^T$$

If the read access has an address, the correct address is used (equation 4.11):

$$g_k R(c_I^T) \quad = \quad R_I^T[x]$$

Using the stage correctness predicate for cycle T and stage w, one easily transforms the right-hand side of both equations into:

$$\begin{array}{rrcl} \text{no address:} & g_k R(c_I^T) & = & R_S^{sI(w,T)} \\ \text{with address:} & g_k R(c_I^T) & = & R_S^{sI(w,T)}[x] \end{array} \qquad (4.16)$$

This allows transforming the claim into:

$$\begin{array}{rrcl} \text{no address:} & R_S^{sI(w,T)} & \overset{!}{=} & R_S^i \\ \text{with address:} & R_S^{sI(w,T)}[x] & \overset{!}{=} & R_S^i[x] \end{array} \qquad (4.17)$$

QED This is concluded using lemma 4.10.

Lemma 4.13 ► Let $sI(k,T) = i$ and $full_k^T$ hold and let the data hazard signal $dhaz_k^T$ be not active. Let $R \in out(w)$ be a specification register with $w > k$ and let the stage correctness predicates P_j hold in all cycles up to cycle T. Let there be an active hit signal.

The claim is that the inputs generated by the function g_kR during cycle T are correct:

$$g_kR(c_I^T) \quad = \quad G_kR(c_S^i)$$

PROOF

Since R is a specification register, the correct value on the right-hand side of the claim is given in the configuration of the specification machine. If the read access does not have an address, this transforms the claim into:

$$g_kR(c_I^T) \quad \overset{!}{=} \quad \begin{cases} R_S^i & : \quad f\Gamma_kRre(c_S^i) \\ 0 & : \quad \text{otherwise} \end{cases}$$

In case of a read access with address, the correct value is defined using the correct value of the address, as in lemma 3.16 (page 82):

$$\begin{aligned} x \quad &:= \quad f\Gamma_kRra(c_S^i) \\ g_kR(c_I^T) \quad &\overset{!}{=} \quad \begin{cases} R_S^i[x] & : \quad f\Gamma_kRre(c_S^i) \\ 0 & : \quad \text{otherwise} \end{cases} \end{aligned}$$

The claim is shown inductively beginning with the last stage and proceeding from stage $k+1$ to stage k. In case of the last stage, which is stage $n-1$, there is nothing to show since there is no stage below to forward from. Assuming the claim holds for stages k' with $k < k' < n$, the claim is shown for stage k as follows:

As in the proof of lemma 4.12, one asserts the correctness of the address value and that the read enable signal is active.

As required in the premise, the data hazard signal R_kdhaz^T is not active. By definition of the data hazard signal, this implies that the valid bit of the stage top is active and that the data hazard signal of stage top is not active.

As described above, one assumes the correctness of the inputs of the stages $k' > k$ in order to show the correctness of the inputs of stage k. Since $top > k$, one can apply the induction premise for stage top. This shows the correctness of the inputs of the stage top:

$$\gamma_{top}R(c_I^T) \quad = \quad \Gamma_{top}R(c_S^{sI(w,T)}) \tag{4.18}$$

The claim is now shown by a case split on the value of top (in PVS, a separate lemma is used for the possible values of top).

Let top $=$ w hold, i.e., the hit is in the stage that writes R. Since $top = w$, $g_k R$ returns the value written into $R.(w+1)$. If the write access does not have an address, this is (equation 4.10):

$$g_k R(c_I^T) \quad = \quad f\gamma_w R(c_I^T) \tag{4.19}$$

As described above, one uses that the inputs of stage top are correct. Formally, one uses equation 4.18, which transforms the last equation into:

$$g_k R(c_I^T) \quad = \quad f\Gamma_w R(c_S^{sI(w,T)}) \tag{4.20}$$

Using this equation, the claim is transformed into:

$$\begin{array}{rccc} \text{no address:} & f\Gamma_w R(c_S^{sI(w,T)}) & \overset{!}{=} & R_S^i \\ \text{with address:} & f\Gamma_w R(c_S^{sI(w,T)}) & \overset{!}{=} & R_S^i[x] \end{array} \tag{4.21}$$

One easily shows that $f\Gamma_w Rwe(c_S^{sI(w,T)}))$ holds by using that the hit signal $R_k hit[w]$ is active (definition 4.4). If the read access does not have an address, lemma 3.13 states:

$$R_S^{sI(w,T)+1} \quad = \quad f\Gamma_w R(c_S^{sI(w,T)}) \tag{4.22}$$

This allows transforming the claim into:

$$R_S^{sI(w,T)+1} \quad \overset{!}{=} \quad R_S^i \tag{4.23}$$

This is concluded by lemma 4.11.

In case of a read access with address, the last thing is to show that the address given by x matches the address actually used for the final write access to R, as given by lemma 3.13:

$$\begin{array}{rcl} f\Gamma_w Rwa(c_S^{sI(w,T)}) & \overset{!}{=} & x \\ & = & f\gamma_k Rra(c_I^T) \end{array}$$

The value on the right-hand side is equal to $f\gamma_w Rwa(c_I^T)$ because the signal $R_k hit[w]$ is active (definition 4.4, page 114). This transforms the claim into:

$$f\Gamma_w Rwa(c_S^{sI(w,T)}) \quad \overset{!}{=} \quad f\gamma_w Rwa(c_I^T)$$

Thus, it is sufficient to assert that the inputs of $f_w Rwa$ are correct:

$$\Gamma_w Rwa(c_S^{sI(w,T)}) \overset{!}{=} \gamma_w Rwa(c_I^T)$$

This is done as described above for the inputs of $f_w R$.

Let top $\neq$ w hold, i.e., the hit is not in the stage that writes R. Since $top \neq w$, there must be a write alias for R for the stage. Let the register Q be the alias register (i.e., $Q =^a R$).

In this case, $g_k R$ returns the value written into $Q.(top+1)$ (by definition of $g_k R$, equation 4.11):

$$g_k R(c_I^T) = \omega_{top} Q(c_I^T) \tag{4.24}$$

As above, one argues that the inputs of $f_{top} Q$ are correct. Thus, the output is correct.

$$\omega_{top} Q(c_I^T) = \Omega_{top} Q(c_S^{sI(top,T)}) \tag{4.25}$$

This allows transforming the claim into:

$$\begin{array}{rrcl} \text{no address:} & \Omega_{top} Q(c_S^{sI(top,T)}) & \overset{!}{=} & R_S^i \\ \text{with address:} & \Omega_{top} Q(c_S^{sI(top,T)}) & \overset{!}{=} & R_S^i[x] \end{array} \tag{4.26}$$

Using lemma 4.11, the claim is transformed into:

$$\begin{array}{rrcl} \text{no address:} & \Omega_{top} Q(c_S^{sI(top,T)}) & \overset{!}{=} & R_S^{sI(top,T)+1} \\ \text{with address:} & \Omega_{top} Q(c_S^{sI(top,T)}) & \overset{!}{=} & R_S^{sI(top,T)+1}[x] \end{array} \tag{4.27}$$

Since there is a hit in stage top, one concludes that the valid signal $Q_{top} valid(c_I^T)$ is active. Using this, one easily shows that the correct valid bit $Q_{top} Valid(c_S^{sI(top-1,T)-1})$ holds:

$$Q_{top} valid(c_I^T)) = \bigvee_{l=stage(Q)}^{top} c_I^T.f_l Qwe.top$$

One transforms the right hand sind by applying the stage correctness predicate for implementation registers, stage $top-1$ and cycle T:

$$\begin{aligned} Q_{top} valid(c_I^T)) &= \bigvee_{l=stage(Q)}^{top} \Omega_{top-1} f_l Qwe(c_S^{sI(top-1,T)-1}) \\ &= Q_{top} Valid(c_S^{sI(top-1,T)-1}) \end{aligned}$$

Since stage top is full, one can apply scheduling invariant 3.3 in order to conclude that

$$sI(top-1,T)-1 \quad = \quad sI(top,T)$$

holds. Thus, $Q_{top}Valid(c_S^{sI(top,T)})$ holds.

This allows using lemma 4.8 for stage top and configuration $sI(top,T)$. If the read access does not have an address, this concludes the claim.

In case of a read access with address, lemma 4.8 states:

$$\Omega_{top}Q(c_S^{sI(top,T)}) \quad = \quad R_S^{sI(top,T)+1}[\Omega_{top-1}\,f_wRwa(c_S^{sI(top,T)})] \tag{4.28}$$

This transforms the claim into:

$$R_S^{sI(top,T)+1}[x] \quad \overset{!}{=} \quad R_S^{sI(top,T)+1}[\Omega_{top-1}\,f_wRwa(c_S^{sI(top,T)})] \tag{4.29}$$

It is therefore left to show that the addresses match:

$$\begin{aligned}\Omega_{top-1}\,f_wRwa(c_S^{sI(top,T)}) \quad &\overset{!}{=} \quad x \\ &= \quad f\gamma_kRra(c_I^T)\end{aligned}$$

The value on the right-hand side is equal to $f_wRwa.top_I^T$ because the signal $R_k^T hit[top]$ is active (definition 4.4). This transforms the claim into:

$$\Omega_{top-1}\,f_wRwa(c_S^{sI(top,T)}) \quad \overset{!}{=} \quad f_wRwa.top^T \tag{4.30}$$

By using the stage correctness predicate for cycle T and stage $top-1$, the right-hand side is transformed into:

$$\Omega_{top-1}\,f_wRwa(c_S^{sI(top,T)}) \quad \overset{!}{=} \quad \Omega_{top-1}\,f_wRwa(c_S^{sI(top-1,T)-1})$$

As above, one can use invariant 3.3 in order to show that $sI(top,T)$ is equal to $sI(top-1,T)-1$. This concludes the claim. QED

The following lemma corresponds to lemma 3.17 (page 85) in the sequential machine:

Lemma 4.14 ► Let T' be greater than zero. Assuming all stage correctness predicates for the cycle $T'-1$, the predicate for stage k holds for cycle T'.

$$(\forall l : \mathsf{P}_l(T'-1)) \quad \Longrightarrow \quad \mathsf{P}_k(T')$$

PROOF The proof proceeds as the proof of lemma 3.17. However, for the case $i > 0$ and ue^{T-1}, one uses lemmas 4.12 and 4.13 for operands that require forwarding. This lemma requires that the data hazard signal is not active. This is shown easily by definition of the stall and data hazard signal and using that the update enable signal is active. QED

4.6 Liveness

4.6.1 Introduction

The liveness criterion of the pipelined machine is identical to the liveness criterion of the prepared sequential machine as presented in chapter 3:

Let c_S^i be any desired configuration of the specification machine. The implementation machine is said to be alive iff for all stages k there exists a time $T \in \mathbb{N}_0$ with $sI(k,T) = i$:

$$\exists T \in \mathbb{N}_0 : sI(k,T) = i$$

As in chapter 3, this is shown by arguing that the update enable signal is alive, as done in lemma 3.22. This lemma has the premise that all stall signals are finite true. In the prepared sequential machine, only external stall signals exist and this property was assumed. This is no longer true for the pipelined machine since internal stall conditions were added (section 4.3).

Thus, a proof that the stall signals are finite true has to be given for the pipelined machine. According to equation 4.14 (page 119), there are three possible reasons for an active stall signal, given that the stage is full:

1. one of the external stall signals is active,
2. the data hazard signal is active,
3. the stall signal of the next stage is active.

Consider the following proof strategy: Beginning with the last stage, which has no next stage, we will argue that the stall signals are finite true. The external stall signals are still assumed to have this property. Furthermore, one shows that the data hazard signal is finite true, which can be

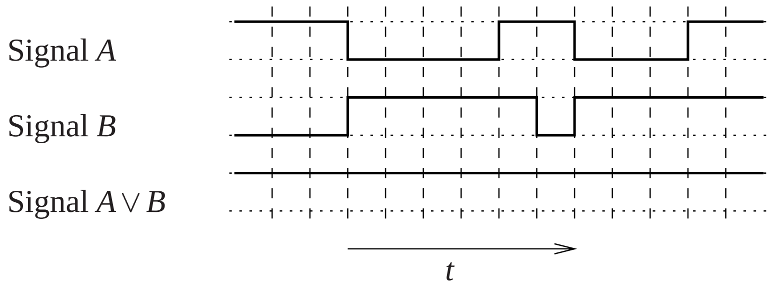


Figure 4.11 Two alternating, finite true signals A and B. The disjunction is not finite true but constant true.

done easily. It is now tempting to conclude that the disjunction of finite true signals is also finite true.

However, this is wrong. A finite true signal is guaranteed to eventually become false. The problem is that there is no guarantee that a signal that is finite true stays false for more than one cycle once it becomes false. In particular, one can think of two alternating signals that are both finite true (figure 4.11). The disjunction never becomes false and therefore cannot be finite true.

"Finite true" therefore is too weak. For the three signals above, one needs a stronger property such that one can conclude that the disjunction is finite true. In case of stall conditions, one needs that the signal actually stays false once it became false until all conditions are false. As soon as all conditions are false, the update enable signal becomes active and the stage therefore proceeds calculating.

4.6.2 Extended Liveness Calculus

The property of a signal that it "stays until" a given event (i.e., signal) is formalized as follows:

Definition 4.5 ► *Stays Until* Let *pred* and $pred_u$ be time predicates and T be a cycle. The predicate *pred* is said to *stay until* $pred_u$ from cycle T, iff the following holds: Given an arbitrary cycle $T' \geq T$ such that $pred_u$ does not hold for cycles T'' with $T \leq T'' < T'$, the predicate *pred* holds for all cycles T'' with $T \leq T'' \leq T'$.

$$stays\_until(pred, T, pred_u) \quad :\Longleftrightarrow$$

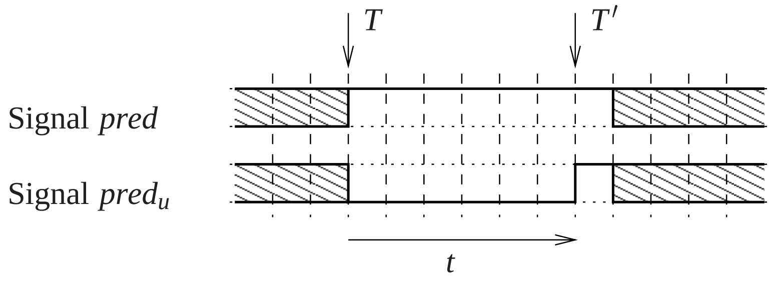


Figure 4.12 Two signals satisfying $stays\_until(pred, T, pred_u)$. A signal shown as a hatched box means that the value of the signal during this cycle does not matter.

$$\begin{array}{ll} \forall T' \mid T' \geq T: & (\forall T'' \mid T \leq T'' < T' : \overline{pred_u(T'')}) \\ \Longrightarrow & (\forall T'' \mid T \leq T'' \leq T' : pred(T'')) \end{array}$$

This is illustrated in figure 4.12. If a signal is shown as a hatched box this means that the value of the signal during this cycle does not matter.

Note that it is not required that signal $pred_u$ ever becomes true after cycle T. In particular, if $pred_u$ never becomes true after cycle T, $pred$ is required to hold for all cycles $T' \geq T$ (one easily shows this using induction).

◄ Lemma 4.15

Let $pred$ and $pred_u$ be time predicates and T be a cycle. Let T' be a cycle with $T' \geq T$. Let $pred$ stay until $pred_u$ after cycle T. If $pred_u$ is off during cycles T'' with $T \leq T'' < T'$, $pred$ also stays until $pred_u$ from cycle T':

$$\begin{array}{l} stays\_until(pred, T, pred_u) \wedge \\ \forall T'' \mid T \leq T'' < T' : \overline{pred_u(T'')} \\ \Longrightarrow stays\_until(pred, T', pred_u) \end{array} \tag{4.31}$$

PROOF

By definition 4.5, $stays\_until(pred, T', pred_u)$ is equivalent to:

$$\begin{array}{ll} \forall t' \mid t' \geq T': & (\forall t'' \mid T' \leq t'' < t' : \overline{pred_u(t'')}) \\ \Longrightarrow & (\forall t'' \mid T' \leq t'' \leq t' : pred(t'')) \end{array} \tag{4.32}$$

By definition of $stays\_until$, $stays\_until(pred, T, pred_u)$ is equivalent to:

$$\begin{array}{ll} \forall t' \mid t' \geq T: & (\forall t'' \mid T \leq t'' < t' : \overline{pred_u(t'')}) \\ \Longrightarrow & (\forall t'' \mid T \leq t'' \leq t' : pred(t'')) \end{array} \tag{4.33}$$

Since $T' \geq T$, one can instantiate formula 4.33 with t' from formula 4.32. This results in:

$$\begin{array}{ll} & (\forall t'' \mid T \leq t'' < t' : \overline{pred_u(t'')}) \\ \Longrightarrow & (\forall t'' \mid T \leq t'' \leq t' : pred(t'')) \end{array} \tag{4.34}$$

Obviously, the implication of equation 4.34 will conclude the claim as given by the implication of equation 4.32. However, it is left to show that the premise of the implication of equation 4.34 holds:

$$\forall t'' \mid T \leq t'' < t' : \overline{pred_u(t'')} \tag{4.35}$$

This is done as follows: if $t'' \geq T'$, one takes the premise in equation 4.32 in order to show $\overline{pred_u(t'')}$. If $t'' < T'$, $\overline{pred_u(t'')}$ holds according to the premise in equation 4.31.

QED

Lemma 4.16 ► Let $pred1$, $pred2$, and $pred_u$ be time predicates. Let T be a cycle. If $pred2$ implies $pred1$ for all cycles T'' with $T'' \geq T$, and $pred2$ stays until $pred_u$ after cycle T, $pred1$ also stays until $pred_u$ after cycle T.

$$\begin{gathered} (\forall T'' \mid T'' \geq T : pred2(T'') \Longrightarrow pred1(T'')) \land \\ stays\_until(pred2, T, pred_u) \\ \Longrightarrow stays\_until(pred1, T, pred_u) \end{gathered}$$

PROOF This lemma is shown easily by expanding the definition of $stays\_until$.

Lemma 4.17 ► Let $pred1$, $pred2$, and $pred_u$ be time predicates. Let T be a cycle. If both $pred1$ and $pred2$ stay until $pred_u$ after cycle T, the conjunction $pred1 \land pred2$ also stays until $pred_u$ after cycle T.

$$\begin{gathered} stays\_until(pred1, T, pred_u) \land stays\_until(pred2, T, pred_u)) \\ \Longrightarrow stays\_until(pred1 \land pred2, T, pred_u) \end{gathered}$$

PROOF This lemma is shown easily by expanding the definition of $stays\_until$. An example for the lemma is given in figure 4.13.

Definition 4.6 ► $\exists^{\geq T}(pred, pred_u)$ Let $pred$ and $pred_u$ be two time predicates. In analogy to the definition of $\exists^{\geq T'}$ (equation 3.25, page 86), one defines an operator that holds iff the predicate $pred$ eventually becomes true in a cycle $T \geq T'$ before $pred_u$

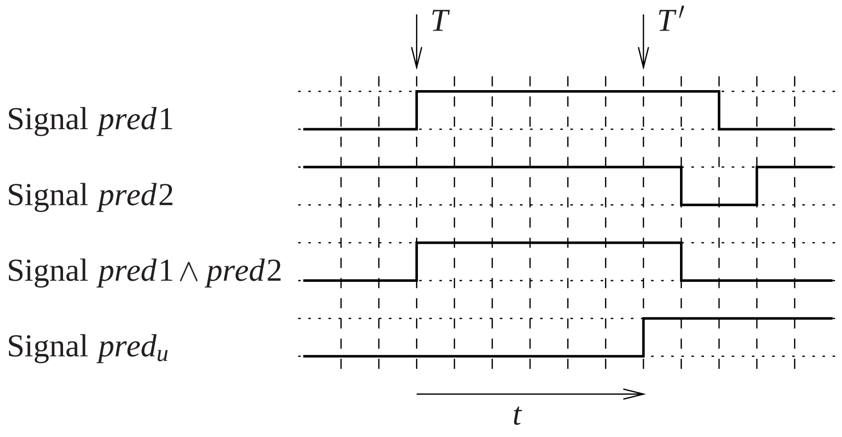


Figure 4.13 Three signals $pred1$, $pred2$, and $pred_u$ satisfying the premise of lemma 4.17: since $stays\_until(pred1, T, pred_u)$ and $stays\_until(pred1, T, pred_u)$ hold, also $stays\_until(pred1 \wedge pred2, T, pred_u)$ holds.

does. Furthermore, it is required that it stays true until $pred_u$ becomes true, as defined in definition 4.5:

$$\begin{aligned}\exists^{\geq T}(pred, pred_u) \quad :\Longleftrightarrow \quad & \exists T' | T' \geq T : pred(T') \wedge \\ & \forall T'' | T \leq T'' < T' : \overline{pred_u(T'')} \wedge \\ & stays\_until(pred, T', pred_u)\end{aligned}$$

This definition is illustrated in figure 4.14.

◄ Lemma 4.18

One easily shows that for any time predicate $pred_u$, $\exists^{\geq T}(pred, pred_u)$ implies that $\exists^{\geq T} pred$ holds:

$$\exists^{\geq T}(pred, pred_u) \quad \Longrightarrow \quad \exists^{\geq T} pred$$

◄ Definition 4.7 *Finite False and Stays Until*

A time predicate *pred* is said to be *finite false and stays until* a given predicate $pred_u$, iff $\exists^{\geq T}(pred, pred_u)$ holds for all $\underline{T}$. In analogy to that, *pred* is said to be *finite true and stays until* $pred_u$ iff $\overline{pred}$ is finite false and stays until $pred_u$.

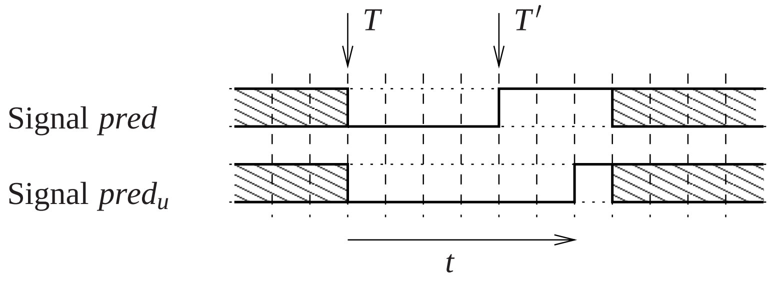


Figure 4.14 Two signals satisfying $\exists^{\geq T}(pred, pred_u)$

The following lemmas are shown easily using lemma 4.18:

Lemma 4.19 ► Let *pred* and $pred_u$ be time predicates. If *pred* is finite false and stays until $pred_u$, it is also finite false as defined in definition 3.4 (page 86).

Lemma 4.20 ► Let *pred* and $pred_u$ be time predicates. If *pred* is finite true and stays until $pred_u$, it is also finite true.

Lemma 4.21 ► Given two time predicates $pred1$ and $pred2$ with $\exists^{\geq T}(pred1, pred_u)$ and $\exists^{\geq T}(pred2, pred_u)$, the conjunction eventually holds after T and before $pred_u$, and stays until $pred_u$.

$$\begin{aligned} &\exists^{\geq T}(pred1, pred_u) \;\wedge\; \exists^{\geq T}(pred2, pred_u) \\ \Longrightarrow\quad &\exists^{\geq T}(pred1 \wedge pred2, T, pred_u) \end{aligned}$$

PROOF By expanding the definition of $\exists^{\geq T}(pred1 \wedge pred2, T, pred_u)$, one gets:

$$\begin{aligned} &\exists T' | T' \geq T : pred1(T') \wedge pred2(T') \wedge \\ &\forall T'' | T \leq T'' < T' : \overline{pred_u(T'')} \wedge \\ &stays\_until(pred1 \wedge pred2, T', pred_u) \end{aligned} \tag{4.36}$$

Since $\exists^{\geq T}(pred1, pred_u)$ and $\exists^{\geq T}(pred2, pred_u)$ hold, there are cycles $t'_1 \geq T$ and $t'_2 \geq T$ such that $pred1(t'_1)$ and $pred2(t'_2)$ hold. Let $t'_1 \geq t'_2$ hold (otherwise, swap $pred1$ and $pred2$[3]). An example for this situation is given in figure 4.15.

[3]In PVS, one actually shows the case $t'_1 < t'_2$ by replaying the proof.

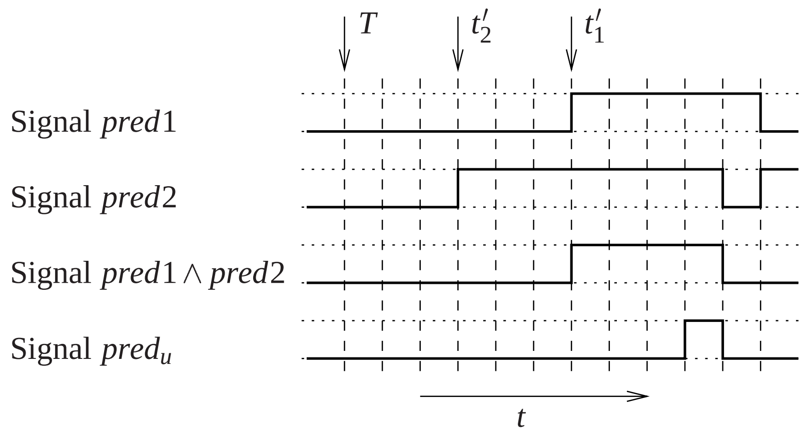


Figure 4.15 Illustration of the proof of lemma 4.21

We will now show that t'_1 satisfies equation 4.36, i.e.:

$$\begin{gathered} pred1(t'_1) \wedge pred2(t'_1) \wedge \\ \forall T''|T \leq T'' < t'_1 : \overline{pred_u(T'')} \wedge \\ stays\_until(pred1 \wedge pred2, t'_1, pred_u) \end{gathered} \tag{4.37}$$

This conjunction consists of four parts, which are now shown separately:

1. As described above, $pred1(t'_1)$ holds by definition of t'_1.
2. The second part, $pred2(t'_1)$, is shown using $\exists^{\geq T}(pred2, pred_u)$: As described above, $pred2(t'_2)$ holds with $t'_2 \leq t'_1$. Furthermore, $pred2$ stays active until $pred_u$ holds, which is after t'_1. Thus, $pred2(t'_1)$ holds.
3. One easily shows $\forall T''|T \leq T'' < t'_1 : \overline{pred_u(T'')}$ by expanding the definition of $\exists^{\geq T}(pred1, pred_u)$.
4. Using lemma 4.15 with $pred2$ and $pred_u$ and cycles T and t'_1, one concludes:

$$stays\_until(pred2, t'_1, pred_u) \tag{4.38}$$

   This allows using lemma 4.17 for $pred1$ and $pred2$ and cycle t'_1:

$$stays\_until(pred1 \wedge pred2, t'_1, pred_u) \tag{4.39}$$

This concludes the claim.

Using lemma 4.21, one easily shows:

Lemma 4.22 ► Given two time predicates $pred1$ and $pred2$ that are both finite false and stay until $pred_u$, the conjunction $pred1 \wedge pred2$ is also finite false and stays until $pred_u$.

Lemma 4.23 ► Given two time predicates $pred1$ and $pred2$ that are both finite true and stay until $pred_u$, the disjunction $pred1 \vee pred2$ is also finite true and stays until $pred_u$.

PROOF Lemma 4.23 is shown easily using lemma 4.22 and the fact that

$$\overline{pred1 \vee pred2} \quad = \quad \overline{pred1} \wedge \overline{pred2} \tag{4.40}$$

QED holds.

The following two lemmas obviously hold (PVS shows them automatically):

Lemma 4.24 ► The predicate *always* (equation 3.22 page 86) is finite false and stays until any predicate.

Lemma 4.25 ► The preciate *never* (equation 3.23 page 86) is finite true and stays until any predicate.

The following lemma is shown easily (PVS shows it automatically):

Lemma 4.26 ► Let $pred1$ and $pred2$ be two time predicates. If $pred1$ holds eventually after cycle T, the disjunction $pred1 \vee pred2$ also holds eventually after cycle T:

$$\exists^{\geq T} pred1 \quad \Longrightarrow \quad \exists^{\geq T}(pred1 \vee pred2)$$

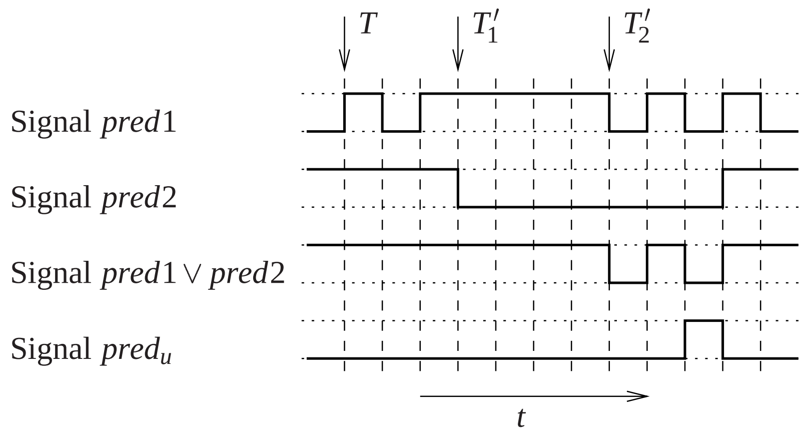


Figure 4.16 Illustration of lemma 4.29

Using lemma 4.26, one easily concludes:

Let *pred*1 and *pred*2 be two time predicates. If *pred*1 is finite false, the disjunction $pred1 \vee pred2$ is also finite false. ◂ Lemma 4.27

Using lemma 4.27 and the definition of finite true, one easily concludes:

Let *pred*1 and *pred*2 be two time predicates. If *pred*1 is finite true, the conjunction $pred1 \wedge pred2$ is also finite true. ◂ Lemma 4.28

Assume one has the disjunction of two signals. One signal is finite true and stays until $pred_u$, the other one is just finite true but implies $\overline{pred_u}$. In this case one can conclude that the disjunction is finite true. This is illustrated in figure 4.16.

Let *pred*1, *pred*2, and $pred_u$ be time predicates. If *pred*1 is finite true and *pred*2 is true false and stays until $pred_u$, and *pred*1 implies $\overline{pred_u}$, the disjunction $pred1 \vee pred2$ is finite true. ◂ Lemma 4.29

The claim is equivalent to: PROOF

$$\forall T \exists^{\geq T} \overline{pred1(T) \vee pred2(T)} \tag{4.41}$$

By definition of $\exists^{\geq T}$, this is equivalent to:

$$\forall T \exists T' \geq T : \overline{pred1(T') \vee pred2(T')} \tag{4.42}$$

Obviously, this is equivalent to:

$$\forall T \exists T' \geq T : (\overline{pred1(T')} \wedge \overline{pred2(T')}) \tag{4.43}$$

According to the premise of the lemma, there is a cycle $T_1' \geq T'$ such that $\overline{pred2}$ holds and stays until $pred_u$. Furthermore, there is also a cycle $T_2' \geq T_1'$ such that $\overline{pred1}$ holds. Let T_2' be the smallest such cycle.

We will now show that cycle T_2' satisfies the claim (equation 4.43), i.e., it is left to show that $\overline{pred2(T_2')}$ holds. This holds since $\overline{pred2}$ is finite true and stays until $pred_u$. The signal $pred_u$ cannot have been active yet, since $pred1$ is implies $\overline{pred_u}$ and T_2' is the smallest cycle after T_1' such that $\overline{pred1}$ holds.

QED

4.6.3 Liveness Proof

In order to prove the liveness of the machine, we have to show that the stall signal of stage k is finite true. Assuming stage k is full, the stall signal is a disjunction of the external stall signals ext_k and the internal stall signals int_k (equation 4.14). We will need to argue that the internal stall signal int_k is finite true and stays until ue_k.

This will be done by induction. The following lemma will be used in order to do the induction step. It states that if one stalls an arbitrary stage for a time that is long enough, eventually all stages below become empty, i.e., the pipeline drains. Let the time predicate $below\_empty_k(T)$ hold iff all stages below stage k are empty during cycle T:

$$below\_empty_k(T) \quad := \quad \forall j \mid k < j < n : \overline{full_j^T} \tag{4.44}$$

Lemma 4.30 ► Let k be a stage number, i.e., $k \in \{0, \ldots, n-1\}$. Let the stall signals of all stages below stage k be finite true and let T be a cycle. This implies that there is a cycle $T' \geq T$ such that if the update enable signal is off from cycle T to $T'-1$, the full bits of the stages below stage k are off during cycle T'.

$$\begin{aligned} &\exists T' \mid T' \geq T : \\ &\quad (\forall T'' \mid T \leq T'' < T' : \overline{ue_k^{T''}}) \implies below\_empty_k(T') \end{aligned}$$

PROOF As before, this is shown by induction on k beginning with $n-1$ and proceeding from $k+1$ to k. For $k=n-1$, there is nothing to show since there are no stages below. Concluding from $k+1$ to k is done as follows:

Since the stall signals of stages below stage k are assumed to be finite true, stall signal $stall_{k+1}$ is also finite true. Thus, there is a cycle $T_1' \geq T$ such that the stall signal $stall_{k+1}^{T_1'}$ is not active. Let T_1' be the smallest such cycle. According to the premise of the lemma, we have $\overline{ue_k^{T_1'}}$. Accoring to lemma 4.1, this implies

$$\overline{full_{k+1}^{T'+1}}.$$

Thus, stage $k+1$ is empty during cycle $T'+1$.

We now apply the induction premise in order to show that the stages below stage $k+1$ eventually also become empty. According to the induction premise, there is a cycle $T_2' \geq T_1'+1$ such that if the update enable signal ue_{k+1} is off from cycle $T_1'+1$ to $T_2'-1$, all full signals below stage $k+1$ are off.

We will now show that during cycle T_2' all stages below stage k are empty. The first step is to show that the full signal of stage $k+1$ actually stays empty until cycle T_2':

$$\forall T'' \mid T_1'+1 \leq T \leq T_2' : \overline{full_{k+1}^{T''}} \tag{4.45}$$

This is done easily by induction on T''. For $T''=T_1'+1$, we already showed the claim above. For cycle $T''+1$, one uses the fact that the full signal is not active in cycle T''. Thus, the stall signal cannot be active. The update enable signal is not active by the premise of the lemma. Thus, the claim can be concluded using lemma 4.1.

It is left to show that the the update enable signal of stage $k+1$ is not active from cycle $T_1'+1$ to cycle $T_2'-1$. This is easily argued since the stage is not full. This concludes the claim. QED

◄ Lemma 4.31

Let k be a stage number but not the last stage. If all stages below stage k are empty during cycle T, this stays so until the output registers of stage k are updated.

$$below\_empty_k(T) \quad \Longrightarrow \quad stays\_until(below\_empty_k, T, ue_k)$$

PROOF By definition of *stays_until* (definition 4.5, page 130), we have to show:

$$\forall T' \mid T' \geq T : \quad (\forall T'' \mid T \leq T'' < T' : \overline{ue_k(T'')}) \tag{4.46}$$

$$\Longrightarrow \quad (\forall T'' \mid T \leq T'' \leq T' : below\_empty_k(T'')) \tag{4.47}$$

This is done by induction on T'. For $T' = T$, the claim holds according to the premise of the lemma. The claim for cycle $T' + 1$ is concluded as follows: The claim is:

$$\forall T'' \mid T \leq T'' \leq (T' + 1) : below\_empty_k(T'') \tag{4.48}$$

For $T \leq T'' \leq T'$, this holds according to the induction premise. Thus, it is left to show this for $T'' = T' + 1$. By definition of $below\_empty_k$, the claim is equal to:

$$\forall j \mid k < j < n : \overline{full_j^{T'+1}} \tag{4.49}$$

Case one: If j is equal to $k+1$, we show $\overline{full_{k+1}^{T'+1}}$ as follows: according to lemma 4.1, a stage becomes full if it was either stalled or if the output registers of the previous stage were updated. The update enable signal of the previous stage, which is stage k, is not active according to the premise of the lemma.

$$\begin{aligned} full_{k+1}^{T'+1} &= ue_k^{T'} \vee stall_{k+1}^{T'} && \text{by lemma 4.1} \\ &= stall_{k+1}^{T'} && \text{because of } \overline{ue_k^{T'}} \end{aligned}$$

The stall signal $stall_{k+1}^{T'}$ cannot be active since stage $k+1$ is not full during cycle T' according to the induction premise. Thus, $full_{k+1}^{T'+1}$ is not active.

Case two: If j is not equal to $k+1$, we show $\overline{full_j^{T'+1}}$ as follows:

$$\begin{aligned} full_j^{T'+1} &= ue_{j-1}^{T'} \vee stall_j^{T'} && \text{by lemma 4.1} \\ &= (full_{j-1}^{T'} \wedge \overline{stall_{j-1}^{T'}}) \vee stall_j^{T'} && \text{because of def. of } ue \end{aligned}$$

The stall signal $stall_j^{T'}$ cannot be active since stage j is not full during cycle T' according to the induction premise. The full signal $full_{j-1}^{T'}$ is also not active because of the induction premise. This concludes the claim. QED

Lemma 4.32 ▶ Assuming that the external stall signals are finite true and stay until ue_k, the disjunction of the external stall signals ext_k is finite true.

PROOF Using lemma 4.23 one concludes that the disjunction is finite true and stays until ue_k. Using lemma 4.20, one concludes that ext_k is finite true.

Lemma 4.33 Let k be a stage number but not the last stage. If the stages below stage k are empty, the internal stall signal int_k is off.

$$below\_empty(k,T) \implies \overline{int_k^T}$$

PROOF By definition, int_k is:

$$int_k^T = dhaz_k^T \vee stall_{k+1}^T$$

If the stages below stage k are empty, $dhaz_k^T$ cannot be active by definition (empty stages never generate a data hazard). If the stages below stage k are empty, so is stage $k+1$. Thus, $stall_{k+1}^T$ cannot be active according to the stall signal convention (convention 4.2). QED

In the following, we will conclude that $stall_k$ is finite true from the same claim for $stall_l$ with $l > k$. The signal $stall_k$ includes the internal signal as defined in equation 4.13. Thus, one has to show that the internal stall signal eventually gets deactivated and stays so until the update enable signal is activated. This is done as follows:

> The internal stall signal of stage k is deactivated if the stages below stage k are empty, *at the latest*.

The term "at the latest", as used in the last sentence, will be formalized by the next lemma. In the last sentences, three time predicates are used:

1. "The internal stall signal … is deactivated" will be referred to by time predicate $pred1$,
2. "the update enable signal is activated" will be referred to by time predicate $pred_u$,
3. "the stages below stage k are empty" will be referred to by time predicate $pred2$.

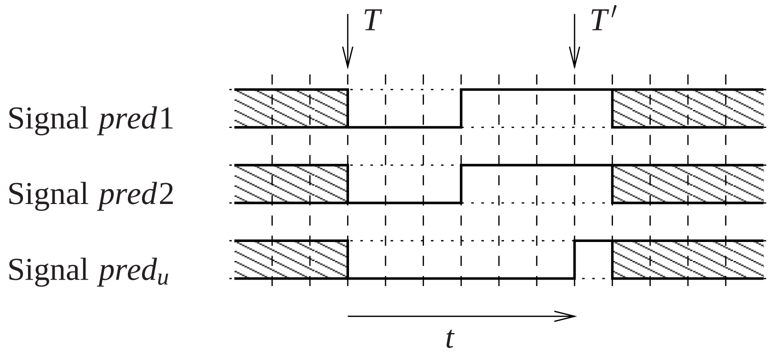


Figure 4.17 Illustration of lemma 4.34: Since $pred2$ implies $pred1$ and $pred2$ becomes active, $pred1$ also becomes active.

According to lemma 4.33, $pred2$ obviously implies $pred1$ (empty stages never generate a hazard or stall signal). Furthermore, one easily shows that $pred_u$ also implies $pred1$ (the update enable signal is not active as long as the stage is stalled). The notion "at the latest" will now be formalized as follows: $pred1$ holds if $pred2$ holds "at the latest" means that assuming $pred1$ does not hold for a time that is long enough, $pred2$ holds eventually. Now there are two cases:

a) The predicate $pred2$ becomes true. Since $pred2$ implies $pred1$, one can conclude that $pred1$ will hold eventually. This case is illustrated in figure 4.17.

b) The predicate $pred1$ becomes true before $pred2$. However, this does not imply that $pred2$ will hold eventually. This case is illustrated in figure 4.18.

The following lemma formalizes this claim:

Lemma 4.34 ► Let T be a cycle, $pred1$, $pred2$, and $pred_u$ time predicates. Furthermore, let the following conditions hold:

1. Let both $pred_u$ and $pred2$ imply $pred1$ after cycle T.

$$\forall T'' \mid T'' \geq T : pred_u(T'') \Longrightarrow pred1(T'')$$
$$\forall T'' \mid T'' \geq T : pred2(T'') \Longrightarrow pred1(T'')$$

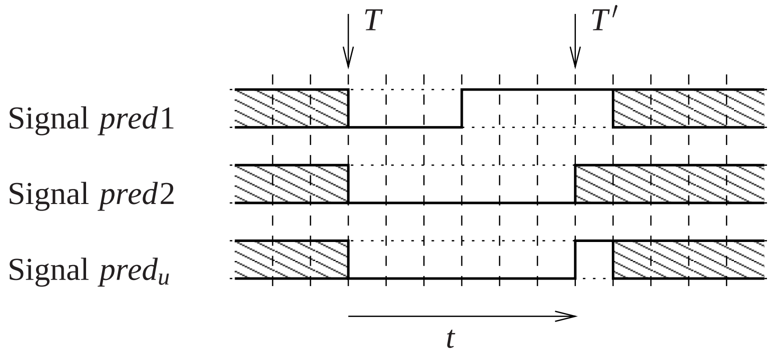


Figure 4.18 Illustration of lemma 4.34: *pred*1 becomes active before *pred*2.

2. Let there be a cycle $T_1' \geq T$ such that if $\overline{pred_u}$ holds for all cycles T'' with $T \leq T'' < T_1'$ then $pred2(T_1')$ holds.

$$\exists T_1' \mid T_1' \geq T : \forall T'' \mid T \leq T'' < T_1' : \overline{pred_u(T'')} \Longrightarrow pred2(T_1')$$

3. If $pred2$ holds in any given cycle $T' \geq T$, it is supposed to stay until $pred_u$ after T'.

$$\forall T' \mid T' \geq T : pred2(T') \Longrightarrow stays\_until(pred2, T', pred_u)$$

The claim is that this implies $\exists^{\geq T}(pred1, pred_u)$.

PROOF

By expanding the definition of $\exists^{\geq T}(pred1, pred_u)$, one gets:

$$\begin{gathered} \exists T' | T' \geq T : pred1(T') \wedge \\ \forall T'' | T \leq T'' < T' : \overline{pred_u(T'')} \wedge \\ stays\_until(pred1, T', pred_u) \end{gathered}$$

Let $\exists^{\geq T} pred_u$ hold. In this case, there is a cycle $T' \geq T$ such that $pred_u$ is active. Let this be the smallest cycle with this property, which exists according to lemma 3.19. We will now show that this cycle satisfies the claim. According to the first condition above, $pred1(T')$ holds. Since T' is the smallest cycle such that $pred_u$ is active,

$$\forall T'' | T \leq T'' < T' : \overline{pred_u(T'')}$$

obviously holds. One easily shows

$$stays\_until(pred1, T', pred_u)$$

by using the fact that $pred_u(T')$ holds. If $\exists^{\geq T} pred_u$ holds, this concludes the claim.

Assume $\exists^{\geq T} pred_u$ does not hold. In this case, $pred_u$ never holds in any cycle $T' \geq T$. This allows using the second condition above in order to conclude that there is a cycle $T_1' \geq T$ such that $pred2(T_1')$ holds. We will now show that this cycle satisfies the claim.

According to the first condition above, $pred1(T_1')$ holds. As $\exists^{\geq T} pred_u$ does not hold, one can conclude that

$$\forall T'' | T \leq T'' < T_1' : \overline{pred_u(T'')}$$

holds. Using the third condition above, one easily concludes that

$$stays\_until(pred2, T_1', pred_u)$$

holds. Using the first condition and lemma 4.16, one shows that

$$stays\_until(pred1, T_1', pred_u)$$

QED holds. This concludes the claim.

Lemma 4.35 ► Assuming that the external stall signals are finite true and stay until ue_k, the stall signal is finite true.

PROOF The proof proceeds by induction on k. We begin with the last stage. The induction step is done by concluding the claim for stage k from the claim for stages $l > k$.

For stage $k = n-1$ (i.e., for the last stage), the claim is shown as follows: in case of the last stage, no forwarding is done, i.e., $dhaz_{n-1}$ is always false. Thus, the stall signal of the last stage is:

$$stall_{n-1}^T \quad = \quad full_{n-1}^T \wedge ext_{n-1}^T \tag{4.50}$$

According to lemma 4.32, this is finite true.

The induction claim for stage $k < (n-1)$ is shown as follows: The stall signal of stage k is:

$$stall_k^T \quad = \quad full_k^T \wedge (ext_k^T \vee int_k^T) \tag{4.51}$$

Using lemma 4.28, one concludes that it is sufficient to show that

$$ext_k^T \vee int_k^T \tag{4.52}$$

is finite true. This is concluded by lemma 4.29 using the predicates ext_k, int_k, and ue_k. In order to apply lemma 4.29, one has to show that the premises of the lemma hold. These premises are:

- The predicate ext_k must be finite true,
- the predicate int_k must be finite true and stay until ue_k,
- the predicate ext_k must imply $\overline{ue_k}$.

The first premise holds according to lemma 4.32. The third premise holds according to the definition of ue_k and $stall_k$. It is left to show that the second premise holds, i.e., that int_k is finite true and stay until ue_k. This is done by using lemma 4.34 as described above.

One now easily concludes the liveness criterion for the pipelined machine:

◀ Theorem 4.36

Assuming that the external stall signals of stage k are finite true and stay until ue_k for all stages k, the pipelined machine is alive.

PROOF

Using lemma 4.35, one concludes that the stall signals are finite true. As in theorem 3.24 (page 88), one concludes that the machine is alive.

4.7 Performance

The machine presented in this chapter almost matches the pipelined DLX presented in [MP00]. One major difference is the stall engine. The stall engine in [MP00] uses only two different clock enable signals. The first clock enable signal controls stages 0 and 1 and the second clock enable signal controls the rest of the pipeline. Thus, stages 0 and 1 are always clocked simultaneously. The same holds for stages 2, 3, and 4.

In contrast to that, the stall engine used in this thesis supports stalling all stages independently. This improves performance. Consider the following example in a five stage integer DLX: The first instruction is a load instruction (`LW`). Let the destination register of this instruction be `R1`. The second instruction is an ALU instruction that calculates the disjunction of

Cycle	1	2	3	4	5	...	10	11	...
IF	`LW`	`ORI`	`ADD`	`ADD`	`ADD`		`SUB`	`XOR`	
ID		`LW`	`ORI`	`ORI`	`ORI`		`ADD`	`SUB`	
EX			`LW`	Bubb.	Bubb.	...	`ORI`	`ADD`	
MEM				`LW`	`LW`		Bubb.	`ORI`	
WB							`LW`	Bubb.	

Figure 4.19 Scheduling in [MP00]: The cache miss in the MEM stage stalls the pipeline completely.

Cycle	1	2	3	4	5	...	10	11	...
IF	`LW`	`ORI`	`ADD`	`ADD`	`SUB`		`XOR`	`SW`	
ID		`LW`	`ORI`	`ORI`	`AND`		`SUB`	`XOR`	
EX			`LW`	Bubb.	`ORI`	...	`ADD`	`SUB`	
MEM				`LW`	`LW`		`ORI`	`ADD`	
WB							`LW`	`ORI`	

Figure 4.20 Scheduling in this thesis: the bubble introduced because of the data hazard is removed. The execution differs from [MP00] beginning with cycle 5.

a register value and an immediate constant (`ORI`). Let register `R1` be the source register. In stage ID, the machine is supposed to read the operand register. However, this register is not yet available in this stage because the load has not yet completed. Thus, in both machines a pipeline bubble is inserted (figure 4.19, cycle 4).

Assume that load instruction causes a data cache miss in stage MEM. The machine in [MP00] stalls the execution completely. In contrast to that, the machine presented in this thesis keeps stages 0 to 2 running for one cycle more by removing the pipeline bubble in stage 2 (figure 4.20, cycle 5). Assume that the data word required for the load instruction is available by cycle 10 in both machines. In the machine presented in [MP00], the bubble proceeds until it reaches the end of the pipeline.

In order to quantify the performance impact of the new stall engine, we performed simulations using the SPEC92 benchmarks as a workload. In case of integer-only workload, the new stall engine speeds up execution on the five stage DLX pipeline by approximately 1.1%. The speedup increases the more long latency instructions, in particular floating point instructions, are involved. Appendix C gives more details on the simulation environment and the results.

4.8 Literature

The concept of the transformation of a prepared sequential machine into a pipelined machine is taken from [MPK00, MP00]. In addition to that, the design of the pipelined DLX used as example is taken from [MP00].

Flynn's classic textbook [Fly95] on pipelined processors states the following on interlock hardware:

> "As any pipelined processor designer knows, a great deal of engineering effort is required to efficiently realize a fully functional set of interlocks."

However, to best of our knowledge, in most of the literature the details of implementing forwarding and interlock hardware are skipped over, including [Fly95]. An exception is [MP00], which presents the interlock and forwarding logic at gate level. The stalling mechanisms described in the literature including those in [MP00] usually assume that a pipeline bubble floats through the complete pipeline [Fly95, HP96]. In contrast to that, the stall engine presented in this thesis supports removal of pipeline bubbles, which speeds up the execution.

In [LO96], Levitt and Olukotun verify a five-stage DLX pipeline by transforming it back into a sequential machine by removing stalling and rollback logic. Liveness is not argued.

In [Hos00], Hosabettu verifies a simple five stage DLX that is not synthesizeable. It has a trivial stalling logic. Stalls caused by slow memory are not covered. The verification is done using the completion function approach and PVS. Liveness is not argued.

Further literature on the verification of pipelined machines is [BM96], which provides a manual proof of a DLX pipeline, Burch, Dill [BD94] verify a very simple pipeline. Henzinger et.al. [HQR98] use refinement mappings in order to model-check a RISC pipeline. Liveness is not argued.

Chapter

5

Speculative Execution

5.1 Introduction

SPECULATIVE EXECUTION is a technique to avoid stalling the pipeline because of data dependencies in situations that do not permit forwarding. Thus, instead of stalling, the calculation is continued with a value that is guessed. As soon as the correct value is available, the correct value and the guessed value are compared. If both are equal, the calculations made with the guessed value are also correct.

If the guessed value and the correct value are different, all calculations made with the guessed value are usually false. This is called *misspeculation*. In this case, the calculation has to be restarted at the stage the guessing is made. This process is called *rollback* (in the literature, the term *squashing* is often used [LO96]). It includes that all changes made to the state of the machine based on false data have to be reverted. The extra cycles required for the rollback and the wrong calculation are called *misspeculation penalty*.

In this chapter, we will describe a generic method that allows to speculate on arbitrary values. The method includes automatic generation of the circuits necessary to detect a misspeculation and to do the rollback in case of a misspeculation. We will then use the method in order to implement branch prediction and precise interrupts.

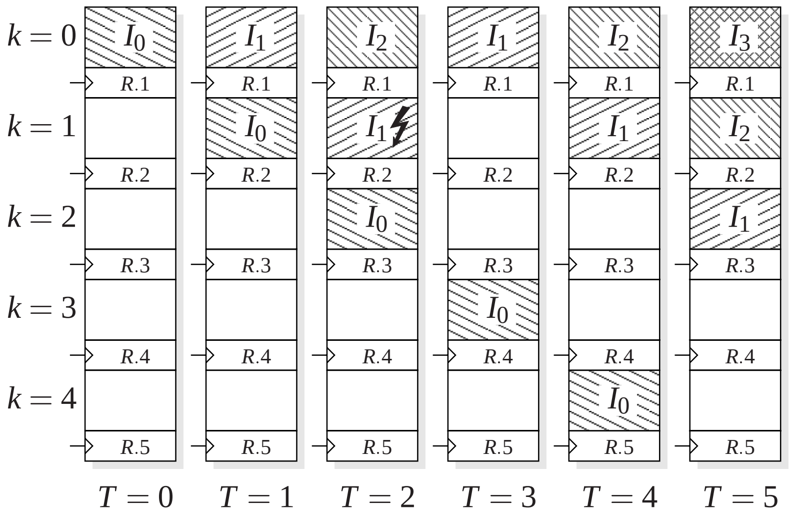


Figure 5.1 Execution of the instructions I_0 to I_3 in a pipelined machine with speculation. Let the *PC* of I_1 be misspeculated. This is detected in stage 2 during cycle $T = 3$, as illustrated by the flash symbol. Stages that are full are hatched.

Example Consider a pipelined machine with five stages. Let us guess (i.e., speculate on) the correct value of the memory address used for the instruction fetch (denoted by *PC*) in stage 0. Assume that the correct value is available in stage $k = 1$ (decode).

Figure 5.1 gives an example what can happen in such a machine: let the mechanism guess the value of *PC* of the instruction I_0 correctly but not of instruction I_1. The machine runs as usual until cycle $T = 2$. In cycle $T = 2$, instruction I_1 is in stage 1 and the misspeculation is detected. Thus, instruction I_1 has to be restarted completely. Assume that this takes one cycle.

In cycle $T = 3$, instruction I_1 therefore is in stage 0 again. The calculation re-starts using the correct value of *PC* that is now known. The instruction I_2 is completely evicted from the pipeline. Note that, however, instruction I_0 proceeds (and terminates) as before. This is justified by the fact that instruction I_0 does not depend on any data that was misspeculated. Table 5.1 shows the schedule of this example.

However, the instruction I_1 might have made changes to the registers in $out(0)$. Instruction I_1 usually relies on the original values, i.e., the values written by I_0. Thus, one has to ensure that instruction I_1 calculates its

	$T=0$	$T=1$	$T=2$	$T=3$	$T=4$	$T=5$
$sI(0,T)$	0	1	2	1	2	3
$sI(1,T)$	0	0	1	1	1	2
$sI(2,T)$	0	0	0	1	1	1
$sI(3,T)$	0	0	0	0	1	1
$sI(4,T)$	0	0	0	0	0	1

Table 5.1 The values of sI in a five stage pipelined machine with speculation

inputs using the values written by I_0 and not I_1. We will now describe how such a mechanism is implemented.

5.2 Stall Engine with Speculation

In this section, we will describe a simple generic speculation mechanism that allows speculating on values of arbitrary implementation registers. The first step is to modify the stall engine such that we can evict instructions from the pipeline in case of misspeculation. For this purpose, we introduce signals $rollback_k$ with $k \in \{0,\dots,n-1\}$. The signal $rollback_k$ is to be activated if misspeculation is detected in stage k. We will later on describe how we detect misspeculation.

Using these signals, a set of signals $rollback'_k$ is defined. The signal $rollback'_k$ is active if the instruction in stage k has to be squashed because of misspeculation. Assume a signal $rollback_k$ is active. In this case, one has to evict all instructions in the stages 0 to k. Thus, $rollback'_k$ is active if a rollback signal of any later stage is active:

$$rollback'_k \quad = \quad \bigvee_{i=k}^{n-1} rollback_i \tag{5.1}$$

One easily speeds up this computation using the parallel prefix circuit as described in section 2.2.4. Using the signals $rollback'$, we make the following changes to the stall engine:

- The update enable signal of a stage k is deactivated if the rollback signal is active. Let ue'_k denote the old update enable signal as used in the previous chapters. Let ue_k denote the new update enable signal. The new update enable signal is:

$$ue_k \quad := \quad ue'_k \wedge \overline{rollback'_k} \tag{5.2}$$

- The transition function for the full bits is changed as follows: Let $\delta'.full.k$ denote the old transition function and let $\delta.full.k$ denote the new one. The new transition function for $k \in \{1,\dots,n-1\}$ is:

$$\delta.full.k \quad := \quad \delta'.full.k \wedge \overline{rollback'_k} \tag{5.3}$$

The following simple lemmas are concluded from the new definition of the signals and the new transition functions:

Lemma 5.1 ▶ A stage is full iff it was updated or stalled in the previous cycle and if there was no rollback:

$$\forall k \geq 1: \quad full_k^{T+1} = (ue_{k-1}^T \vee stall_k^T) \wedge \overline{rollback'^T_k}$$

The signal $full_0$ is always active:

$$full_0^T = 1$$

All other signals $full_k$ are not active during cycle 0:

$$\forall k \geq 1: \quad full_k^0 = 0$$

This lemma is a counterpart of lemma 4.1 of the pipelined machine without speculation.

Lemma 5.2 ▶ If a stage is full and is updated, the next stage is updated, too.

$$\forall k \geq 1: \quad full_k^T \wedge ue_{k-1}^T \quad \Longrightarrow \quad ue_k^T$$

This lemma is a counterpart of lemma 4.3 of the pipelined machine without speculation.

PROOF According to the definition of the update enable signals, we have to show 1) $full_k^T$, 2) $\overline{stall_k^T}$, and 3) $\overline{rollback'^T_k}$.

According to the premise of the lemma, $full_k^T$ holds. We show $\overline{stall_k^T}$ as in the proof of lemma 4.3.

We show $\overline{rollback'^T_k}$ as follows: assume $rollback'^T_k$ holds. In this case, $rollback'^T_{k-1}$ also holds. Thus, ue^T_{k-1} cannot be active. This is a contradiction to the premise of the lemma.

◄ Lemma 5.3

If a stage is full and if its output registers are not updated and if no rollback is made, the full bit is preserved.

$$\forall k \geq 1: \quad full_k^T \wedge \overline{ue_k^T} \wedge \overline{rollback'^T_k} \quad \Longrightarrow \quad full_k^{T+1}$$

PROOF

By the definition of the update enable signals, one concludes that $stall_k^T$ holds. The claim is concluded using lemma 5.1.

◄ Lemma 5.4

If a configuration in a stage moves into the next stage (i.e., the output registers of a stage are updated), and if the next configuration is not clocked into the stage, the full bit is cleared:

$$\forall k \geq 1: \quad full_k^T \wedge ue_k^T \wedge \overline{ue_{k-1}^T} \quad \Longrightarrow \quad \overline{full_k^{T+1}}$$

PROOF

By the definition of the update enable signals, one concludes $\overline{stall_k^T}$ and $\overline{rollback'^T_k}$. The claim is concluded by lemma 5.1.

The following lemma is the counterpart of lemma 4.6 in the pipelined machine without speculation. The proof is proceeds as in chapter 4.

◄ Lemma 5.5

Stage k is full at the earliest in cycle k.

$$full_k^T \quad \Longrightarrow \quad T \geq k$$

5.3 Schedule with Speculation

Using the signals $rollback'_k$, it is possible to give a recursive specification of a scheduling function $sI(k,T)$ for the pipelined machine with speculation that reflects the changes caused by a rollback.

It is constructed as follows: In “normal operation”, i.e., if no speculation is made, the scheduling function should match the scheduling function of the pipelined machine without speculation. However, in case of a rollback, the scheduling function must provide values such that the instructions that are evicted never entered the pipeline.

This allows for a recursive definition of the scheduling function of the prepared sequential machine: For sake of simplicity, we split the definition of the function into three cases: 1) T=0, 2) a rollback is made, and 3) no rollback is made.

If $T = 0$ holds, $sI(k,T)$ is zero, just as before:

$$sI(k,0) \quad := \quad 0 \tag{5.4}$$

If $T \neq 0$ holds and if no rollback is made, i.e., $rollback'^{T-1}_k$ does not hold, we use the definition from chapter 3:

$$sI(k,T) \quad := \quad \begin{cases} sI(k,T-1) & : \quad \overline{ue^{T-1}_k} \\ sI(0,T-1)+1 & : \quad ue^{T-1}_k \wedge k=0 \\ sI(k-1,T-1) & : \quad ue^{T-1}_k \wedge k \neq 0 \end{cases}$$

If $T \neq 0$ holds and a rollback is made, i.e., $rollback'^{T-1}_k$ holds, we aim to provide values as if the instructions that are evicted never were put into the pipeline.

Assume the following example: Instruction I_0 does not use speculation and proceeds through the pipeline as usual. Instruction I_1 uses speculation and we misspeculate. This is detected in cycle $T = 3$ and stage $k = 2$. In table 5.2, we depict a standard pipelined schedule such that I_1 is not put into the pipeline before cycle $T = 4$. In table 5.3, we depict a schedule such that I_1 uses speculation instead. Note that the schedules match after the rollback in cycle $T = 4$.

In this example, during cycle $T = 3$, the following signals are active: because of the misspeculation, $rollback^3_2$ is active. This implies that the signals $rollback'_0$ to $rollback'_2$ are active by definition of these signals.

We construct the scheduling function for this case as follows: for all stages with rollback, we take the value of the scheduling function from cycle $T-1$ from the last stage in that we detect a rollback. If $rollback'_{n-1}$ is active, this is stage $n-1$. If not so, this is stage k such that $rollback'_k$ holds

T	0	1	2	3	4	5	6	7	8
$sI(0,T)$	0	1	1	1	1	2	3	4	5
$sI(1,T)$	0	0	1	1	1	1	2	3	4
$sI(2,T)$	0	0	0	1	1	1	1	2	3
$sI(3,T)$	0	0	0	0	1	1	1	1	2
$sI(4,T)$	0	0	0	0	0	1	1	1	1

Table 5.2 The values of *sI* in a five stage pipelined machine without speculation. Instruction I_1 is delayed until cycle 5.

T	0	1	2	3	4	5	6	7	8
$sI(0,T)$	0	1	2	3	**1**	2	3	4	5
$sI(1,T)$	0	0	1	2	**1**	1	2	3	4
$sI(2,T)$	0	0	0	1	**1**	1	1	2	3
$sI(3,T)$	0	0	0	0	1	1	1	1	2
$sI(4,T)$	0	0	0	0	0	1	1	1	1

Table 5.3 The values of *sI* in a five stage pipelined machine with speculation. We misspeculate on instruction I_1 and detect this in cycle 4. In cycle 5, the execution proceeds as if instruction I_1 was delayed until cycle 5.

	$T=0$	$T=1$	$T=2$	$T=3$	$T=4$	$T=5$
$sI(0,T)$	0	1	2	3	1	2
$sI(1,T)$	0	0	1	2	1	1
$sI(2,T)$	0	0	0	1	1	1
$sI(3,T)$	0	0	0	0	1	1
$sI(4,T)$	0	0	0	0	0	1

Table 5.4 Illustration of the recursion made for $sI(k,4)$ in case of a rollback.

but $rollback'_{k+1}$ does not. We use the predicate $\rho(k,T)$ as a shorthand:

$$\rho(k,T) \quad :\Longleftrightarrow \quad rollback'^{T}_{k} \wedge (k=n-1 \vee \overline{rollback'^{T}_{k+1}})$$

We assert the claim above in the following lemma:

Lemma 5.6 ▶ The construction described above provides the last stage with active rollback signal.

$$\rho(k,T) \quad \Longrightarrow \quad k = \max\{j \in \{0,\ldots,n-1\} \mid rollback^{T}_{j}\}$$

PROOF One easily shows this inductively using the fact that if $rollback'_{k+1}$ is active, this implies that the signal $rollback'_{k}$ is also active.

Thus, if $\rho(k,T-1)$ holds, we take $sI(k,T-1)$ as value for $sI(k,T)$. If it does not hold, we use recursion in order to get the desired value: we walk down the pipeline from k to $k+1$ until $\rho(k,T-1)$ holds:

$$sI(k,T) \quad := \quad \begin{cases} sI(k,T-1) & : \quad \rho(k,T-1) \\ sI(k+1,T) & : \quad \text{otherwise} \end{cases}$$

Obviously, this simplifies to:

$$sI(k,T) \quad := \quad \begin{cases} sI(k,T-1) & : \quad k=n-1 \vee \overline{rollback'^{T-1}_{k+1}} \\ sI(k+1,T) & : \quad \text{otherwise} \end{cases}$$

This recursion is illustrated in table 5.4. It is no longer obvious that this recursion terminates for all values k and T. One argues as follows: the recursion terminates as soon as $T=0$ is reached. In case of no rollback, T decreases by one. In case of a rollback, either T decreases or k decreases. However, T decreases if the end of the pipeline is reached at the latest, i.e., if $k=n-1$ holds.

5.4 Scheduling Invariants

In this section, we will show that the scheduling invariants presented in chapter 3 still hold for the stall engine of the pipelined machine with interrupts. We have to make a small change to invariant one for the rollback case. Invariants two and three still hold without any change.

◀ Invariant 5.1

Assume that the rollback signal $rollback'^{T-1}_{k+1}$ is not active or that k is the last stage. If the update enable signal of stage k is active in cycle $T-1$, the value of the scheduling function for that stage increases by one. If the update enable signal of the stage is not active, the value does not change. For $T > 0$:

$$sI(k,T) \quad = \quad \begin{cases} sI(k,T-1) & \text{if } ue_k^{T-1} = 0 \\ sI(k,T-1)+1 & \text{if } ue_k^{T-1} = 1 \end{cases}$$

◀ Invariant 5.2

Given a cycle T, the values of the scheduling functions of two adjacent stages are either equal or the value of the scheduling function of the earlier stage is greater by one. This also holds in case of a rollback.

◀ Invariant 5.3

The value of the scheduling function of the earlier stage is greater by one iff the full bit of the later stage is set. For $k > 0$:

$$full_k^T = 1 \Leftrightarrow sI(k-1,T) = sI(k,T)+1$$

Negating both sides of the last equation and applying invariant 5.2 results in:

$$full_k^T = 0 \Leftrightarrow sI(k-1,T) = sI(k,T)$$

This also holds in case of a rollback.

PROOF

The proof of the invariants proceeds as in chapter 3: Let $P_i(T)$ denote that invariant i holds for the pipelined machine with speculation for the cycle T. The claim is concluded as in chapter 3:

$$\begin{array}{rcl} P_3(T-1) & \Longrightarrow & P_1(T) \\ P_1(T) \wedge P_2(T-1) \wedge P_3(T-1) & \Longrightarrow & P_2(T) \\ P_1(T) \wedge P_2(T-1) \wedge P_3(T-1) & \Longrightarrow & P_3(T) \end{array}$$

Proof of Invariant 5.1 We make a case split on the value of the rollback signal $rollback'^{T-1}_{k}$:

1. Let the rollback signal $rollback'^{T-1}_{k}$ be active. Since $rollback'^{T-1}_{k+1}$ is not active or k is the last stage, stage k is the last stage with active rollback signal. The update enable signal ue^{T-1}_{k} is not active in this case by definition. Thus, the claim is:

   $$sI(k,T) \quad = \quad sI(k,T-1)$$

   This holds by definition of $sI(k,T)$.

2. Let the rollback signal $rollback'^{T-1}_{k}$ be not active. As we exclude the case of a rollback, the proof proceeds as the proof of invariant 3.1 presented in chapter 3.

Proof of Invariant 5.2 Let us consider the stages $k-1$ and k with $k>0$. Let $rollback'^{T-1}_{k}$ be active. We start with the induction claim:

$$\begin{array}{rrcl} & sI(k-1,T) & = & sI(k,T)+1 \\ \vee & sI(k-1,T) & = & sI(k,T) \end{array} \tag{5.5}$$

The second equation holds because of the definition of $sI(k-1,T)$ and because the rollback signal is active.

Let $rollback'^{T-1}_{k}$ be not active. In this case, k is either the last stage or $rollback'^{T-1}_{k+1}$ is not active. Thus, no rollback is involved and the proof proceeds as the proof of invariant 3.2 in chapter 4.

Proof of Invariant 5.3 For $T=0$, the claim can be shown by definition unfolding and using lemma 5.5. For $T>0$, according to lemma 5.1, the claim is equivalent to:

$$(ue^{T-1}_{k-1} \vee stall^{T-1}_{k}) \wedge \overline{rollback'^{T-1}_{k}} \quad \Longleftrightarrow \quad sI(k-1,T) = sI(k,T)+1$$

As before, the proof in chapter 4 can be repeated if the rollback signal $rollback'^{T-1}_{k}$ is not active. Thus, let $rollback'^{T-1}_{k}$ hold. This implies that $sI(k-1,T)$ is equal to $sI(k,T)$. Thus, the right hand side of the equivalence in the claim cannot hold. The left hand side of the equivalence also does not hold because $rollback'^{T-1}_{k}$ holds. This concludes the claim. QED

5.5 Speculative Inputs

For sake of simplicity, we restrict ourselves to the case that the speculation is done in the first stage. Let R be a denominator for a value we want to guess. Let $R \in \sigma$ denote this fact. The speculation mechanism is added in three steps:

1. The first step is to add functions that do the guessing of the value. We name those functions f_0Rs by convention. These functions are called *speculation functions* and can take arbitrary specification registers as arguments as described in section 3.2.4 (page 41). In analogy to the notation used in the previous chapters, this set of registers is denoted by $dep\_s(R,0)$. All other notation used for register transition functions also applies for the speculation functions.
2. We add registers that record whether we still have to speculate or whether the real value is already known. We denote this register by cR. The domain of this register is one bit. If it is set, the correct value of R is known. If not, we have to speculate. We initialize these registers with zero.

   We furthermore add registers that save the real value in case of a rollback. The registers are named R and have the same domain as the value we are guessing. These registers are initialized with an arbitrary value, e.g., zero.
3. We make the guessed value provided by f_0Rs available as input for the register transition functions of stage 0. We do not allow a recursion here, i.e., the input of a speculation function must not be a speculative value.

   The input generation function g_0R for such a speculative value is defined as follows: in case the bit in cR is set, we return the value in $c.R$. If not so, we guess the value using f_0Rs.

$$g_0R(c_I^T) \quad := \quad \begin{cases} c_I^T.R & : \quad c_I^T.cR = 1 \\ f\gamma_0Rs(c_I^T) & : \quad \text{otherwise} \end{cases} \tag{5.6}$$

5.6 Detecting Misspeculation

The mechanism above allows guessing a register value. The guessed value can be used as normal input to register transition functions. However, we

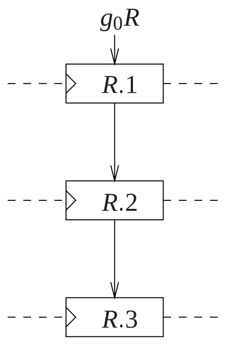


Figure 5.2 The speculative value R is guessed by stage 0 and then stored in registers $R.1$, $R.2$, and so on.

have to detect and handle the case that the speculation fails. In order to detect that we misspeculated, it is necessary to store the value guessed to have it available later on. For $R \in \sigma$, we do so by adding instances of an implementation register named R, i.e., $R.1$, $R.2$, and so on.

If ue_0 is active and such a register $R.1$ with $R \in \sigma$ is updated, one simply writes the value provided by g_0R into the register, i.e., the guessed value. In case of registers $R.k$ with $R \in \sigma$ and $k > 1$, we just take the value from the previous stage, i.e., from $R.(k-1)$. This is depicted in figure 5.2.

In addition to the check for misspeculation, the value in these registers can be used in order to read the speculative value in stages other than the first stage. This is handled just like a normal read access to an implementation register.

A misspeculation is detected as follows: let $R \in out(k)$ be an instance of such a register. If a value is written into the register by write accesses as used in the previous chapters, this value is compared with the value that is in the instance of the register in the previous stage. If they do not match, a misspeculation is detected and a rollback is signaled.

For this purpose, we define a misspeculation signal $R_k misspec$ for each such register $R.(k+1)$. It is active if the value provided by the write access and the value in the register do not match and if the stage is full but not stalled.

$$\begin{aligned} R_k misspec(c_I) \quad := \quad & (f\gamma_k R(c_I) \neq c_I.R.k) \wedge \\ & full_k \wedge \overline{stall_k} \end{aligned} \tag{5.7}$$

This is depicted in figure 5.3. The test for the stall signal is motivated as

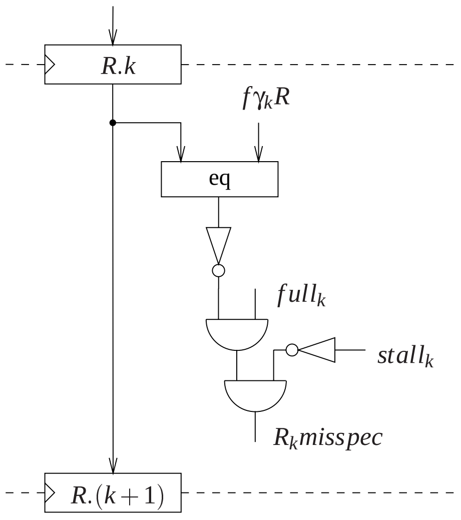


Figure 5.3 The speculative value R is compared with the value provided by the register transition function. If they do not match, a misspeculation is signaled.

follows: the function $f_k R$ takes inputs. These inputs might be forwarded. Thus, they are only guaranteed to be valid if the stall signal is not active. Furthermore, we require that the functions $f_k R$ do not depend on values that are guessed.

We use these signals in order to calculate the rollback signal of stage k: It is just the disjunction of the $R_k misspec$ signals:

$$rollback_k(c_I) \quad := \quad \bigvee_{R.k \in \sigma} R_k misspec(c_I) \tag{5.8}$$

5.7 Rollback

During rollback, we have to revert changes to the machine made by the instructions that used misspeculated data. Thus, the state of the machine has to be changed as if the instructions that used misspeculated data never entered the machine.

The rollback is realized as follows: the original values of the registers that are changed during the speculation are saved in temporary registers. All calculations store their results in the original place as before. If the

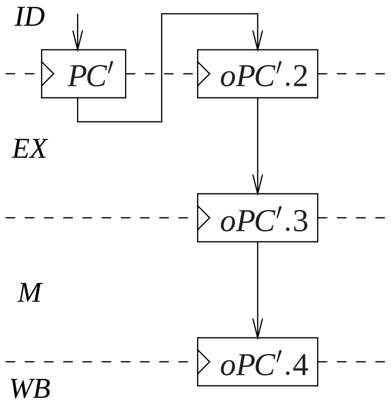


Figure 5.4 Saving the original value of $PC'.2$ in oPC' for reading or rollback by stages 2, 3, and 4.

speculation fails, the original values are restored from the temporary registers. If the speculation turns out to be correct, the values in the temporary registers are just ignored. By convention, we name the temporary register oQ if the name of the original register is Q.

In order to save hardware cost, *we restore specification registers only*. In particular, we do not restore the implementation registers in case of a rollback. The only justification for this is saving the gates and latches required for the rollback in case of implementation registers. The price paid for this is extra proof effort, since we have to argue that not restoring implementation registers does not affect data consistency.

Example Consider a pipelined machine and a specification register PC' that is written by stage 1 (decode). In the same cycle in that one clocks a new value into $PC'.2$, the old value of the register is saved in an *implementation* register called $oPC'.2$ (figure 5.4).

If this value is required in any later stage for rollback or any other purpose, extra instances can be added to the stages in between. This is the usual method to add instances of implementation registers, as already described in chapter 3. Note that duplicating Q into oQ is expensive regarding hardware cost. We therefore assume that Q is neither a register file nor a memory.

Remember that $\omega_k Q$ denotes the value clocked into register Q. In order to realize rollback, we change the function $\omega_k Q$ as follows: in case no rollback is made, the function returns the same value as before. In case a rollback is made, we have to select the appropriate instance of the register oQ. Note that actually more than one rollback signal can be active simultaneously. In this case, we have to take the original values from the latest stage with active rollback signal. Remember that we used $\rho(j,T)$ in order to denote that stage j has this property. We now change the new value clocked into $Q.(k+1)$ as follows:

$$\omega_k Q(c_I^T) \quad := \quad \begin{cases} f\gamma_k Q(c_I^T) & : \quad ue_k = 1 \\ oQ^T.j & : \quad \rho(j,T) \\ Q^{T-1}.(k+1) & : \quad \text{otherwise} \end{cases} \tag{5.9}$$

We implement this using multiplexers (figure 5.5). This implementation is similar to the circuit in figure 4.10 (page 116) we use in order to implement the minimum required for forwarding in chapter 4.

The implementation described here takes one cycle in order to detect misspeculation and handle the rollback. The calculation of the next configuration begins in the next cycle. In some designs, in particular in case of branch prediction, the calculation of the next configuration begins in the same cycle the misspeculation is detected. This saves one cycle but may increase cycle time. Thus, this is a CPI vs. cycle time tradeoff. However, we do not further evaluate this.

5.8 Extended Read Access Semantics

5.8.1 Specification Registers

In chapter 3, we did not allow read accesses to specification registers R in stages $k > stage(R)$. We now define semantics for such read accesses. This is not related to speculation. In fact, one can define the same semantics for the prepared sequential and pipelined machine without speculation. The only reason why we did not introduce it in one of the previous chapters is that we did not need such read accesses.

We aim to define read accesses to specification registers R in stages $k > stage(R)$ such that the claim of the input correctness lemmas still holds:

$$g_k R(c_I^T) \quad \overset{!}{=} \quad R_S^{sI(k,T)}$$

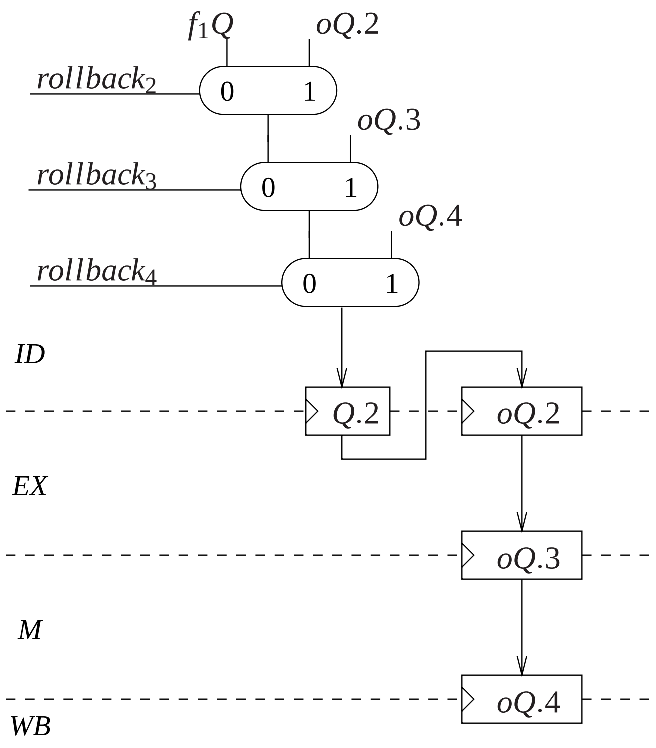


Figure 5.5 Selecting the correct value for restoring $Q.2$ in case of a rollback.

We realize this by reading the implementation register $oR.k$, as introduced above. However, we do not define such read accesses with address.

We have to prove the claim above: in order to do so, we extend the stage correctness predicates as introduced in chapter 3. The claim for registers $oR.(k+1) \in out(k)$ is:

$$oR^T.(k+1) \quad = \quad \begin{cases} R_S^{sI(k,T)-1} & : \ sI(k,T) > 0 \\ 0 & : \ \text{otherwise} \end{cases}$$

Assuming this stage correctness predicate, we easily show that inputs calculated according to the rules above are correct:

◀ Lemma 5.7

Let $full_k^T$ hold and let R be a specification register and $k > stage(R)$. Assuming that the stage correctness predicate P_{k-1} holds in cycle T, the inputs generated by the functions g_kR during cycle T are correct:

$$g_kR(c_I^T) \quad \overset{!}{=} \quad R_S^{sI(k,T)}$$

PROOF

By definition of g_kR, the value of $oR.k$ is read:

$$g_kR(c_I^T) \quad = \quad c_I^T.oR.k \tag{5.10}$$

By using the stage correctness predicate for register R, stage $k-1$, cycle T, we transform the right hand side:

$$g_kR(c_I^T) \quad = \quad R_S^{sI(k-1,T)-1} \tag{5.11}$$

According to invariant 5.3, we have $sI(k-1,T) = sI(k,T)+1$. Thus, we get:

$$g_kR(c_I^T) \quad = \quad R_S^{sI(k,T)} \tag{5.12}$$

This is the claim.

QED

5.8.2 External Signals

We further extend the read access semantics by defining *external signals*. We allow accessing external signals by adding the name of the signal to the list of registers a register transition function depends on. Let R be an external signal that is read by stage k. The signal has an arbitrary domain $\mathtt{W}\ (R)$. We assume a mapping from the instruction numbers into $\mathtt{W}\ (R)$ that defines the value of the signal in the specification machine:

$$R_S: \quad \mathbb{N} \longrightarrow \mathtt{W}\ (R)$$

Thus, the correct value of an external input R in stage k is:

$$G_k R(c_S) \quad := \quad R_S(c_S)$$

We have to assume that we get exactly the correct value if an instruction in stage k reads R. This is done if stage k is full and not stalled.

$$full_k^T \wedge \overline{stall_k^T} \quad \Longrightarrow \quad g_k R(c_I^T) = G_k R(sI(k,T))$$

Obviously, this is inconsistent if the same signal is read in multiple stages. We therefore assume that a signal is read in exactly one stage.

5.9 Branch Prediction

5.9.1 The DLX without Delayed PC

Many microprocessors do not use delayed branch semantics because of binary compatibility with earlier, sequential versions. One well-known example is the Intel x86 family [Yeu84, Int95b]. Removing the delayed PC from the specification significantly complicates a pipelined implementation.

In this section, we will give a specification of a DLX without Delayed PC. We will then use speculation as described above in order to build a pipelined DLX that provably implements this specification.

The first step is to remove the registers PC' and DPC from the specification. We add a single register PC instead. The other registers (GPR, DM) remain unchanged. As in chapter 2, let the signal I denote the instruction word fetched. The address used to fetch I is taken from the register PC and no longer from DPC:

$$I(c) \quad = \quad IM(c.PC) \tag{5.13}$$

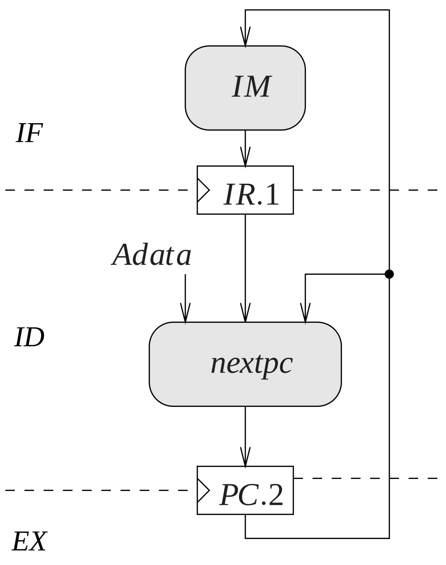


Figure 5.6 Instruction fetch and next PC calculation in a prepared sequential DLX without Delayed PC

The transition function for the register *PC* is the same as for *PC*′:

$$\delta.PC(c) \quad = \quad nextpc(I, op1(c), c.PC) \qquad (5.14)$$

The transition functions of *DM* and *GPR* remain unchanged. In case of a jump and link instruction, we take $PC+4$ and no longer $PC'+4$.

5.9.2 The Sequential DLX without Delayed PC

Implementing and verifying a prepared sequential machine without delayed PC is trivial. One takes the prepared sequential machine from chapter 3 with minimal modifications. One just renames *PC*′ into *PC* and removes the *DPC* register. The instruction fetch is made using *PC* as register (figure 5.6). No speculation is necessary. The proof of correctness follows the proof given in chapter 3.

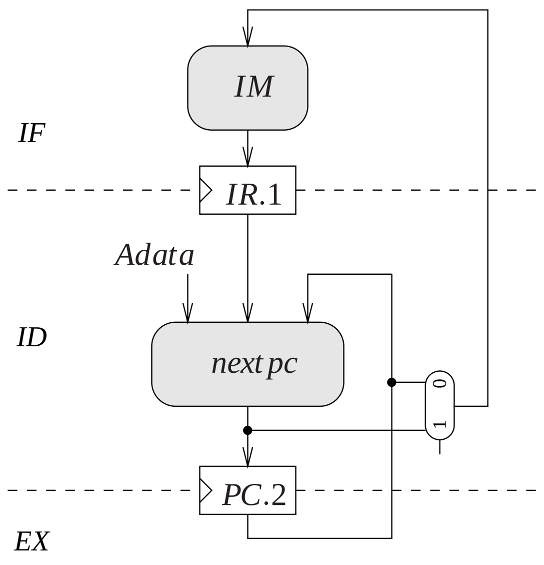


Figure 5.7 Instruction fetch and next PC calculation in a pipelined DLX without Delayed PC and without speculation

5.9.3 The Pipelined DLX without Delayed PC

In chapter 4, we transformed the prepared sequential machine with Delayed PC into a pipelined machine. This is still feasible for the machine without Delayed PC. However, we have to forward register $PC.2$ into the instruction fetch stage (figure 5.7). According to the forwarding mechanism as given in the previous chapter, we have to select between the value in the register $PC.2$ and the value written into $PC.2$ depending on the value of the full bit $full.1$.

If the decode stage is full, which is the common case, we have to use the value provided by the *nextpc* circuit as address for the instruction fetch. In particular, the *next pc* circuit uses the first GPR operand as input. This operand might be forwarded, too. We therefore get a data path that passes the ALU and the *nextpc* circuit and the instruction memory. We consider such a path to be too long.

A common approach to this problem is using *branch prediction*. The problem is the GPR operand. The GPR operand is used in order to decide whether the branch is taken or not in case of a branch instruction. In case of a jump register instruction, the operand value is used as target address.

The idea of branch prediction is to guess whether a branch is taken or not. There are various methods to realize this. Implementing branch predictors lies beyond the scope of this thesis. There is a vast amount of literature on sophisticated branch predictors, e.g. [Smi81, LS84, YP92, CHYP94, PS94].

However, branch prediction is of no use regarding jump register instructions. Since jump register instructions are much less common than branch instructions, a feasible solution is to stall the execution until the GPR operand is available. Another solution is to guess the branch target, too. This is what we implement. As for the branch predictor, we do not elaborate how to implement the target predictor.

We implement branch prediction as follows: the first step is to move register *PC* from stage 1 (decode) to stage 0 (fetch). This allows reading the *PC* register in stage 1 without any forwarding. The next PC is now calculated as follows: if the instruction fetched is neither a branch or jump, we just take the old value and increment it by four. If it is a branch, we guess whether it is taken or not. If it is a jump register instruction, we guess the branch target. We denominate these guessed values by $branch\_taken$ and $branch\_target$. We pass the address of the instruction to the predictor (figure 5.8). Figure 5.9 shows how the new PC is calculated using the guessed values.

We instantiate the rollback mechanism as described in section 5.7 (page 161). The old PC value is stored in a register $oPC.1$. This allows restoring the PC in case of a rollback (figure 5.10). Thus, the register $PC.1$ is clocked if ue_0 is active or if the rollback signal $rollback'_1$ is active. The update enable signal is used in order to select the appropriate source.

The register $oPC.1$ is also used for reading the *PC* register in stage 1 (decode stage): we read the *PC* register for jump and link instructions. Since $1 > stage(PC) = 0$, we have to use the extended read access semantics as introduced above.

As described in section 5.6 (page 159), the guessed values are stored in instances of implementation registers. In case of the DLX with branch registers these registers are $branch\_taken.1$ and $branch\_target.1$. In stage 1 (decode), we can calculate the correct values. We add a write access to the register $branch\_taken.2$. The value written is just $bjtaken\_imp$ as defined

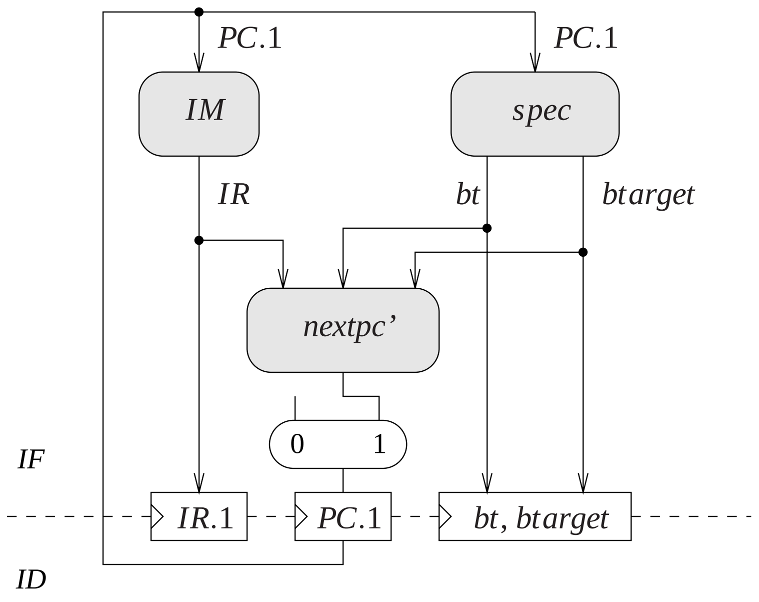


Figure 5.8 Instruction fetch and next PC calculation in a pipelined DLX without Delayed PC and with speculation. Let *bt* be a shorthand for *branch_taken* and *btarget* be a shorthand for *branch_target*. The prediction unit is denoted by *spec*. The circuits for providing the old PC value in case of a rollback are omitted.

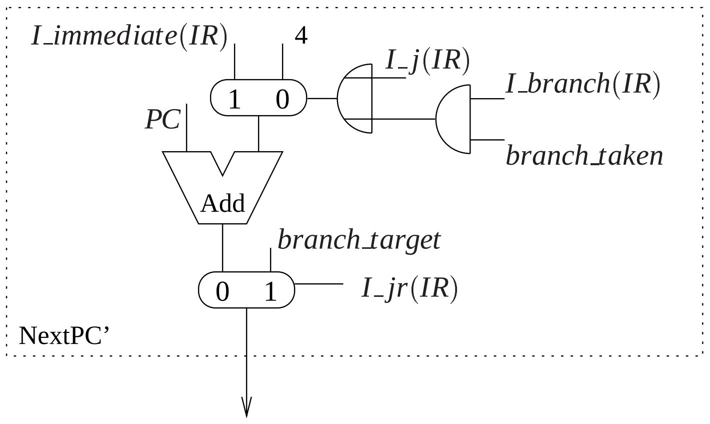


Figure 5.9 Calculating the next PC using speculation

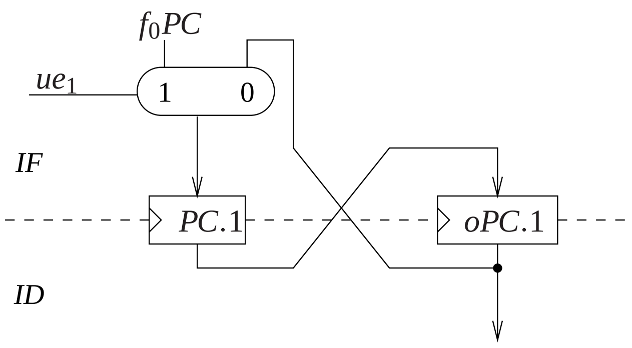


Figure 5.10 Restoring the PC register in case of a rollback: The register PC is clocked if ue_0 or $rollback'_1$ is active or in case of a reset. If ue_0 is active, the next PC is clocked into $PC.1$ and the old PC is clocked into $oPC.1$. If ue_0 is not active, the old PC from $oPC.1$ is clocked into $PC.1$. The multiplexer used in order to handle the reset case is omitted.

in the previous chapter:

$$f_1 branch\_taken(IR, GPRa) \quad = \quad bjtaken\_imp(IR, GPRa)$$

◀ Lemma 5.8

Given correct inputs, the function above calculates the correct value of *branch_taken*. We denote this correct value by $\Omega branch\_taken(c_S)$:

$$\Omega branch\_taken(c_S) \quad := \quad bjtaken(I(c_S), op1(c_S))$$

The functions $bjtaken$, I, and $op1$ are defined in chapter 2.

The claim is that $f_1 branch\_taken$ returns this value given correct inputs:

$$f\Gamma_1 branch\_taken(c_S^i) \quad = \quad \Omega branch\_taken(c_S^i)$$

PROOF

By expanding $f\Gamma_1 branch\_taken$, we get the following claim:

$$bjtaken\_imp(\Omega_0 IR(c_S^i), G_1 GPRa(c_S^i)) \quad \overset{!}{=} \quad \Omega branch\_taken(c_S^i)$$

Using lemma 3.3 (page 57), we transform this into:

$$bjtaken(\Omega_0 IR(c_S^i), G_1 GPRa(c_S^i)) \quad \overset{!}{=} \quad \Omega branch\_taken(c_S^i)$$

By definition of $\Omega branch\_taken$, the claim is equal to:

$$bjtaken(\Omega_0 IR(c_S^i), G_1 GPRa(c_S^i)) \quad \overset{!}{=} \quad bjtaken(I(c_S^i), op1(c_S^i))$$

This is concluded as in lemma 3.15 (correctness of the transition functions of the DLX without branch prediction): The first step is to assert that $\Omega_0 IR(c_S^i)$ is equal to $I(c_S^i)$. In case the instruction coded by $I(c_S^i)$ is a jump instruction, the claim immediately follows from the definition of $bjtaken$. In case of a branch instruction, one asserts that $G_1 GPRa(c_S^i)$ is equal to $op1(c_S^i)$. QED

Furthermore, we add a write access to the register $branch\_target$. The value written is $GPRa$ (the first GPR operand) if we have a jump register instruction and zero otherwise:

$$f_1 branch\_target(IR, GPRa) \quad = \quad \begin{cases} GPRa & : \quad I\_jr(IR) \\ 0 & : \quad \text{otherwise} \end{cases}$$

As above, we define a correct value for $branch\_target$. This is the GPR operand in case of a jump register instruction. In case of any other instruction, we use zero.

$$\Omega branch\_target(c_S^i) \quad = \quad \begin{cases} op1(c_S^i) & : \quad I\_jr(I(c_S^i)) \\ 0 & : \quad \text{otherwise} \end{cases}$$

Lemma 5.9 ► Given correct inputs, $f_1 branch\_target$ calculates this value:

$$f\Gamma_1 branch\_target(c_S^i) \quad = \quad \Omega branch\_target(c_S^i)$$

PROOF By expanding $f\Gamma_1 branch\_target$ and swapping left hand side and right hand side for readability, we get the following claim:

$$\Omega branch\_target(c_S^i) \quad \overset{!}{=} \quad \begin{cases} G_1 GPRa(c_S^i) & : \quad I\_jr(\Omega_0 IR(c_S^i)) \\ 0 & : \quad \text{otherwise} \end{cases}$$

One easily asserts that $\Omega_0 IR(c_S^i)$ is equal to $I(c_S^i)$. This transforms the claim into:

$$\Omega branch\_target(c_S^i) \quad \overset{!}{=} \quad \begin{cases} G_1 GPRa(c_S^i) & : \quad I\_jr(I(c_S^i)) \\ 0 & : \quad \text{otherwise} \end{cases}$$

In case $I\_jr(I(c_S^i))$ does not hold, one easily concludes the claim by expanding the definition of $\Omega branch\_target$. In case $I\_jr(I(c_S^i))$ holds, one expands $\Omega branch\_target$ and gets the following claim:

$$op1(c_S^i) \overset{!}{=} G_1GPRa(c_S^i)$$

One asserts this easily using that we have a jump register instruction. QED

5.10 Data Consistency

5.10.1 Data Consistency Criterion

The data consistency criterion for both sequential and pipelined machines is that we match values of the registers in the implementation machine with values taken from the specification machine. This no longer works in a machine with speculation. As an example, consider the *PC* register in the pipelined machine without Delayed PC. If the speculation fails, we actually write wrong values into this register. This wrong value might never occur in the specification machine.

This gets even worse if one considers a machine that detects the misspeculation in even later stages, e.g., in stage 3 or 4. In such a machine, subsequent instructions are fetched using a wrong PC. This might lead to completely undefined results. This is illustrated in figure 5.11: assume instruction I_0 does not require speculation and that we misspeculated while instruction I_1 was in stage 0. If I_1 is in stage 3, we have the following situation: in registers $R.k$ with $k > 3$, there is still correct data. In registers $R.3$, we have the misspeculated data. In registers $R.k$ with $k < 3$, we have data calculated using misspeculated data.

The last stage that contains misspeculated data is called *speculation stage*. In the example above, this is stage 3. If no misspeculation is done, this is stage 0. In analogy to the scheduling function $I(k,T)$, let $\Sigma(T)$ denote the number of this stage during cycle T.

$$\Sigma: \quad \mathbb{N}_0 \longrightarrow \{0,\dots,n-1\}$$

We now adjust our data consistency criterion as follows: we no longer claim anything for registers $R.k$ with $k < \Sigma(T)$. For registers $R.k$ with

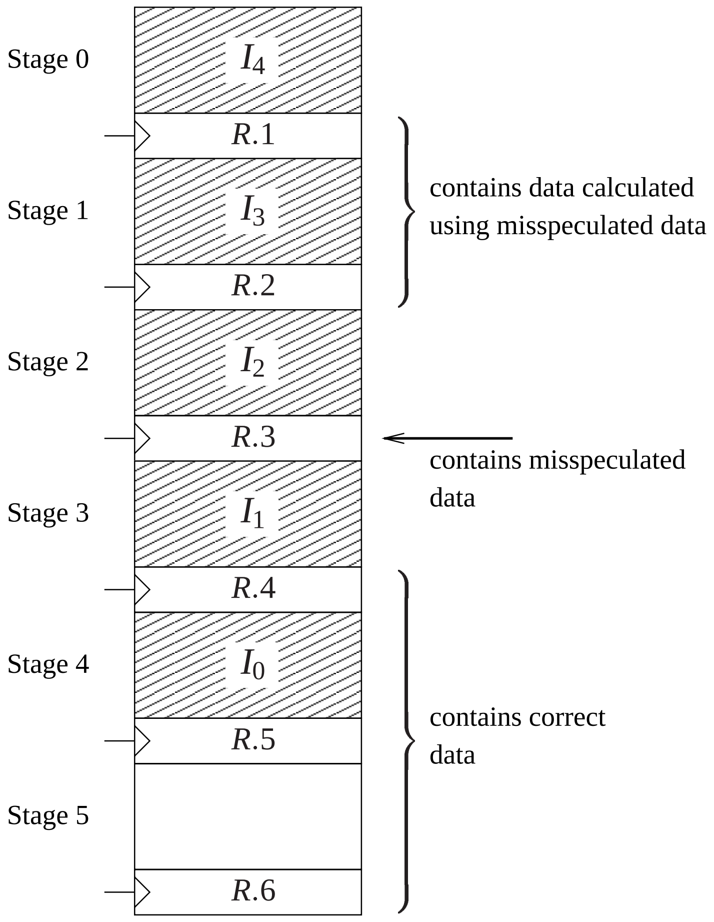


Figure 5.11 Illustration of the data consistency criterion for machines with speculation: let stage 3 be the latest stage with misspeculated data. Stages 4 and 5 contain correct data, stages 1 and 2 contain data that were calculated using misspeculated data.

$k > \Sigma(T)$, we use the very same criterion as before. For the registers $R.k$ with $k = \Sigma(T)$, we need to distinguish the registers.

Obviously, there are some registers $R.k$ that contain wrong data, in particular the implementation registers that hold the misspeculated data, i.e., the registers $R.k$ with $R \in \sigma$. But there may be more registers with wrong data, namely those that have been calculated using the misspeculated data as input. We denote the set of registers $R.(k+1)$ that is calculated using speculated data by $\sigma(k)$. Note that if one uses a speculative input $R \in \sigma$ in a stage later or equal than the misspeculation is detected, it is no longer considered a speculative input. This is motivated as follows: if the input value is used for subsequent calculations, we know it is correct. Otherwise, we make a rollback and the value is not used.

For all registers $R.k$ that are not element of $\sigma(k-1)$, we maintain the original correctness criterion. Formally, we redefine the stage correctness predicates introduced in chapter 3 as follows: the stage correctness predicate P_k no longer contains a claim about registers that are involved in speculation, i.e., those that are element of $\sigma(k)$.

As before, let $s\mathsf{P}_k(T)$ denote the stage correctness predicate for the specification registers and let $i\mathsf{P}_k(T)$ denote the stage correctness predicate for the implementation registers. The new stage correctness predicate P_k for the output registers of stage k holds if both $s\mathsf{P}_k$ and $i\mathsf{P}_k$ hold, as before:

$$\mathsf{P}_k(T) \iff s\mathsf{P}_k(T) \wedge i\mathsf{P}_k(T)$$

The stage correctness predicate $s\mathsf{P}_k(T)$ for the specification registers is the same as before but without the registers involved in speculation: Thus, for all specification registers $R \in out(k)$ and $R \notin \sigma(k)$ the following condition must hold:

$$R_I^T = R_S^{sI(k,T)}$$

Furthermore, we modify the claim for implementation registers. We have to do so because we do not restore implementation registers in case of a rollback. As described in section 5.7 (page 161), this is motivated by saving hardware cost. Let $R.(k+1)$ be an implementation register. The claim depends on the full bit $full_{k+1}^T$. If it is active, the claim for R_I^T stays the same as before. If it is not active, we just do not claim anything for R_I^T. Thus, for all implementation registers $R \in out(k)$ and $R \notin \sigma(k)$ the following condition must hold:

$$full^T_{k+1} \quad \Longrightarrow \quad R^T_I.(k+1) = \begin{cases} 0 & : \quad sI(k,T) = 0 \\ \Omega_k R(c_S^{sI(k,T)-1}) & : \quad \text{otherwise} \end{cases}$$

If $full^T_{k+1}$, we have $sI(k,T) = sI(k,T) + 1$ according to invariant 5.3. Thus, $sI(k,T) = 0$ cannot happen if $full^T_{k+1}$ holds. The condition above therefore simplifies to:

$$full^T_{k+1} \quad \Longrightarrow \quad R^T_I.(k+1) = \Omega_k R(c_S^{sI(k,T)-1})$$

In case of a rollback, the full bits of the affected stages are cleared. Thus, we no longer have to show anything for the implementation registers in those stages until new values are stored there. However, this new induction premise is weaker than the old one. We have to verify that it is still sufficient for showing that the inputs of the transition functions are correct.

One easily asserts this. The lemmas used to argue the correctness of input registers have the premise that the full bit is active (e.g., lemma 3.16, page 82, lemma 4.7, page 103).

For the registers that are calculated using speculative values, we define a separate predicate $\mathsf{P}'_k(T)$. The predicate holds iff the values in the registers that are in $\sigma(k)$ are correct. In case of specification registers, the correctness criterion is as before: we use the value provided by the specification machine. In case of implementation registers, we used to define the correct value using a definition for correct inputs. However, the implementation registers $R.(k+1) \in \sigma(k)$ depend on speculative values.

We therefore need a notion of a correct speculative value. As described above,the function $\Omega R(c_S)$ denotes the correct value of a speculative value $R \in \sigma$ given a configuration of the specification machine. Using that function, we define the correctness predicate for speculative implementation registers as before.

This also allows defining a predicate $\mathsf{S}_{-1}(T)$, which holds iff we speculate correctly during cycle T. The predicate is used for the speculation in done stage 0 only. Formally, this is done using the function $\Omega R(c_S)$. We speculate correctly iff the input generation circuit provides this value for all registers $R \in \sigma$:

$$\mathsf{S}_{-1}(T) \quad :\Longleftrightarrow \quad \forall R \in \sigma : \; g_0 R(c^T_I) = \Omega R(c_S^{sI(0,T)})$$

As described above, the guessed values are stored in implementation registers and are propagated by adding instances of these implementation registers. We therefore define a correctness predicate $\mathsf{S}_k(T)$ for those registers R with $R \in \sigma$ and $R \in out(k)$. The predicate is defined in analogy to the predicate $i\mathsf{P}_k$ for implementation registers. As for the implementation registers, we only claim anything if the full bit is active:

$$\mathsf{S}_k(T) \quad :\Longleftrightarrow \quad \left(full_{k+1}^T \Longrightarrow R_I^T.(k+1) = \Omega R(c_S^{sI(k,T)-1}) \right)$$

Examples The use of the stage correctness predicates is illustrated in figures 5.12, 5.13, and 5.14: In all figures, we summarize four classes of registers:

- By $S.k$, we summarize the non-speculative specification registers,
- by $S'.k$, we summarize the speculative specification registers,
- by $I.k$, we summarize the implementation registers that are not speculative, i.e., they are not element of $\sigma(k-1)$,
- by $I'.k$, we summarize the implementation registers that are speculative, i.e., they are element of $\sigma(k-1)$.

If the box of the register is drawn using stronger lines, this denotes that the correctness of the value in the register is claimed.

In figure 5.12, we show the transition from cycle T to cycle $T+1$ if instruction I_0 moves from stage 0 to 1 and does not misspeculate. The speculation function Σ is zero in both cycles. In cycle T, we claim the correctness of the specification registers only, i.e., of the registers $S.k$ and $S'.k$. Since no full bit is set, we do not claim anything for implementation registers. As soon as I_0 is in stage 1, the full bit $full_1$ is set. Thus, we claim the correctness of the implementation registers $I.1$ during cycle $T+1$. Since $\Sigma(T+1)$ is zero, we did not misspeculate. We therefore also claim the correctness of the speculative implementation registers $I'.1$ during cycle $T+1$. Since we always claim the correctness of the registers $S.k$, we omit those registers in later figures.

In figure 5.13, we show the case that we misspeculate. The speculation function Σ therefore is 1 in cycle $T+1$. We therefore no longer claim that the values in $S'.1$ or $I'.1$ are correct. However, we still claim that the values in $I.1$ are correct: these values do not depend on the guessed values.

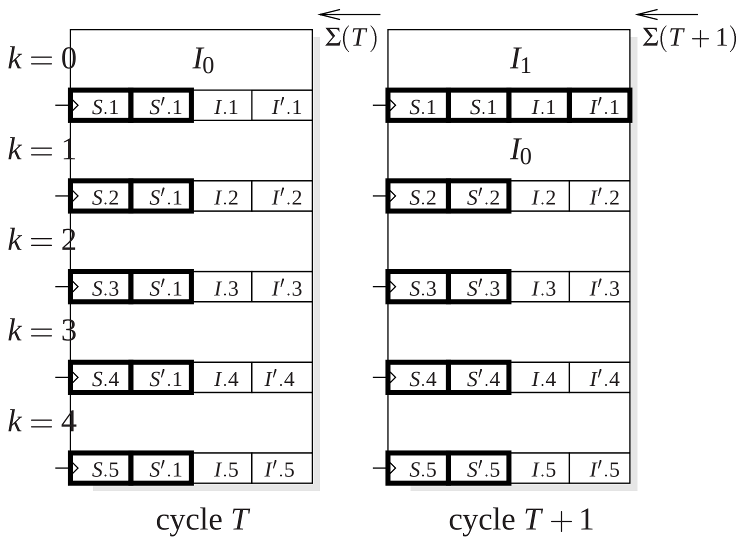


Figure 5.12 Claim of correctness in case we do not misspeculate. $\Sigma(T)$ points to the speculation stage. We claim correctness for registers drawn with stronger lines.

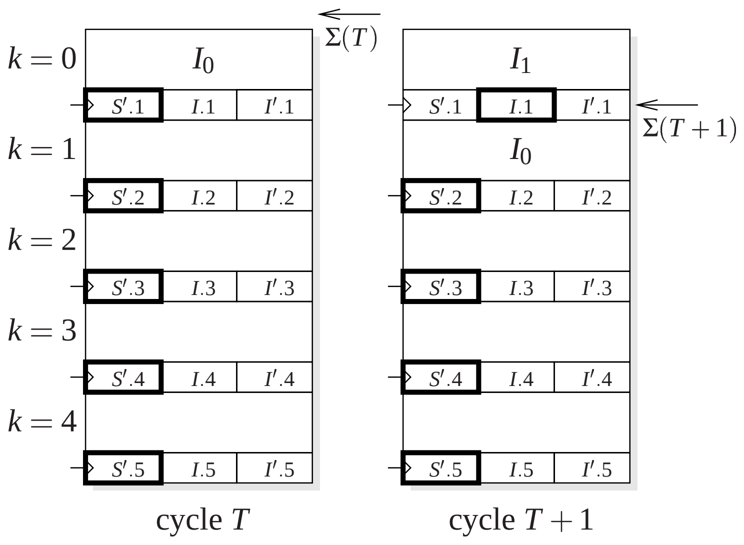


Figure 5.13 Claim of correctness in case we misspeculate

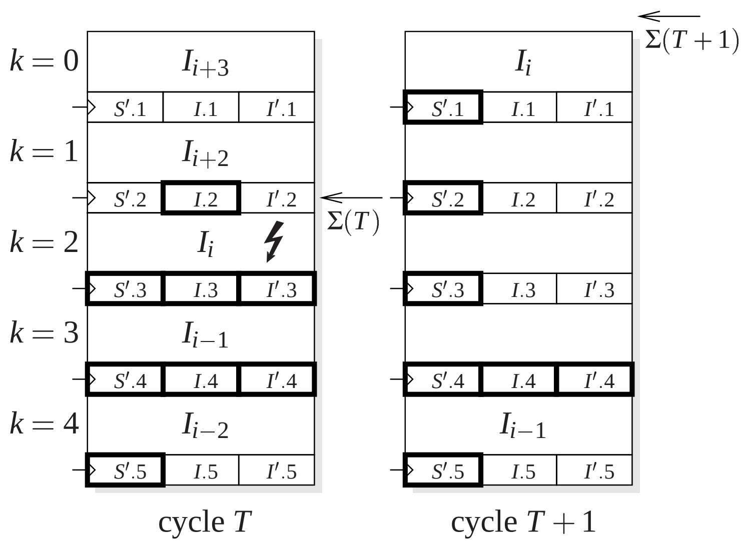


Figure 5.14 Claim of correctness in case we do a rollback. After a rollback, Σ is zero.

In figure 5.14, we show the rollback case. The speculation function Σ is 2 in cycle T and we detect the misspeculation in that cycle, as indicated by the flash symbol. Thus, the speculation function is again 0 in cycle $T+1$. As before, for cycle T and stage $\Sigma(T) = 2$, we only claim the correctness of the values in $I.2$. For all later stages, we claim the correctness of the values in $S'.k$, and $I.k/I'.k$ iff the stage is full. In the example, stages 3 and 4 are full, we therefore claim the correctness of the values in $I.3$, $I'.3$, $I.4$, and $I'.4$.

Registers of the DLX without Delayed PC As an example, consider the pipelined DLX without Delayed PC described above. We have three specification registers, which are *PC*, *GPR*, and *DM*. Of these registers, only the register *PC* depends on speculative inputs since we detect any misspeculation in stage 1. Thus, we have:

$$PC \in \sigma(0)$$

Thus, we remove the claim for *PC* from $s\mathsf{P}_0$ and add it to $s\mathsf{P}'_0$ instead:

$$s\mathsf{P}'_0(T) \quad :\Longleftrightarrow \quad PC.1^T = PC_S^{sI(k,T)}$$

We have speculative values *branch_taken* and *branch_target*. The predicate $\mathsf{S}_{-1}(T)$ therefore is:

$$\begin{aligned}\mathsf{S}_{-1}(T) \quad :\Longleftrightarrow \quad & g_0 branch\_taken(c_I^T) = \Omega branch\_taken(c_S^{sI(0,T)}) \wedge \\ & g_0 branch\_target(c_I^T) = \Omega branch\_target(c_S^{sI(0,T)})\end{aligned}$$

In $out(0)$, we have instances of the speculative values *branch_taken* and *branch_target*. The claim for those registers is in the predicate $\mathsf{S}_0(T)$:

$$\begin{aligned}&\mathsf{S}_0(T) \;:\Longleftrightarrow \\ (full_{k+1}^T \quad \Longrightarrow \quad & branch\_taken^T.1 = \Omega branch\_taken(c_S^{sI(0,T)-1}) \wedge \\ & branch\_target^T.1 = \Omega branch\_target(c_S^{sI(0,T)-1}))\end{aligned}$$

5.10.2 Properties of the Pipeline

In this section, we conclude basic data consistency properties. We start with a lemma that asserts that the machine is initialized properly:

Lemma 5.10 ▶ The predicates $\mathsf{P}_k(T)$, $\mathsf{P}'_k(T)$, and $\mathsf{S}_k(T)$ hold for cycle $T = 0$ and $k \geq 0$.

PROOF One easily asserts this lemma using that $sI(k,0) = 0$ holds.

For the following data consistency properties, we define a shorthand for the term “the inputs of stage k are correct”. In analogy to the stage correctness predicates, we use two predicates: one for inputs not affected by misspeculation, and one for inputs affected by misspeculation.

Let $\mathsf{I}_k(T)$ denote that the inputs of stage k that are not affected by misspeculation are correct during cycle T. One shows this using the input correctness lemmas. These lemmas in turn depend on certain stage correctness predicates. In order to argue the correctness of the inputs of stage k, we need the stage correctness predicates P of the stages $k-1$ and later stages. In addition to that, we can use the stage correctness predicates P' for stage k and later ones:

$$\begin{aligned}\mathsf{I}_k(T) \quad :\Longleftrightarrow \quad & (\forall l : l \geq k-1 \quad \Longrightarrow \quad \mathsf{P}_l(T)) \wedge \\ & (\forall l : l \geq k \qquad\; \Longrightarrow \quad \mathsf{P}'_l(T))\end{aligned}$$

In analogy to that, let $\mathsf{I}'_k(T)$ denote that the inputs of stage k that are affected by misspeculation are correct during cycle T. The inputs affected by misspeculation depend on the stage. In case of stage $k = 0$, we have the guessed data, i.e., we use the predicate $\mathsf{S}_{-1}(T)$. In case of stages $k > 0$, we use the predicates $\mathsf{S}_{k-1}(T)$ and $\mathsf{P}'_{k-1}(T)$:

$$\mathsf{I}'_k(T) \quad :\Longleftrightarrow \quad \begin{cases} \mathsf{S}_{k-1}(T) & : \ k = 0 \\ \mathsf{S}_{k-1}(T) \wedge \mathsf{P}'_{k-1}(T) & : \ \text{otherwise} \end{cases}$$

◀ Lemma 5.11

Let the non-speculative inputs of stage k be correct during cycle T and let the output registers of stage k be not affected by a rollback. In this case, the stage correctness predicate P_k holds during cycle $T+1$.

$$(k = n-1 \vee \overline{rollback'^{T}_{k+1}}) \wedge \mathsf{I}_k(T) \quad \Longrightarrow \quad \mathsf{P}_k(T+1)$$

◀ Lemma 5.12

Let all inputs of stage k be correct during cycle T and let the output registers of stage k be not affected by a rollback. In this case, the stage correctness predicate P'_k holds during cycle $T+1$.

$$(k = n-1 \vee \overline{rollback'^{T}_{k+1}}) \wedge \mathsf{I}_k(T) \wedge \mathsf{I}'_k(T) \quad \Longrightarrow \quad \mathsf{P}'_k(T+1)$$

One easily asserts lemmas 5.11 and 5.12 as done in the pipelined machine without speculation.

Obviously, if one combines the lemmas 5.11 and 5.12, one gets that correctness of all inputs implies the correctness of all outputs unless there is a rollback.

◀ Lemma 5.13

If the update enable signal ue_k is off and if the output registers of stage k are not affected by a rollback, all predicates that hold in cycle T also hold in cycle $T+1$:

$$\begin{aligned} \mathsf{P}_k(T) &\quad \Longrightarrow \quad \mathsf{P}_k(T+1) \\ \mathsf{P}'_k(T) &\quad \Longrightarrow \quad \mathsf{P}'_k(T+1) \\ \mathsf{S}_k(T) &\quad \Longrightarrow \quad \mathsf{S}_k(T+1) \end{aligned}$$

PROOF

The proof is trivial and uses the fact that neither the values in the registers nor the predicates change from cycle T to $T+1$.

For all machines with speculation, we assume that there is a stage in that we detect any misspeculation at the latest. For example, in the DLX without Delayed PC, we detect the branch missprediction in stage 1 (decode) at the latest. We denote the number of this stage by λ.

Lemma 5.14 ▶ If the update enable of stage λ is active, and if the non-speculative inputs of the stage are correct, we did not misspeculate, i.e., the registers $R.\lambda$ holding the propagated speculative values have correct values.

$$ue_\lambda^T \wedge \mathbb{I}_\lambda(T) \quad \Longrightarrow \quad \mathbb{S}_{\lambda-1}(T)$$

PROOF Remember that $\mathbb{S}_{\lambda-1}(T)$ is defined as follows:

$$full_\lambda^T \quad \Longrightarrow \quad R_I^T.\lambda = \Omega R(c_S^{sI(\lambda-1,T)-1})$$

Since ue_λ^T holds, $full_\lambda^T$ also holds. Thus, we have to show:

$$R_I^T.\lambda \quad \overset{!}{=} \quad \Omega R(c_S^{sI(\lambda-1,T)-1})$$

According to invariant 5.3, we have $sI(\lambda,T) = sI(\lambda-1,T)-1$. Thus, the claim is transformed into:

$$R_I^T.\lambda \quad \overset{!}{=} \quad \Omega R(c_S^{sI(\lambda,T)})$$

This lemma is shown easily using the fact that ue_λ^T implies that the rollback signal $rollback_\lambda^T$ cannot be active. Furthermore, $stall_\lambda^T$ is not active and $full_\lambda^T$ is active. Thus, the signals $R_\lambda misspec^T$ are also not active for speculative registers R. Thus, by definition of the *misspec* signal, we have:

$$f\gamma_\lambda R(c_I^T) \quad = \quad R_I^T.\lambda$$

Since the inputs are correct, we have:

$$f\Gamma_\lambda R(c_S^{sI(\lambda,T)}) \quad = \quad R_I^T.\lambda$$

This allows transforming the claim into:

$$f\Gamma_\lambda R(c_S^{sI(\lambda,T)}) \quad \overset{!}{=} \quad \Omega R(c_S^{sI(\lambda,T)})$$

For the pipelined machine with branch prediction, we have two speculative registers R, which are *branch_taken* and *branch_target*.

The claim above for *branch_taken* is concluded by lemma 5.8, and the claim for the register *branch_target* is concluded by lemma 5.9.

The following lemma asserts that the guessed data is passed correctly from one stage to the next if the update enable signal is active. Note that this includes the case that the guessed data is wrong.

◀ Lemma 5.15

If the update enable signal of stage k is active and if the non-speculative inputs of stage k are correct, the predicates $\mathsf{S}_k(T+1)$ holds iff the predicate $\mathsf{S}_{k-1}(T)$ holds.

$$ue_k^T \wedge \mathsf{I}_k(T) \quad \Longrightarrow \quad \mathsf{S}_k(T+1) = \mathsf{S}_{k-1}(T)$$

One easily asserts this lemma by expanding the predicates S.

◀ Lemma 5.16

Let there be a rollback in stage $k \geq 1$ and cycle T. If the values in the non-speculative output registers of stage $k-1$ are correct, we do the rollback correctly, i.e., the correctness predicates for the output registers of stages $l < k$ hold during cycle $T+1$.

$$\begin{aligned} & \rho(k,T) \wedge \mathsf{P}_{k-1}(T) \\ \Longrightarrow \quad & \forall 0 \leq l < k : \mathsf{P}_l(T+1) \wedge \mathsf{P}'_l(T+1) \wedge \mathsf{S}_l(T+1) \end{aligned}$$

PROOF

One easily asserts this by expanding the predicates. For implementation registers, we do not have to show anything, because the rollback clears the full bits. For specification registers, we use the predicate $\mathsf{P}_{k-1}(T)$.

◀ Lemma 5.17

Let stage k be full during cycles T and $T+1$. If the update enable signal of the stage is not active and if the output registers of stage k are not affected by a rollback, the predicate $\mathsf{S}_k(T)$ is equal to $\mathsf{S}_k(T+1)$.

$$\begin{aligned} & \overline{ue_k^T} \wedge (k = N-1 \vee \overline{rollback'^T_{k+1}}) \wedge full_{k+1}^T \wedge full_{k+1}^{T+1} \\ \Longrightarrow \quad & \mathsf{S}_k(T) = \mathsf{S}_k(T+1) \end{aligned}$$

PROOF

One easily shows this by arguing as follows: Since the update enable signal is off and there is no rollback, the values of the registers do not change. The claims of the predicates do not change either, since the full bit is active in both cycles.

◀ Lemma 5.18

If a $rollback'_k$ signal is active, there is a stage j such that this is the last stage with active $rollback'$ signal.

$$rollback'^T_k \quad \Longrightarrow \quad \exists j \geq k : \rho(j,T)$$

PROOF One easily shows this lemma using induction. One starts with stage k and proceeds from k to $k+1$ until one either reaches the end of the pipeline or a stage without active $rollback'$ signal.

5.10.3 Data Consistency Invariants

We introduced the speculation stage function $\Sigma(T)$ above without giving a definition. In analogy to $sI(k,T)$, we now give a recursive definition of $\Sigma(T)$. The recursive definition is constructed as follows: During cycle $T=0$, we obviously have $\Sigma(T)=0$, i.e., no instruction with misspeculated data is in the pipeline.

The definition of $\Sigma(T)$ for $T>0$ is constructed as follows: we consider the stage that $\Sigma(T-1)$ points to. There are three cases:

1. If the update enable signal of the stage is not active and if the stage is not affected by a rollback, the value of the speculation stage function must stay the same, i.e., $\Sigma(T)=\Sigma(T-1)$.

2. If the update enable signal of the stage is active, the instruction in the stage moves into the next stage. In case of stage $\Sigma(T-1)=0$, we need to distinguish whether we misspeculated or not. We misspeculated iff $\mathsf{S}_{-1}(T-1)$ does not hold. If we misspeculated, $\Sigma(T)$ must be one. If not so, $\Sigma(T)$ remains zero.

   In case of stage $\Sigma(T-1)>0$, we already know that we misspeculated, i.e., $\Sigma(T)=\Sigma(T-1)+1$.

   In addition to that, we define an upper bound for $\Sigma(T)$ that is λ. In case $\Sigma(T-1)$ is greater or equal than λ, we define $\Sigma(T)$ to be zero.

3. In case of a rollback, as indicated by $rollback'^{T-1}_{\Sigma(T-1)}$, the speculation stage function becomes zero.

Thus, we define $\Sigma(T)$ for $T>0$ as follows: If $ue^{T-1}_{\Sigma(T-1)}$ holds, $\Sigma(T)$ is:

$$\Sigma(T) \quad := \quad \begin{cases} 1 & : \quad \Sigma(T-1)=0 \wedge \overline{\mathsf{S}_{-1}(T-1)} \\ \Sigma(T-1)+1 & : \quad 0<\Sigma(T-1)<\lambda \\ 0 & : \quad \text{otherwise} \end{cases}$$

If $ue^{T-1}_{\Sigma(T-1)}$ does not hold, $\Sigma(T)$ is:

$$\Sigma(T) \quad := \quad \begin{cases} 0 & : \; rollback'^{\,T-1}_{\Sigma(T-1)} \\ \Sigma(T-1) & : \; \text{otherwise} \end{cases}$$

Using this recursive definition for $\Sigma(T)$, we conclude several properties.

◀ Lemma 5.19

The latest stage we detect misspeculation in, i.e., stage λ, is an upper bound for the speculation stage function.

$$\Sigma(T) \quad \leq \quad \lambda$$

PROOF

One easily shows this using the definition of the speculation stage function above.

◀ Lemma 5.20

The stage $\Sigma(T)$ is full during cycle T, i.e., there is an instruction in the last stage containing misspeculated data.

$$full^T_{\Sigma(T)}$$

PROOF

The claim is shown using induction on T. For $T = 0$, one easily asserts the claim using that $\Sigma(T)$ is zero and that $full_0$ is always active.

For $T + 1$, we show the claim by a case split on the value of $ue^T_{\Sigma(T)}$.

- If $ue^T_{\Sigma(T)}$ holds, we have to show either $full^{T+1}_{\Sigma(T)+1}$ or $full^{T+1}_0$. The later claim holds because $full_0$ is always active. We show $full^{T+1}_{\Sigma(T)+1}$ by applying lemma 4.1:

  $$full^{T+1}_{\Sigma(T)+1} \quad = \quad (ue^T_{\Sigma(T)} \vee stall^T_{\Sigma(T)+1}) \wedge \overline{rollback'^{\,T}_{\Sigma(T)+1}}$$

  Since $ue^T_{\Sigma(T)}$ holds, the $rollback'$ signal cannot be active and we get that $full^{T+1}_{\Sigma(T)+1}$ holds.

- If $ue^T_{\Sigma(T)}$ does not hold and we have a rollback, we have to show that $full^{T+1}_0$, which one easily asserts as above.

  If $ue^T_{\Sigma(T)}$ does not hold and we do not have a rollback, we have to show that $full^{T+1}_{\Sigma(T)}$ holds. This follows directly form lemma 4.4 for cycle T and stage $\Sigma(T)$.

QED

The following lemma is easily concluded from lemma 5.13 and lemma 5.15:

Lemma 5.21 ▶ If we have correct inputs (both speculative and non-speculative) during cycle T, and the values in the speculative output registers of stage k are correct during cycle T, the values are also correct during cycle $T+1$.

$$(k = n-1 \lor \overline{rollback'^{T}_{k+1}}) \land \mathsf{I}'_k(T) \land \mathsf{I}_k(T) \land \mathsf{S}_k(T) \implies \mathsf{S}_k(T+1)$$

We now claim two speculation invariants. We will later on show these invariants using induction.

Invariant 5.4 ▶ If $\Sigma(T)$ is not zero, at least one speculative register of the output registers of stage $\Sigma(T)-1$ has wrong values:

$$\Sigma(T) \geq 1 \implies \overline{\mathsf{S}^{T}_{\Sigma(T)-1}}$$

We will later on use this invariant in order to claim that we actually are able to detect misspeculation. The following invariant is the data consistency claim as introduced above:

Invariant 5.5 ▶ The data consistency predicates of all registers that are outputs of stages $k \geq \Sigma(T)$ hold during cycle T. In addition to that, the predicate for the non-speculative registers that are output registers of stage $\Sigma(T)-1$ holds.

$$\begin{aligned} k \geq \Sigma(T)-1 &\implies \mathsf{P}_k(T) \\ k \geq \Sigma(T) &\implies \mathsf{P}'_k(T) \\ k \geq \Sigma(T) &\implies \mathsf{S}_k(T) \end{aligned}$$

One easily asserts the following two claims by expanding the predicates:

Lemma 5.22 ▶ Let invariant 5.5 hold for cycle T. For all stages $k \geq \Sigma(T)$, the non-speculative inputs of stage k are correct.

$$k \geq \Sigma(T) \implies \mathsf{I}_k(T)$$

Lemma 5.23 ▶ Let invariant 5.5 hold for cycle T. For stages $k > \Sigma(T)$, the speculative inputs of stage k are correct.

$$k > \Sigma(T) \implies \mathsf{I}'_k(T)$$

The following lemma will be used as the induction step for showing invariant 5.4:

◄ Lemma 5.24

Let both speculation invariants hold during cycle T. This implies that invariant 5.4 holds during cycle $T+1$.

PROOF

We do a case split on the values of the update enable signal $ue^{T}_{\Sigma(T)}$ and $rollback'^{T}_{\Sigma(T)}$.

If $ue^{T}_{\Sigma(T)}$ holds, there are three cases for $\Sigma(T)$:

1. Let $\Sigma(T)$ be zero. If $\mathsf{S}_{-1}(T)$ holds, $\Sigma(T+1)$ is also zero and we have nothing to show.

   Thus, let $\mathsf{S}_{-1}(T)$ not hold. In this case, we have $\Sigma(T+1) = 1$ and we therefore have to disprove $\mathsf{S}_{0}(T+1)$. This is easily done using lemma 5.15 and lemma 5.22.

2. Let $\Sigma(T) \geq \lambda$ hold. In this case, $\Sigma(T+1)$ is zero and we have nothing to show.

3. Let $0 < \Sigma(T) < \lambda$ hold. In this case, we have $\Sigma(T+1) = \Sigma(T)+1$. We have to disprove $\mathsf{S}_{\Sigma(T)+1}(T+1)$. As before, this is done using lemma 5.15 and lemma 5.22.

Let $ue^{T}_{\Sigma(T)}$ not hold and let $rollback'^{T}_{\Sigma(T)}$ hold. In this case, $\Sigma(T+1)$ is zero and we have nothing to show.

Let both $ue^{T}_{\Sigma(T)}$ and $rollback'^{T}_{\Sigma(T)}$ not hold. In this case, $\Sigma(T+1)$ is equal to $\Sigma(T)$. We have to disprove $\mathsf{S}_{\Sigma(T)-1}(T+1)$. According to lemma 5.17 for stage $\Sigma(T)-1$, we have:

$$\mathsf{S}_{\Sigma(T)-1}(T+1) \quad = \quad \mathsf{S}_{\Sigma(T)-1}(T)$$

The right hand side does not hold because of the induction premise. However, we have to prove the premises of lemma 5.17: We disprove $ue^{T}_{\Sigma(T)-1}$ as follows: according to lemma 5.20, stage $\Sigma(T)$ is full during cycle T. Since $ue^{T}_{\Sigma(T)}$ is not active, we would overwrite the contents of stage $\Sigma(T)$. This is not possible, as asserted by lemma 4.3.

In addition to that, we have to show that both $full^{T}_{\Sigma(T)}$ and $full^{T+1}_{\Sigma(T)}$ hold, which is easily done using lemma 5.20.

QED

The following lemma will be used as induction step for the case that ue_k^T does not hold.

Lemma 5.25 ► Let the speculation invariants hold in cycle T. Let the update enable signal ue_k^T be not active. This implies that $\mathsf{S}_k(T+1)$ and $\mathsf{P}'_k(T+1)$ hold if $k \geq \Sigma(T)$ and that $\mathsf{P}_k(T+1)$ holds if $k \geq \Sigma(T) - 1$:

$$\begin{array}{rcl} k \geq \Sigma(T) - 1 & \Longrightarrow & \mathsf{P}_k(T+1) \\ k \geq \Sigma(T) & \Longrightarrow & \mathsf{P}'_k(T+1) \\ k \geq \Sigma(T) & \Longrightarrow & \mathsf{S}_k(T+1)) \end{array}$$

Note that the claim of this lemma is not identical with speculation invariant 5.5. On the left hand side, we have $\Sigma(T)$ and on the right hand side, we have the predicates for cycle $T+1$.

PROOF If the output registers of stage k are affected by a rollback, we conclude that there is a last stage $l \geq k+1$ with active rollback signal (lemma 5.18). We then use lemma 5.16 in order to conclude the claim.

If the output registers of stage k are not affected by a rollback, we use lemma 5.12 in order to conclude the claim. QED

Lemma 5.26 ► Let the speculation invariants hold in cycle T. This implies that $\mathsf{S}'_k(T+1)$ and $\mathsf{P}'_k(T+1)$ hold if $k \geq \Sigma(T)+1$ and that $\mathsf{P}_k(T+1)$ if $k \geq \Sigma(T)$:

$$\begin{array}{rcl} k \geq \Sigma(T) & \Longrightarrow & \mathsf{P}_k(T+1) \\ k \geq \Sigma(T)+1 & \Longrightarrow & \mathsf{P}'_k(T+1) \\ k \geq \Sigma(T)+1 & \Longrightarrow & \mathsf{S}_k(T+1) \end{array}$$

PROOF If ue_k^T does not hold, we use lemma 5.25 in order to conclude the claim.

Thus, let ue_k^T hold. This implies that the output registers of stage k are not affected by a rollback. We conclude $\mathsf{P}_k(T+1)$ using the lemma 5.22 (non-speculative inputs correct) and lemma 5.11. We conclude $\mathsf{P}'_k(T+1)$ using lemma 5.22 and lemma 5.23 (speculative inputs correct), and lemma 5.12. We conclude $\mathsf{S}_k(T+1)$ using lemma 5.15. QED

The following lemmas are the induction step for showing invariant 5.5. For sake of simplicity, we case-split using the values of the update enable and rollback signals. Lemma 5.27 shows the claim if $ue^{T}_{\Sigma(T)}$, lemma 5.28 shows the claim if $rollback'^{T}_{\Sigma(T)}$ is active, and lemma 5.29 shows the claim if neither signal is active.

◄ Lemma 5.27

Let both speculation invariants hold during cycle T and let the update enable signal $ue^{T}_{\Sigma(T)}$ be active. This implies that invariant 5.5 holds during cycle $T+1$.

PROOF

We do a case split on $\Sigma(T)$ and on $\Sigma(T+1)$.

1. If both $\Sigma(T)$ and $\Sigma(T+1)$ are zero, we conclude the claim as follows: we conclude $\mathsf{P}_k(T+1)$ for $k \geq 0$ using lemma 5.22 (inputs correct) and lemma 5.11.

   We conclude $\mathsf{P}'_k(T+1)$ for $k > 0$ using lemma 5.23, 5.22 (inputs correct) and lemma 5.12. For $k = 0$, we conclude S_{-1} using the fact that $\Sigma(T+1)$ is zero. We can then apply lemma 5.12.

   We conclude $\mathsf{S}_k(T+1)$ for $k \geq 0$ using lemma 5.21. The premises of this lemma are shown as before.

2. Let $\Sigma(T)$ be λ and $\Sigma(T+1)$ be zero. In this case, we conclude the claim using lemma 5.22 (inputs correct) and lemma 5.14.

3. Let $\Sigma(T) > \lambda$ hold. This is disproved using lemma 5.19.

4. The case $0 < \Sigma(T) < \lambda$ and $\Sigma(T+1) = 0$ is a contradiction to the definition of Σ.

5. Let $\Sigma(T+1)$ be not zero. In this case, $\Sigma(T+1) = \Sigma(T)+1$ must hold because of the active update enable signal. Because of $\Sigma(T+1) = \Sigma(T)+1$, we can conclude the claim using lemma 5.26.

This concludes the claim.

QED

◄ Lemma 5.28

Let both speculation invariants hold during cycle T and let the update enable signal $ue^{T}_{\Sigma(T)}$ be not active and let the rollback signal $rollback'^{T}_{\Sigma(T)}$ be active. This implies that invariant 5.5 holds during cycle $T+1$.

PROOF

Since $rollback'^{T}_{\Sigma(T)}$ holds, we have $\Sigma(T+1) = 0$. Thus, we have to show all three predicates for all $k \geq 0$.

Since we have an active rollback signal, there is a stage $j \geq \Sigma(T)$ that signaled the rollback, as asserted by lemma 5.18. There are three cases regarding the value of j:

1. Let $k = j$ hold. In this case, we have the stage in which the rollback is detected. The output registers of this stage are not updated, i.e., ue_k^T does not hold. We therefore are able to apply lemma 5.13, which shows the claim.

2. Let $k > j$ hold. In this case, we conclude the claim using lemma 5.26. This is feasible because of $j \geq \Sigma(T)$.

3. Let $k < j$ hold. In this case, the output registers of stage k are affected by the rollback and claim follows from lemma 5.16.

QED

Lemma 5.29 ► Let both speculation invariants hold during cycle T and let the update enable signal $ue_{\Sigma(T)}^T$ and the rollback signal $rollback'^T_{\Sigma(T)}$ be not active. This implies that invariant 5.5 holds during cycle $T+1$.

PROOF Since the update enable signal $ue_{\Sigma(T)}^T$ and the rollback signal $rollback'^T_{\Sigma(T)}$ are not active, we have $\Sigma(T+1) = \Sigma(T)$.

If ue_k^T does not hold, we conclude the claim using lemma 5.25.

If ue_k^T holds, we obviously have $k \neq \Sigma(T)$. For $k > \Sigma(T)$, we conclude the claim using lemma 5.26. For $k < \Sigma(T) - 1$, there is nothing to show. For $k = \Sigma(T) - 1$, we argue that $ue_{\Sigma(T)-1}^T$ cannot hold. According to lemma 5.20, $full_{\Sigma(T)}^T$ holds. This is a contradiction to lemma 5.2.

QED

Lemma 5.30 ► Both speculation invariants hold.

Note that speculation invariant 5.2 implies the data consistency of the specification registers.

PROOF We show this by induction on T. For cycle $T = 0$, one easily concludes the claim using that $\Sigma(0) = 0$ and using lemma 5.10. The claim for $T+1$ is concluded from the claim for cycle T using the lemmas 5.24, 5.27, 5.28, and 5.29.

5.11 Liveness

5.11.1 Liveness Proof Strategy

As for the pipelined machine without speculation, we desire to prove that the pipelined machine with speculation is alive. We maintain the very same liveness criterion as we used for the prepared sequential and pipelined machine without speculation.

Unfortunately, we cannot repeat the liveness arguments of the pipelined machine without speculation. This arises from the fact that the machine with speculation restarts instructions in case of misspeculation. We will therefore have to argue that this does not cause an infinite loop of rollbacks.

Informally, we argue as follows: in case there is no rollback, the execution proceeds as in the machine without speculation. In case there is a rollback, we argue that we will not misspeculate on the same instruction twice. However, this only holds for rollbacks in the speculation stage (lemma 5.16). For rollbacks in earlier stages, we cannot make any claim. We therefore only consider the latest stage that is full. In case there is a rollback in the latest stage that is full, we can claim that this must be the speculation stage Σ in case of a rollback.

Formally, we define a function $M(T)$ that maps a cycle T to the number of the latest full stage:

$$M(T) \quad := \quad \max\{k \mid full_k^T\}$$

In order to show liveness, we have to show that for all instructions i and stages k there is a cycle T such that $sI(k,T) = i$ holds. We consider the instruction in stage $M(T)$. Let this be instruction i. We will show that this instruction will eventually arrive in the last stage using the arguments above. We will then conclude that instruction i must have been in all stages below at least once, which satisfies our claim.

After that, we argue that the instruction in the last stage will eventually leave the pipeline. After that, there must be a stage such that instruction $i+1$ is the last stage in the pipeline. This is the first stage in the worst case. We can now repeat the arguments made for instruction i for instruction $i+1$ and so on.

This proof strategy is illustrated in figure 5.15: in cycle T_1, we have instruction I_i in stage $k=2$. This is also the latest full stage, i.e., $M(T_1)=2$.

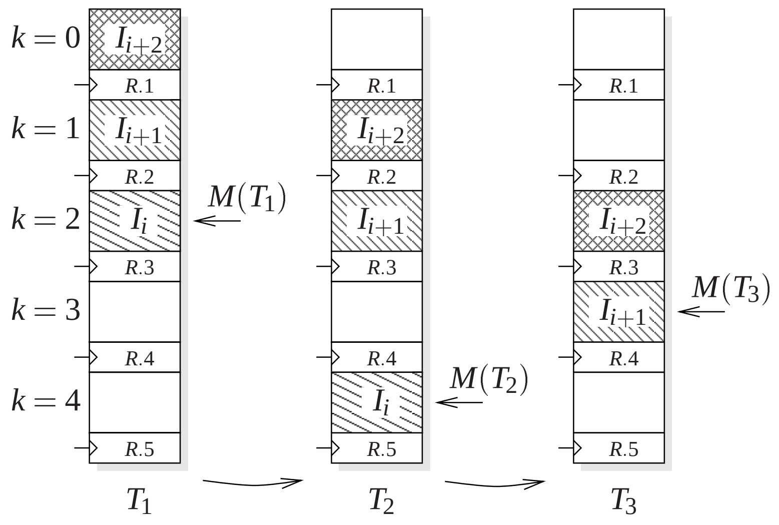


Figure 5.15 Illustration of the liveness proof strategy for machines with speculation. $M(T)$ points to the latest full stage.

This instruction will eventually arrive in the last stage. Let this be true in cycle T_2. The instruction will eventually leave the pipeline. Let this be true in cycle T_3. Then there is a stage such that instruction I_{i+1} is in the last full stage. In the example, this is stage $k = 3$.

We will now formalize this proof.

5.11.2 Properties of M(T)

We conclude a set of trivial lemmas from the definition of $M(T)$:

Lemma 5.31 ▶ This maximum exists for all T.

PROOF One easily concludes this using that $full_0^T$ is active for all T by definition of the signal.

Lemma 5.32 ▶ For $T = 0$, $M(T)$ is zero:

$$M(0) \quad = \quad 0$$

One easily asserts this by the definition of the initial values of the full bits.

◀ Lemma 5.33

Stage $M(T)$ is full during cycle T, which one concludes by definition of max:

$$full_{M(T)}^T$$

◀ Lemma 5.34

All stages below stage $M(T)$ are not full.

$$k > M(T) \quad\Longrightarrow\quad \overline{full_k^T}$$

◀ Lemma 5.35

The predicate $below\_empty_k(T)$ (page 138) with $k = M(T)$ holds for all cycles T.

$$\forall T : \qquad below\_empty_{M(T)}(T)$$

One easily concludes this by using lemma 5.34 and the definition of $below\_empty$.

◀ Lemma 5.36

A stage is the latest full stage iff the stage is full and all stages below are empty.

$$M(T) = k \quad\Longleftrightarrow\quad full_k^T \wedge below\_empty_k(T)$$

◀ Lemma 5.37

In case $rollback'^T_{M(T)}$ holds, $M(T+1)$ is zero. If $ue_{M(T)}$ is active and $M(T)$ is the last stage, we do not claim anything. If $ue_{M(T)}$ is active and $M(T)$ is not the last stage, we claim that $M(T+1)$ is $M(T)+1$. In any other case, we claim that $M(T+1)$ is equal to $M(T)$.

$$\overline{ue_{M(T)}^T \wedge M(T) = n-1}$$

$$\Longrightarrow M(T+1) = \begin{cases} 0 & : \quad rollback'^T_{M(T)} \\ M(T)+1 & : \quad ue^T_{M(T)} \\ M(T) & : \quad \text{otherwise} \end{cases}$$

PROOF

We do a case split on the values of $rollback'^T_{M(T)}$ and $ue^T_{M(T)}$.

1. Let $ue^T_{M(T)}$ hold. Thus, we only have to consider $M(T) \neq n-1$. In this case, we claim that

$$M(T+1) \quad \overset{!}{=} \quad M(T)$$

holds.

We instantiate lemma 5.36 with cycle $T+1$ and stage $M(T)+1$. This is:

$$\begin{array}{rl} & M(T+1) = M(T)+1 \\ \iff & full^{T+1}_{M(T)+1} \wedge below\_empty_{M(T)+1}(T+1) \end{array}$$

Thus, the claim holds iff the right hand side of the equivalence above holds. We show $full^{T+1}_{M(T)+1}$ using lemma 4.1 (full bit transition function):

$$full^{T+1}_{M(T)+1} \quad = \quad (ue^T_{M(T)} \vee stall^T_{M(T)+1}) \wedge \overline{rollback'^T_{M(T)+1}}$$

As $ue^T_{M(T)}$ holds, this simplifies to:

$$full^{T+1}_{M(T)+1} \quad = \quad \overline{rollback'^T_{M(T)+1}}$$

We conclude that this rollback signal cannot be active since $ue^T_{M(T)}$ is active.

It is left to show that $below\_empty_{M(T)+1}(T+1)$ holds, i.e., that all stages below stage $M(T)+1$ are empty during cycle $T+1$:

$$\forall j \mid j > M(T)+1: \quad \overline{full^{T+1}_j}$$

We apply lemma 4.1, which replaces $\overline{full^{T+1}_j}$:

$$\forall j \mid j > M(T)+1: \quad \overline{ue^T_{j-1} \vee stall^T_j) \wedge \overline{rollback'^T_j}}$$

This simplifies to:

$$\forall j \mid j > M(T)+1: \quad (\overline{ue^T_{j-1}} \wedge \overline{stall^T_j}) \vee rollback'^T_j$$

We disprove ue^T_{j-1} using that stage $j-1$ is not full during cycle T. We disprove $stall^T_j$ using that stage j is not full during cycle T. This concludes the claim.

2. Let $rollback'^{T}_{M(T)}$ hold. In this case, we claim that
$$M(T+1) \overset{!}{=} 0$$
holds, i.e., we have to show that stage 0 is full and all stages below are empty. One easily asserts this by using the fact that the rollback clears all full bits $full_j^{T+1}$ with $0 < j \leq M(T)$ and that $full_0^{T+1}$ holds by definition.

3. Let both $rollback'^{T}_{M(T)}$ and $ue^{T}_{M(T)}$ not hold. In this case, we claim that
$$M(T+1) \overset{!}{=} M(T)$$
holds, i.e., that the number of the last full stage does not change from cycle T to cycle $T+1$. We use lemma 5.36 with cycle $T+1$ and stage $M(T)$. This is:
$$\begin{aligned} & M(T+1) = M(T) \\ \iff \quad & full^{T+1}_{M(T)} \wedge below\_empty_{M(T)}(T+1) \end{aligned}$$
Thus, the claim holds iff the right hand side of this equivalence holds. For $M(T) = 0$, this claim holds by definition of the full signal. For $M(T) > 0$, we show $full^{T+1}_{M(T)}$ using lemma 5.3 (full bits do not get lost) for stage $M(T)$ and cycle T:
$$(full^{T}_{M(T)} \wedge \overline{ue^{T}_{M(T)}} \wedge \overline{rollback'^{T}_{M(T)}} \quad \Longrightarrow \quad full^{T+1}_{M(T)}$$
One easily concludes that $full^{T}_{M(T)}$ holds by lemma 5.33.

It is left to show that $below\_empty_{M(T)}(T+1)$ holds, i.e., that all stages below stage $M(T)$ are empty during cycle $T+1$:
$$\forall j \mid j > M(T) : \quad \overline{full_j^{T+1}}$$
One asserts this using the transition function as above. QED

5.11.3 Rollback Properties

◄ Lemma 5.38

The last stage with active $rollback'$ signal is full and not stalled. Remember that we used $\rho(k,T)$ as a shorthand for the fact that stage k is the last stage with active rollback signal.
$$\rho(k,T) \quad \Longrightarrow \quad full_k^T \wedge \overline{stall_k^T}$$

PROOF One easily concludes this from the definition of the *rollback* signals. The *misspec* signals are only active if the stage is full and not stalled. Since we have the last full stage, there cannot be an instruction below that causes a rollback.

Lemma 5.39 ► If we do a rollback in stage $M(T)$ during cycle T, this is the last stage with active *rollback′* signal.

$$rollback'^{T}_{M(T)} \implies \rho(M(T),T)$$

One easily concludes this using the fact that all stages below stage $M(T)$ are empty.

We have to argue that we only do a rollback in case of a misspeculation. Furthermore, we have to argue that the correct value is saved in case of a misspeculation. We do this using the following two lemmas:

Lemma 5.40 ► Let $T > 0$ be a cycle and let stage k be the last stage with active *rollback′* signal during cycle $T-1$. This implies that the values of the scheduling function for stages $l \leq k$ during cycle T match the number of the instruction in stage k during cycle $T-1$.

$$T > 0 \wedge \rho(k,T-1) \wedge l \leq k \implies sI(l,T) = sI(k,T-1)$$

PROOF We prove this claim using induction on l. The induction starts with $l = k$ and proceeds from l to $l-1$.

For $l = k$, we conclude the claim by expanding the definition of $sI(k,T)$.

For $l-1$, we have the following claim:

$$sI(l-1,T) \overset{!}{=} sI(k,T-1)$$

According to the induction premise, we have:

$$sI(l,T) = sI(k,T-1)$$

This allows transforming the claim into:

$$sI(l-1,T) \overset{!}{=} sI(l,T)$$

One easily asserts that $rollback'^{T-1}_{l}$ holds using the definition of the signal. After that, one expands the definition of $sI(l-1,T)$ on the left hand side. This concludes the claim.

QED

The following lemma argues about the correctness of the rollback mechanism. Given correct non-speculative inputs, we claim that we only rollback in case of misspeculation. Furthermore, we claim that we correctly restore the registers destroyed by the misspeculation.

◀ Lemma 5.41

If stage k is the last stage with active $rollback'$ signal during cycle T and if the non-speculative inputs of this stage are correct, we have two claims: a) we misspeculated, and b) the correct values are in the speculative registers in cycle $T+1$.

$$\rho(k,T) \wedge \mathrm{I}_k(T) \implies \overline{\mathrm{S}_{k-1}(T)} \wedge \mathrm{S}_{-1}(T+1)$$

PROOF

We show that we misspeculated as follows: because of rollback in stage k, at least one $misspec_k$ signal must be active. Let $R_k misspec$ be that signal. Thus, we have:

$$f\gamma_k R(c_I^T) \quad \neq \quad R.k^T$$

Since $f_k R$ does not depend on inputs that are speculative by definition, we can argue that $f_k R$ gets correct inputs. For example, in the DLX with branch prediction, these functions use IR and $GPRa$ as inputs. These registers are not calculated using the guessed values. Thus, we have:

$$f\Gamma_k R(c_S^{sI(k,T)}) \quad \neq \quad R.k^T$$

Using the correctness of $f_k R$ (lemmas 5.8 and 5.9 for the DLX with branch prediction), we get that the correct value of R is different from the value in $R.k$ during cycle T:

$$\Omega R(c_S^{sI(k,T)}) \quad \neq \quad R.k^T$$

Since we have a rollback, we can conclude that stage k is full using lemma 5.38. This allows applying invariant 3.3, which transforms this into:

$$\Omega R(c_S^{sI(k-1,T)-1}) \quad \neq \quad R.k^T$$

Because $full_k^T$ holds, this implies that $\mathrm{S}_{k-1}(T)$ does not hold (compare the definition of S as given in section 5.10.1). This concludes the first claim.

We show that we store the correct values in the speculative registers as follows: By definition of $\mathsf{S}_{-1}(T+1)$, we have to show for the speculative values R:

$$g_0R(c_I^{T+1}) \quad \overset{!}{=} \quad \Omega R(c_S^{sI(0,T+1)})$$

One easily shows that in case of a rollback g_0R returns the values in R:

$$c_I^{T+1}.R \quad \overset{!}{=} \quad \Omega R(c_S^{sI(0,T+1)})$$

These registers hold the value provided by f_kR in case of a rollback by definition. Thus, the claim is transformed into:

$$f\gamma_kR(c_I^T) \quad \overset{!}{=} \quad \Omega R(c_S^{sI(0,T+1)})$$

As above, we argue that the inputs of f_kR are correct, which transforms the claim into:

$$f\Gamma_kR(c_S^{sI(k,T)}) \quad \overset{!}{=} \quad \Omega R(c_S^{sI(0,T+1)})$$

As above, we apply the lemma that shows the correctness of f_kR, which transforms the claim into:

$$\Omega R(c_S^{sI(k,T)}) \quad \overset{!}{=} \quad \Omega R(c_S^{sI(0,T+1)})$$

It is left to show that $sI(k,T)$ is equal to $sI(0,T+1)$. This is easily done using lemma 5.40. QED

Consider the following situation: Let us have a rollback in cycle T. Using the lemma above, we can conclude that we have the correct data in the registers $c_I.R$ during cycle $T+1$. Now let stage 0 be stalled during cycle $T+1$ for any reason. We now have to argue that the correct data is preserved for subsequent cycles until another rollback happens or the update enable signal gets activated. If not so, we could get an infinite loop of rollbacks if we “forget” the correct data because of stalls.

Lemma 5.42 ► Given that both ue_0^T and $rollback'^T_0$ are not active and we have no misspeculation, we also have no misspeculation in cycle $T+1$.

$$\overline{ue_0^T} \wedge \overline{rollback'^T_0} \wedge \mathsf{S}_{-1}^T \quad \Longrightarrow \quad \mathsf{S}_{-1}^{T+1}$$

PROOF By definition of S_{-1}, the claim is:

$$g_0R(c_I^T) = \Omega R(c_S^{sI(0,T)}) \quad \overset{!}{\Longrightarrow} \quad g_0R(c_I^{T+1}) = \Omega R(c_S^{sI(0,T+1)})$$

One easily concludes this claim as follows: Using invariant 5.1, one argues that $sI(0,T) = sI(0,T+1)$ holds. This transforms the claim into:

$$g_0R(c_I^T) \quad \overset{!}{=} \quad g_0R(c_I^{T+1})$$

One easily asserts this using the fact that the registers g_0R depends on do not change from cycle T to $T+1$ because of the disabled rollback and update enable signals.

QED

We now define a predicate $Mc(T)$ that holds if we have a guarantee that the instruction in stage $M(T)$ will not rollback. We argue that once an instruction causes a rollback, we have a guarantee that it will not do so a second time. Thus, we will later on prove that $Mc(T)$ implies that the rollback signal of stage $M(T)$ is not active during cycle T.

We provide a recursive definition for $Mc(T)$ in analogy to the scheduling function $sI(k,T)$. In cycle $T = 0$, we have no guarantee that instruction I_0 will not cause a rollback. Thus, we define $Mc(0)$ to be false.

For $T > 0$, we define $Mc(T)$ using the rollback and update enable signals. If $rollback'^{T-1}_{M(T-1)}$ holds, we have a rollback and we argue that the instruction in stage $M(T-1)$ during cycle $T-1$ will not rollback a second time. Because of the rollback, that instruction is in stage 0 during cycle T. Because the rollback happened in the latest full stage, all stages later than stage 0 are empty during cycle T. Thus, the instruction that caused the rollback is still in the latest full stage. Thus, we define $Mc(T)$ to be true for this case.

$$rollback'^{T-1}_{M(T-1)} \quad \Longrightarrow \quad Mc(T) = 1$$

In case the update enable of stage $M(T-1)$ is active, the instruction proceeds into the next stage. We claim that the guarantee is maintained, i.e., that $Mc(T) = Mc(T-1)$ holds. In case there is no next stage, i.e., in case of stage $M(T-1) = n-1$, the instruction we have a guarantee for leaves the pipeline. In this case, we no longer have a guarantee and therefore define $Mc(T)$ to be false.

$$\begin{aligned} ue^{T-1}_{M(T-1)} \wedge M(T-1) \neq n-1 \quad &\Longrightarrow \quad Mc(T) = Mc(T-1) \\ ue^{T-1}_{M(T-1)} \wedge M(T-1) = n-1 \quad &\Longrightarrow \quad Mc(T) = 0 \end{aligned}$$

In case neither rollback nor update enable is active, we claim that $Mc(T)$ is $Mc(T-1)$. In order to summarize, the complete definition of $Mc(T)$ is:

$$Mc(T) \quad = \quad \begin{cases} 0 & : \quad T = 0 \\ 1 & : \quad rollback'^{\,T-1}_{M(T-1)} \\ 0 & : \quad ue^{T-1}_{M(T-1)} \wedge M(T-1) = n-1 \\ Mc(T-1) & : \quad \text{otherwise} \end{cases}$$

Lemma 5.43 ▶ Let T and T' be cycles with $T' \geq T$. Given that both the update enable and *rollback'* signals of stage $M(T)$ are not active during cycles $T \leq T'' < T'$, $M(T')$ is equal to $M(T)$ and $Mc(T')$ is equal to $Mc(T)$.

$$\begin{aligned} & (\forall T \leq T'' < T' : \overline{ue^{T''}_{M(T)}} \wedge \overline{rollback'^{\,T''}_{M(T)}}) \\ \Longrightarrow \quad & M(T) = M(T') \wedge Mc(T) = Mc(T') \end{aligned}$$

PROOF We show this claim using induction on T'. For $T' = 0$, we have $T = T'$ and the claim obviously holds. For $T'+1$, we easily conclude the claim as follows:

1. We conclude $M(T'+1) = M(T')$ by using lemma 5.37 and the fact that the update enable and *rollback'* signals are not active. We then use the induction premise in oder to conclude $M(T') = M(T)$.

2. We conclude $Mc(T'+1) = Mc(T')$ by expanding the definition of $Mc(T'+1)$ and the fact that the update enable and *rollback'* signals are not active. We then use the induction premise in oder to conclude $Mc(T') = Mc(T)$.

QED

Lemma 5.44 ▶ For all stages k, let the external stall signals of stage k be finite true and stay until ue_k. For all cycles T, the stall signal of stage $M(T)$ eventually gets deactivated after cycle T.

$$\exists^{\geq T} \overline{stall_{M(T)}}$$

PROOF Remember that the stall signal is calculated using internal and external stall signals:

$$stall^T_k \quad = \quad full^T_k \wedge (ext^T_k \vee int^T_k)$$

The internal stall signals handle data hazards and pipeline stalls, the external stall signals are used for caches, for example. According to lemma

4.32 (page 140), the disjunction of the external stall signals ext_k is finite true. Thus, there is a cycle $T' \geq T$ such that $ext^{T'}_{M(T)}$ is not active. Let T' be the earliest such cycle.

Observe that both lemma 4.33 (page 141) and lemma 4.31 (page 139) hold also in the pipelined machine with speculation (the proof uses the same arguments).

According to lemma 5.35, we have $below\_empty(M(T),T)$. According to lemma 4.31, the stages below stage $M(T)$ stay empty at least until $ue_{M(T)}$ becomes active. One easily concludes that this does not happen before cycle T' because before cycle T', the external stall signal is active. Thus, we have $below\_empty(M(T),T')$.

Lemma 4.33 states that empty stages do not cause internal stall signals. According to this lemma, the internal stall signals of stage $M(T)$ cannot be active during cycle T' because the stages below stage $M(T)$ are empty.

Thus, both ext_k and int_k are not active during cycle T'. This implies that the stall signal is not active and the claim holds.

QED

Informally, consider an instruction in a stage. Assuming the stall signal of the stage will eventually be deactivated, the instruction in the stage either moves into the next stage or gets evicted because of a rollback.

◄ Lemma 5.45

Formally, let stage k be full during cycle T. Let there be a cycle $T' \geq T$ such that the stall signal $stall_k$ is not active. This implies that either the update enable signal of stage $M(T)$ or the $rollback'$ signal of stage $M(T)$ eventually gets activated.

$$full^T_k \wedge \exists^{\geq T}\overline{stall_k} \implies \exists^{\geq T}(ue_k \vee rollback'{}_k)$$

PROOF

The proof is done in analogy to the proof of the counterpart lemma of the machine without speculation, lemma 3.20 (page 87).

◄ Lemma 5.46

For all cycles T, either the update enable signal of stage $M(T)$ or the $rollback'$ signal of stage $M(T)$ eventually gets activated.

$$\exists^{\geq T}(ue_{M(T)} \vee rollback'{}_{M(T)})$$

PROOF This claim is shown by instantiating lemma 3.20 with stage $M(T)$. We show $full_{M(T)}^{T}$ using lemma 5.33. We show that the stall signal will eventually be deactivated by using lemma 5.44.

Lemma 5.47 ▶ The speculation stage $\Sigma(T)$ is always above or equal to the last full stage.

$$\Sigma(T) \quad \le \quad M(T)$$

PROOF According to lemma 5.20, stage $\Sigma(T)$ is full during cycle T. Thus, this cannot be below the last full stage.

Lemma 5.48 ▶ The non-speculative inputs of stage $M(T)$ are always correct.

$$\mathsf{I}_{M(T)}(T)$$

PROOF By definition of $\mathsf{I}_{M(T)}(T)$, we have to show:

$$\begin{array}{c}(\forall l : l \ge M(T)-1 \quad \Longrightarrow \quad \mathsf{P}_l(T)) \ \wedge \\ (\forall l : l \ge M(T) \quad \Longrightarrow \quad \mathsf{P}'_l(T))\end{array}$$

Remember that $\mathsf{P}_l(T)$ denoted non-speculative output registers of stage l and $\mathsf{P}'_l(T)$ denoted the output registers of stage l that depend on speculative registers. We easily conclude both claims using the data consistency of the machine (invariant 5.5, page 186) and $\Sigma(T) \le M(T)$ (lemma 5.47). QED

Lemma 5.49 ▶ If $Mc(T)$ holds, the speculation registers that are output of stage $M(T)-1$ hold correct values.

$$Mc(T) \quad \Longrightarrow \quad \mathsf{S}_{M(T)-1}(T)$$

PROOF We show this claim by induction on T. For $T=0$, we have nothing to show since $Mc(0)$ does not hold.

For $T+1$, we show the claim as follows:

In case we have a rollback, i.e., if $rollback'^{T}_{M(T)}$ is active, we have $M(T+1)=0$ and we have to show $\mathsf{S}_{-1}(T+1)$, which is easily done using lemma 5.41.

In case we do not have a rollback but an active update enable signal $ue^T_{M(T)}$, we have $Mc(T+1) = Mc(T)$. In case $Mc(T+1)$ does not hold, there is nothing to show. Thus, $Mc(T)$ holds, and we therefore have $\mathsf{S}_{M(T)-1}(T)$. Using lemma 5.15, we conclude $\mathsf{S}_{M(T)}(T+1)$. According to lemma 5.37, we have $M(T+1) = M(T)+1$, and therefore $\mathsf{S}_{M(T+1)-1}(T+1)$, which concludes the claim.

In case both signals are not active, we have to do a case split on the value of $M(T)$: In case $M(T)$ is zero, we conclude the claim using lemma 5.42. If not so, we argue that $ue^T_{M(T)-1}$ cannot be active using lemma 4.3. According to lemma 5.37, we have $M(T+1) = M(T)$, thus, we have to show $\mathsf{S}_{M(T)}(T+1)$, which is easily done using lemma 5.13 for stage $M(T)$.

QED

The following lemma shows that the “intended meaning” of $Mc(T)$ is achieved, i.e., that $Mc(T)$ implies that we do not have a rollback in stage $M(T)$.

◄ Lemma 5.50

If $Mc(T)$ holds, the *rollback′* signal of stage $M(T)$ cannot be active during cycle T.

$$Mc(T) \implies \overline{rollback'^{T}_{M(T)}}$$

PROOF

According to lemma 5.49, we have $\mathsf{S}_{M(T)-1}(T)$.

Assume $rollback'^{T}_{M(T)}$ is active. Using lemma 5.39, we can conclude that $\rho(M(T),T)$ holds, i.e., stage $M(T)$ is the last stage with active *rollback′* signal. Using lemma 5.48, we conclude $\mathtt{I}_{M(T)}(T)$. This allows applying lemma 5.41 for stage $M(T)$.

Lemma 5.41 states that $\mathsf{S}_{M(T)-1}(T)$ cannot hold, which is a contradiction.

QED

We now proceed in the liveness proof as follows: we show that an instruction in the last full stage is live, i.e., eventually moves into the next stage. The first step is to show this assuming we have a guarantee that the instruction will not cause a rollback. One easily shows this.

The next step is to conclude that this also happens in case we do not have that guarantee. We do this by arguing that the instruction will rollback at most once in the worst case.

Lemma 5.51 ▶ Assume we have a guarantee that the instruction in stage $M(T)$ will not rollback. In this case, we claim that there is a cycle $T' \geq T$ such that the update enable signal is active and no rollback is signaled during cycles T to T'.

$$Mc(T) \implies \exists T' \geq T: ue_{M(T)}^{T'} \wedge \forall T \leq T'' \leq T': \overline{rollback'^{T''}_{M(T)}}$$

PROOF According to lemma 5.46, we have a cycle $T' \geq T$ such that either the rollback signal $rollback'_{M(T)}$ or the update enable signal $ue_{M(T)}$ is active. Let T' be the smallest such cycle.

We will disprove that $rollback'_{M(T)}$ can be active. Using lemma 5.43, we conclude that $Mc(T')$ holds. Using lemma 5.50, we conclude that $rollback'_{M(T')}$ cannot be active.

Thus, $ue_{M(T)}$ must be active during cycle T'. We will show that cycle T' satisfies the claim. It is left to show that $rollback'_{M(T)}$ is not active from the cycles T to T'. For cycle T', we conclude this from the definition of the update enable signal (the update enable signal is not active in case of a rollback). For cycles T'' with $T \leq T'' < T'$, we conclude this from the fact that T' is the smallest cycle such that either $rollback'_{M(T)}$ or $ue_{M(T)}$ is active. QED

The following lemma is the counterpart of lemma 3.23 (page 88) for the machine without speculation. The proof is done in analogy to the proof of lemma 3.23.

Lemma 5.52 ▶ Let T and $T' \geq T$ be cycles. Let the update enable signal ue_k and the $rollback'_k$ signal of a stage k be off during the cycles T'' with $T' > T'' \geq T$. The value of the scheduling function does not change from cycle T to T'.

$$\forall T'' | T' > T'' \geq T : \overline{ue_k^{T''}} \wedge \overline{rollback'^{T''}_k} \implies sI(k,T) = sI(k,T')$$

Lemma 5.53 ▶ Given that $Mc(T)$ holds, there is a cycle $T' \geq T$ such that the next instruction is in stage $M(T)$.

$$Mc(T) \implies \exists T' \geq T: sI(M(T),T') = sI(M(T),T) + 1$$

PROOF Let T'' be the earliest cycle with active update enable signal according to lemma 5.51, i.e., we have $ue^{T''}_{M(T)}$. Using lemma 5.52, we conclude that the value of the scheduling function for stage $M(T)$ does not change from cycle T to T'':

$$sI(M(T),T) \quad = \quad sI(M(T),T'')$$

We then use invariant 5.1 in order to conclude that the value of the scheduling function for stage $M(T)$ increases by one from cycle T'' to cycle $T''+1$:

$$sI(M(T),T''+1) \quad = \quad sI(M(T),T'')+1$$

Thus, cycle $T''+1$ satisfies our claim. QED

Lemma 5.54

Let $Mc(T)$ hold, i.e., we have a guarantee that the instruction in stage $M(T)$ will not rollback. Furthermore, let this stage not be the last stage, i.e., $M(T) < n-1$. In this case, there is a cycle T' such that the instruction in stage $M(T)$ during cycle T is now in stage $M(T)+1$. Furthermore, the last full stage during cycle T' is stage $M(T)+1$ and the instruction in that stage is guaranteed not to rollback.

$$\begin{aligned}\Longrightarrow \quad \exists T' \geq T: \quad & sI(M(T)+1,T') = sI(M(T),T) \wedge \\ & M(T') = M(T)+1 \wedge \\ & Mc(T')\end{aligned}$$

PROOF

Let T'' be the earliest cycle with active update enable signal according to lemma 5.51, i.e., we have $ue^{T'}_{M(T)}$. We will show that $T''+1$ satisfies the claim above. We show the three parts of the claim separately.

1. We show $sI(M(T)+1,T''+1) = sI(M(T),T)$ as follows: Using the same arguments as in the proof of lemma 5.53, we conclude:

   $$sI(M(T),T''+1) \quad = \quad sI(M(T),T)+1$$

   One easily shows that the full signal $full^{T''+1}_{M(T)+1}$ is active using that the update enable signal is active. We then apply invariant 5.3, which states:

   $$sI(M(T),T''+1) \quad = \quad sI(M(T)+1,T''+1)+1$$

   Thus, the first part of the claim is satisfied by $T''+1$.

2. We show $M(T''+1) = M(T)+1$ as follows: Using lemma 5.43, we conclude that M does not change from cycle T to T''. Using lemma 5.37, we conclude $M(T''+1) = M(T'')+1$. Thus, $T''+1$ satisfies the claim.

3. We show $Mc(T''+1)$ as follows: Using lemma 5.43, we conclude that M does not change from cycle T to T''. By definition of $Mc(T''+1)$, we conclude $Mc(T''+1) = Mc(T'')$. Thus, $T''+1$ satisfies the claim.

QED

We can extend the claim of lemma 5.54 to multiple stages using induction:

Lemma 5.55 ► Let $Mc(T)$ hold, i.e., we have a guarantee that the instruction in stage $M(T)$ will not rollback. Consider a stage $k \geq M(T)$. The claim is that there is a cycle $T' \geq T$ such that the instruction in stage $M(T)$ during cycle T is in stage k during cycle T'. Furthermore, we claim that stage k is the last full stage during cycle T' and that the instruction will not rollback.

$$\begin{aligned} k \geq M(T) \implies \quad \exists T' \geq T: \quad & sI(k,T') = sI(M(T),T) \wedge \\ & M(T') = k \wedge \\ & Mc(T') \end{aligned}$$

PROOF We show the claim by induction on k. For $k = M(T)$, the claim obviously holds. For the step from k to $k+1$, we apply lemma 5.54.

The following lemma has the very same claim as lemma 5.53. However, we no longer premise that the instruction in stage $M(T)$ is guaranteed not to rollback.

Lemma 5.56 ► For all cycles T, there is a cycle $T' \geq T$ such that the next instruction moves into stage $M(T)$.

$$\exists T' \geq T: \; sI(M(T),T') = sI(M(T),T)+1$$

PROOF We use lemma 5.46 in order to conclude that there is a cycle $T'' \geq T$ such that either the update enable or *rollback'* signal is active. Let T'' be the earliest such cycle. In case the update enable signal is active, we conclude the claim as done in the proof of lemma 5.53.

In case the rollback signal is active, we conclude the claim as follows: Using lemma 5.52, we conclude that the value of the scheduling function for stage $M(T)$ does not change from cycle T to T'':

$$sI(M(T),T) \quad = \quad sI(M(T),T'')$$

We then use lemma 5.43 in order to conclude that $M(T'') = M(T)$. This allows applying lemma 5.39. Lemma 5.39 states that stage $M(T)$ is the last stage with active rollback signal during cycle T''. This allows applying lemma 5.40 with $l = 0$. Lemma 5.40 states that $sI(0,T''+1)$ is equal to $sI(M(T),T'')$.

Because of the rollback, we have $M(T''+1) = 0$ by lemma 5.37. Thus, we have:

$$sI(M(T),T) \quad = \quad sI(M(T''+1),T''+1)$$

Since we have a rollback, we now have a guarantee that the instruction in stage $M(T''+1)$ during cycle $T''+1$ will not rollback. Thus, we can apply lemma 5.55 in order to conclude that this instruction eventually moves into stage $M(T)$. Let t be that cycle.

$$sI(M(T),T) \quad = \quad sI(M(T),t)$$

We will then use lemma 5.53 in order to conclude that the value of the scheduling function will eventually increase by one. Let t' be that cycle.

$$sI(M(T),T)+1 \quad = \quad sI(M(T),t')$$

Thus, cycle t' satisfies the claim. QED

The following lemma has a similar claim as lemma 5.54. However, we do not premise that we have a guarantee that the instruction in stage $M(T)$ will not rollback.

◄ Lemma 5.57

Let $M(T)$ not be the last stage. There is a cycle $T' \geq T$ such that the instruction in stage $M(T)$ during cycle T is in stage $M(T)+1$ during cycle T'. Furthermore, stage $M(T)+1$ is the last full stage during cycle T'.

$$\begin{aligned} M(T) < n-1 \Longrightarrow \quad \exists T' \geq T : \quad & sI(M(T)+1,T') = sI(M(T),T) \wedge \\ & M(T') = M(T)+1 \end{aligned}$$

k	$full_k$	$sI(k,T)$	
0	1	4	
1	1	3	
2	0	3	
3	1	2	
4	1	1	
5	1	0	$\longleftarrow M(T)$
6	0	0	
7	0	0	$\longleftarrow n-1$

Table 5.5 Illustration of lemma 5.59: In a pipeline with $n = 8$ stages, we have $M(T) = 5$ and therefore $sI(5,T) = sI(7,T)$.

PROOF The proof follows the same pattern as the proof of the lemma 5.56: in case the update enable signal becomes active, we argue as in lemma 5.54. If not so, we have a rollback and continue as in lemma 5.56.

The following lemma is an inductive extention of lemma 5.57.

Lemma 5.58 ► Consider an instruction in the last full stage during cycle T. There is a cycle T' such that this instruction is in the last stage and such that the last stage is the last full stage.

$$\begin{aligned} \exists T' : \quad & sI(n-1,T') = sI(M(T),T) \wedge \\ & M(T') = n-1 \end{aligned}$$

PROOF Let k be the number of the last full stage. One easily concludes this claim by induction on k. One starts with the last stage and proceeds inductively from stage k to stage $k-1$ until the desired stage is reached. The induction step is argued using lemma 5.57.
QED

Lemma 5.59 ► The value of the scheduling function $sI(M(T),T)$ is equal to the value of the scheduling function in the last stage $sI(n-1,T)$.

$$sI(M(T),T) \quad = \quad sI(n-1,T)$$

This lemma is illustrated exemplary in table 5.5.

PROOF Let k be the number of the last full stage. One easily concludes this claim by induction on k. One starts with the last stage and proceeds inductively from stage k to stage $k-1$ until the desired stage is reached. The induction step from k to $k-1$ is argued as follows: since $full_k$ does not hold, one can use the scheduling invariants 5.2 and 5.3 in order to argue that

$$sI(k,T) \quad = \quad sI(k-1,T)$$

holds. QED

◀ Lemma 5.60 For all instructions I_i, there is a cycle T such that the value of the scheduling function for the last stage and cycle T is i.

$$\exists T: \qquad sI(n-1,T) = i$$

PROOF One shows this claim using induction on i. For $i = 0$, $T = 0$ satisfies the claim.

For $i+1$, we show the claim as follows: According to the induction premise, there is a cycle T such that $sI(n-1,T) = i$ holds. According to lemma 5.59, we have instruction I_i also in the last full stage, i.e., $sI(M(T),T) = i$.

We use lemma 5.58 in order to argue that instruction i is eventually in the last stage, i.e., we have a cycle T' such that $sI(n-1,T') = i$ and $M(T') = n-1$. We then use lemma 5.56 in order to conclude that there is a cycle T'' such that $sI(n-1,T'') = i+1$. Thus, cycle T'' satisfies the claim. QED

5.11.4 Liveness Proof

Using lemma 5.60, we show that for all instructions I_i, there is a cycle T such that this instruction is in the last stage of the pipeline. However, our liveness criterion as proposed in chapter 3 is stronger: it requires that we can provide such a cycle T for each stage and not just for the last stage.

We will now argue as follows: given that an instruction is in the last stage, there must be cycles T' such that I_i was in all stages $k < n-1$ before. For intuition, this means that instructions never skip over a stage.

Lemma 5.61 ▶ Let $k > 0$ be a stage. Let I_i be the instruction given by $sI(k,T)$. In this case, there is a cycle T' such that $sI(k-1,T')$ is i. For intuition, if you have an instruction in a stage $k > 0$, there must be an earlier cycle such that this instruction is in the previous stage.

$$k > 0 \wedge sI(k,T) = i \implies \exists T' : sI(k-1,T') = i$$

PROOF We show this claim using induction on T. For $T = 0$, $T' = 0$ satisfies the claim since we have $sI(k,0) = sI(k-1,0) = 0$.

For $T+1$, we have the following claim:

$$k > 0 \wedge sI(k,T+1) = i \overset{!}{\implies} \exists T' : sI(k-1,T') = i$$

We show the claim as follows: Assume we have $k < n-1$ and an active $rollback'_{k+1}$ signal during cycle T. We will show that cycle $T+1$ satisfies $sI(k-1,T+1) = i$. By definition of the $rollback'$ signals, $rollback'_k$ must be active during cycle T. This implies that $sI(k-1,T+1)$ is equal to $sI(k,T+1)$ by definition of $sI(k-1,T+1)$.

Let $k = n-1$ or $\overline{rollback'^{T}_{k+1}}$ hold. If the update enable signal ue_k^T is active, the desired instruction was in stage $k-1$ during cycle T, which satisfies the claim. If the update enable signal ue_k^T is not active, the instruction was in stage k during cycle T. In this case, we apply the induction premise, which provides a cycle T' that satisfies the claim. QED

Lemma 5.62 ▶ We extend the argument of the previous lemma inductively for multiple stages: Let k and $l \leq k$ be stages and let I_i be the instruction given by $sI(k,T)$. In this case, there is a cycle T' such that $sI(l,T')$ is i.

$$sI(k,T) = i \wedge l \leq k \implies \exists T' : sI(l,T') = i$$

PROOF We show the claim using induction on l. We start with $l = k$ and proceed from l to $l-1$. For $l = k$, the claim obviously holds. For the step from stage l to $l-1$ we apply lemma 5.61.

Theorem 5.63 For all instructions I_i and stages k, there is a cycle T such that $sI(k,T)$ is equal to i. This is the liveness criterion proposed in chapter 3.

$$\exists T: \quad sI(k,T) = i$$

PROOF Using lemma 5.60, we conclude that there is a cycle T' such that $sI(n-1,T') = i$ holds. For $k = n-1$, this satisfies the claim.

Thus, let $k \neq n-1$ hold. In this case, we apply lemma 5.62, which provides us with a cycle T'' that satisfies the claim. QED

5.12 Precise Interrupts

5.12.1 Definition

Interrupts are events that change the flow of control of a program by means other than a branch instruction [MP00]. They are used in order to realize virtual memory, fast I/O, and arithmetic error handling.

In case of an interrupt, the state of the machine is saved and the execution proceeds with an interrupt service routine (ISR). After the interrupt service routine is done, the state of the machine is restored and the execution of the program proceeds.

An interrupt between instruction I_{i-1} and I_i is precise if instructions I_0 to I_{i-1} are completed before starting the ISR and later instructions $(I_i, \ldots)$ did not change the state of the machine [SP88, Mül97].

5.12.2 The DLX with Interrupts

The specification of a DLX with interrupts used in the following section is taken from [MP00]. Interrupts are events other than branches that modify

Table 5.6 Special purpose registers used for exception handling

address	name	meaning
0	SR	status register
1	ESR	exception status register
2	ECA	exception cause register
3	EPC	the exception PC
4	$EDPC$	the exception delayed PC
5	$EDATA$	exception data register

the flow of control. Each such event is assigned a number in $\{0,1,\ldots\}$. If such an event occurs, the next instruction fetched and executed is taken from a special interrupt service routine. The address of this interrupt service routine is denoted by *SISR*. After the interrupt service routine is done, there are three ways to resume the execution:

1. The interrupted instruction is repeated.
2. The execution is continued with the instruction that follows the interrupted instruction.
3. The program execution is aborted.

In order to support interrupt handling, the instruction set architecture of the machine is extended. A set of registers is added to the configuration of the machine: the registers are called *special purpose registers* and are listed in table 5.6. Each register is 32 bits wide.

In order to access these new registers, two instructions are added: the instruction *movs*2*i* reads a special purpose register and stores the value in a GPR register. The instruction *movi*2*s* reads a GPR register and stores the value in a given special purpose register. The transition function $\delta.GPR$ is changed accordingly. Given an instruction word I, these instructions are indicated by $I\_movi2s(I)$ and $I\_movs2i(I)$.

The special purpose register *SR* is used in order to mask interrupts. If bit j in the register *SR* is set, the interrupt number j is handled. If bit j is not set, the interrupt is suppressed. However, not all interrupts can be suppressed using *SR*. Interrupts that can be suppressed are called *maskable*.

Table 5.7 lists the interrupts supported by the DLX without floating point instructions. This list is taken from [MP00]. The reset interrupt occurs

Interrupt	Symbol	Priority	Resume	Maskable
reset	*reset*	0	abort	no
illegal instruction	*ill*	1		
misaligned access	*mal*	2		
page fault IM	*ipf*	3	repeat	
page fault DM	*dpf*	4		
trap	*trap*	5	continue	
FXU overflow	*ovf*	6		yes
external I/O	$ex[j]$	$7+j$		

Table 5.7 The Interrupts and their priority

directly in the initial configuration of the machine. Thus, we start the execution after reset at the interrupt service routine and no longer at address zero.

The illegal instruction interrupt occurs iff the instruction word fetched does not encode a valid instruction. The misaligned access interrupt occurs iff the instruction fetch or if the data memory access is not well-aligned. The page fault IM/DM interrupts occur iff the memory system signals a page fault during an instruction fetch or data memory access, respectively. The trap interrupt is caused by a special instruction *trap*. It can be used for system calls, for example. The trap instruction allows passing an immediate constant as parameter.

The FXU overflow interrupt occurs if an unmasked overflow occurs during an ALU instruction. The external I/O interrupts occur if an external signal $ex_S[j]$ with $j \geq 0$ is active. These external interrupts can be used in order to realize fast I/O such as access to hard disks or networks.

Let *CA* denote a 32-bit signal that is defined as follows: iff an interrupt with number j occurs, bit j of this signal is active. Let c be a configuration of the specification machine. Using *CA*, the 32-bit signal *MCA* is defined as follows:

$$MCA(c)[j] \quad := \quad \begin{cases} CA(c)[j] & : \text{ if interrupt } j \text{ is not maskable} \\ CA(c)[j] \wedge SR[j] & : \text{ if interrupt } j \text{ is maskable} \end{cases}$$

Thus, an interrupt is handled if there is at least one bit in $MCA(c)$ set. This is indicated by the one bit signal *JISR*:

$$JISR(c) \quad := \quad \exists j \in \{0,\ldots,31\} : MCA(c)[j]$$

If multiple interrupts occur, the interrupt with the lowest number is handled with priority. The interrupt that is handled is indicated by a 32-bit signal *il* (interrupt level). If no interrupt is to be handled, all bits of *il* are zero. If there is an interrupt j to handle, exactly bit j is set.

$$il(c)[j] \quad := \quad \begin{cases} 1 & : \; JISR(c) \wedge \\ & \quad j = \min\{i \in \{0,\dots,31\} \,|MCA(c)[i]\} \\ 0 & : \; \text{otherwise} \end{cases}$$

The same interrupt service routine is used in order to handle all interrupts. Thus, in order to enable this interrupt service routine to distinguish the events that cause interrupts, a new special purpose register *ECA* is added to the configuration set of the machine. In case of an interrupt, the value of $MCA(c)$ is stored in *ECA*. The interrupt service routine is expected to handle the interrupt event with the smallest number j such that the bit $ECA[j]$ is set.

Instruction Fetch We support two interrupts that affect the instruction fetch. We check whether the instruction word address is misaligned. Given an effective address *ea*, the function $imal(ea)$ holds if we have a misaligned instruction word:

$$imal(ea) \quad := \quad ea[0] \vee ea[1] \tag{5.15}$$

Furthermore, we support page faults for the instruction memory access. Page faults are indicated by an external signal $ipf_S(c)$.

If no page fault happens and if the instruction word is not misaligned, the instruction word $I(c)$ is defined as in chapter 2. In particular, we are back to using Delayed PC and no longer use branch prediction. In case of a misaligned instruction word or a page fault, we use zero as instruction word. The instruction encoded by zero actually turns out to be a NOP.

$$I(c) \quad := \quad \begin{cases} 0 & : \; imal(c.DPC) \vee ipf_S(c) \\ IM[c.DPC] & : \; \text{otherwise} \end{cases} \tag{5.16}$$

The transition functions of PC' and DPC are changed in order to realize the jump to the interrupt service routine and the rfe instruction. This instruction is used in order to return from the interrupt service routine. In case of an rfe instruction, the registers SR, PC', and DPC are restored from the corresponding special purpose registers.

In case of an interrupt, the new value of PC' is the address of the interrupt service routine (SISR) plus four, i.e., the second instruction of the interrupt service routine. In case of an rfe instruction, the value in EPC is taken. Otherwise, the next PC is calculated as in the machine without interrupts. We define a function $nextpc'(I, op1, PC, EPC)$ as follows: in case of an rfe instruction, it returns EPC. Otherwise, the value provided by $nextpc$ as defined in chapter 2 is returned:

$$nextpc'(I, op1, PC, EPC) \quad := \quad \begin{cases} EPC & : \ I\_rfe(I) \\ nextpc(I, op1, PC) & : \ \text{otherwise} \end{cases}$$

As before, $op1$ is the first GPR operand. We use this new $nextpc'$ function in order to define the new transition function for the PC' register:

$$\delta.PC'(c) \quad := \quad \begin{cases} SISR + 4 & : \ JISR(c) \\ nextpc'(I, op1, c.PC', c.EPC) & : \ \text{otherwise} \end{cases}$$

The transition function of DPC is no longer the identity. In case of an interrupt, the new value of DPC is the address of the interrupt service routine (SISR), i.e., the first instruction of the interrupt service routine. In case of an rfe instruction, the value in $EDPC$ is restored. Otherwise, the new value of DPC is calculated as in the machine without interrupts.

$$\delta.DPC(c) \quad := \quad \begin{cases} SISR & : \ JISR(c) \\ c.EDPC & : \ \overline{JISR(c)} \wedge I\_rfe(c) \\ c.PC' & : \ \text{otherwise} \end{cases}$$

Data Memory Exceptions We have two exceptions that are caused by data memory accesses: data memory page faults are used in order to implement virtual memory, data memory misalignment interrupts indicate a misaligned memory access.

Data memory page faults are indicated by an external signal dpf. A misaligned memory access is detected using the the effective address of the memory access and the instruction word.

The functions $memW$ and $memH$ hold if the memory operand of the given instruction is of word or half-word size, respectively. In case of stores, we only support word size accesses.

$$\begin{aligned} memW(I) &= (I\_load(Iw) \wedge I\_lw(Iw)) \vee I\_store(Iw) \\ memH(I) &= (I\_load(Iw) \wedge (I\_lh(Iw) \vee I_l hu(Iw))) \end{aligned}$$

Given an effective address EA, we have a misaligned address, if we have a word access with active $EA(0)$ or active $EA(1)$ or if we have a half-word access with active $EA(0)$. This is indicated by *malAc*:

$$\begin{aligned} malAc(I,EA) &= (memW(I) \wedge (EA(0) \vee EA(1)) \vee \\ &\quad (memH(I) \wedge (EA(0)) \end{aligned}$$

We have a data memory misalignment exception in case of a load or store instruction with misaligned address:

$$dmal(Iw,EA) = (I\_load(I) \vee I\_store(I)) \wedge malAc(I,EA)$$

Transition Function of the SPRs Let S_j be a special purpose register. In case there is no interrupt, we define the register transition function $\delta.S_j$ as follows: we take the first GPR operand in case we have a *movi*2*s* instruction with appropriate address and the old value otherwise:

$$\overline{JISR(c)} \Longrightarrow$$
$$\delta.S_j(c) = \begin{cases} op1(c) & : I\_movi2s(I) \wedge \\ & \quad \langle I\_immediate(I)[4:0] \rangle = j \\ c.S_j & : \text{otherwise} \end{cases}$$

The transition function in case there is an interrupt depends on the register.

As described above, we store the value of *MCA* in the register *ECA* in case of an interrupt:

$$\overline{JISR(c)} \Longrightarrow \delta.ECA(c) = MCA(c)$$

The *EDATA* special purpose register is used in order to store additional information about the exception. In case of a trap instruction, the immediate constant provided with the instruction is stored in *EDATA*. This allows passing of an argument to the interrupt service routine. In case of a page fault or misaligned memory access, the memory address accessed is stored in *EDATA*. In case of any other interrupt, we store zero in *EDATA*.

Let $mem(c)$ indicate that we execute a load or store instruction:

$$mem(c) := I\_load(I) \vee I\_store(I) \qquad (5.17)$$

Let $dmemea(c)$ denote the effective address of a data memory access. In case of an interrupt, the transition function for $EDATA$ is:

$$\overline{JISR(c)} \Longrightarrow$$
$$\delta.EDATA(c) = \begin{cases} c.DPC & : imal(c.DPC) \vee ipf_S(c) \\ dmemea(c) & : dpf_S(c) \wedge mem(c) \\ I\_immediate(c) & : I\_trap(I) \\ 0 & : \text{otherwise} \end{cases}$$

Furthermore, the values of the registers DPC and PC' are saved in special purpose registers $EDPC$ and EPC in order to support resuming the instruction after the execution of the interrupt service routine. This depends on whether the interrupt is of type repeat or continue. This is indicated by a one bit signal $repeat$:

$$repeat(c) \quad := \quad (il(c) = 3) \vee (il(c) = 4)$$

If the interrupt is of type repeat, the values of DPC and PC' in the current configuration are taken. If the interrupt is of type continue or abort, the values are taken that point to the following instruction, as calculated by $nextpc'$. In case of an interrupt, the transition functions for EPC and $EDPC$ are:

$$\overline{JISR(c)} \Longrightarrow$$
$$\delta.EPC(c) = \begin{cases} c.PC' & : repeat(c) \\ nextpc'(I, op1, c.PC', c.EPC) & : \text{otherwise} \end{cases}$$
$$\delta.EDPC(c) = \begin{cases} c.DPC & : repeat(c) \\ c.EDPC & : \overline{repeat(c)} \wedge I\_rfe(I) \\ c.PC' & : \text{otherwise} \end{cases}$$

In case of an interrupt, the register SR is set to zero. This masks all interrupts, which prevents that the interrupt service routine is interrupted:

$$\overline{JISR(c)} \quad \Longrightarrow \quad \delta.SR(c) = 0$$

In order to restore the register SR before resuming the program, the value of SR is saved in the special purpose register ESR. In case of an interrupt of type repeat, the value from the current configuration is taken. In case of an interrupt of type continue or abort, the value calculated for the next configuration is taken:

$$\overline{JISR(c)} \Longrightarrow$$
$$\delta.ESR(c) = \begin{cases} op1(c) & : \overline{repeat(c)} \wedge I\_movi2s(I) \wedge \\ & \quad \langle I\_immediate(I)[4:0] \rangle = 0 \\ c.SR & : \text{otherwise} \end{cases}$$

Furthermore, in case of an interrupt of type repeat, the write access to GPR and to the memory has to be suppressed. This is realized by modifying the transition function for GPR accordingly.

A complete description how the interrupt service routine is to be implemented such that it behaves like a procedure is given in [MP00].

5.12.3 Hardware for the DLX with Interrupts

In this section, we describe small circuits that are used for interrupt handling.

MCA The circuit $MCA(CA, SR)$ calculates the masked cause register given CA and the status register SR:

$$MCA\_impl(CA, SR)[i] := \begin{cases} CA[i] \wedge SR[i] & : 6 \leq i < 32 \\ CA[i] & : \text{otherwise} \end{cases}$$

Lemma 5.64 ▶ The circuit MCA is correct:

$$MCA\_impl(c_S) = MCA(CA(c_S), c_S.SR)$$

PROOF One easily asserts this claim by expanding the definitions of the functions MCA_impl and MCA.

JISR We calculate the $JISR$ signal using a zero tester and MCA:

$$JISR\_impl(MCA) := \overline{zerotester(MCA)}$$

Lemma 5.65 ▶ The calculation of $JISR$ is correct:

$$JISR\_impl(MCA(c_S)) = JISR(c_S)$$

One easily asserts this claim by expanding the definition of both functions and by applying lemma 2.2 (correctness of the zero tester, page 16).

repeat We calculate the repeat signal as done in [MP00]:

$$\begin{aligned} repeat\_impl(MCA) \quad := \quad & \overline{(MCA[0] \vee MCA[1] \vee MCA[2])} \\ & \wedge (MCA[3] \vee MCA[4]) \end{aligned}$$

◀ Lemma 5.66

The circuit *repeat_impl* is correct:

$$repeat\_impl(MCA(c_S)) \quad = \quad repeat(c_S)$$

PROOF

Let $MCA(c_S)$ be zero. In this case, one easily asserts that the bit provided by *repeat_impl* is not active. Furthermore, one asserts that $JISR(c_S)$ does not hold. This implies that $il(c_S)$ is also zero by definition. Thus, $repeat(c_S)$ does not hold, which concludes the claim for $MCA(c_S) = 0$.

Let $MCA(c_S)$ be not zero. In this case, one easily asserts that $JISR(c_S)$ is active. Furthermore, there is a smallest j such that $MCA(c_S)[j]$ holds. If this j is smaller than 3 or greater than 4, we do not have an interrupt of type repeat. One easily asserts that $repeat\_impl(MCA(s_S))$ does not hold in this case.

If it is equal to 3 or 4, we have an interrupt of type repeat. One easily asserts that $repeat\_impl(MCA(s_S))$ holds in this case.

QED

Decoder In order to realize the special purpose register file, we need a decoder:

◀ Definition 5.1
Decoder

Let k be an integer and n be 2^k. A decoder is a circuit with inputs $a \in bvec[k]$ and outputs $b \in bvec[n]$ such that for all i

$$b_i = 1 \quad \Longleftrightarrow \quad \langle a \rangle = i.$$

An implementation can be found in [MP95]. A PVS proof is covered by [BJK01].

5.12.4 Configuration of the Pipelined DLX with Interrupts

We implement the pipelined DLX with interrupts using speculation. Using the generic speculation mechanism from this chapter, and the generic forwarding mechanism from chapter 4, implementing the pipelined DLX with interrupts is quite easy. We do this in three steps:

1. We start with the pipelined machine without interrupts as presented in chapter 4. We add the special purpose registers, as described above, to the configuration set.
2. We add two speculative values: the first value, *JISR*, is a one bit register that indicates an interrupt. The second value, $repeat$, is a one bit register that is set iff the interrupt is of type $repeat$. Given those two speculative inputs, we can almost copy the specification above in order to get an implementation.
3. We add write accesses to *JISR* and $repeat$ in order to detect any misspeculation.

Figure 5.16 gives an overview of the DLX pipeline with precise interrupts. We now describe the changes to the pipelined machine in detail.

Configuration Set We extend the configuration set of the pipelined machine without interrupts by the special purpose registers as given by the specification of the DLX with interrupts. We furthermore add a set of implementation registers that we will describe later on.

Initial Configuration As before, the initial values of *GPR* and *DM* are arbitrary but fixed. The register *DPC* is initialized with *SISR*, the register PC' with $SISR+4$. This will cause the ISR to be executed. All special purpose registers except for *ECA* are initialized with zero. The register *ECA* is initialized with one in order to indicate the reset.

5.12.5 Transition Functions of Stage 0

In stage 0, we do the instruction fetch. This is done by a write access to the *IR* register. The write access depends on *DPC* and on ipf. We follow

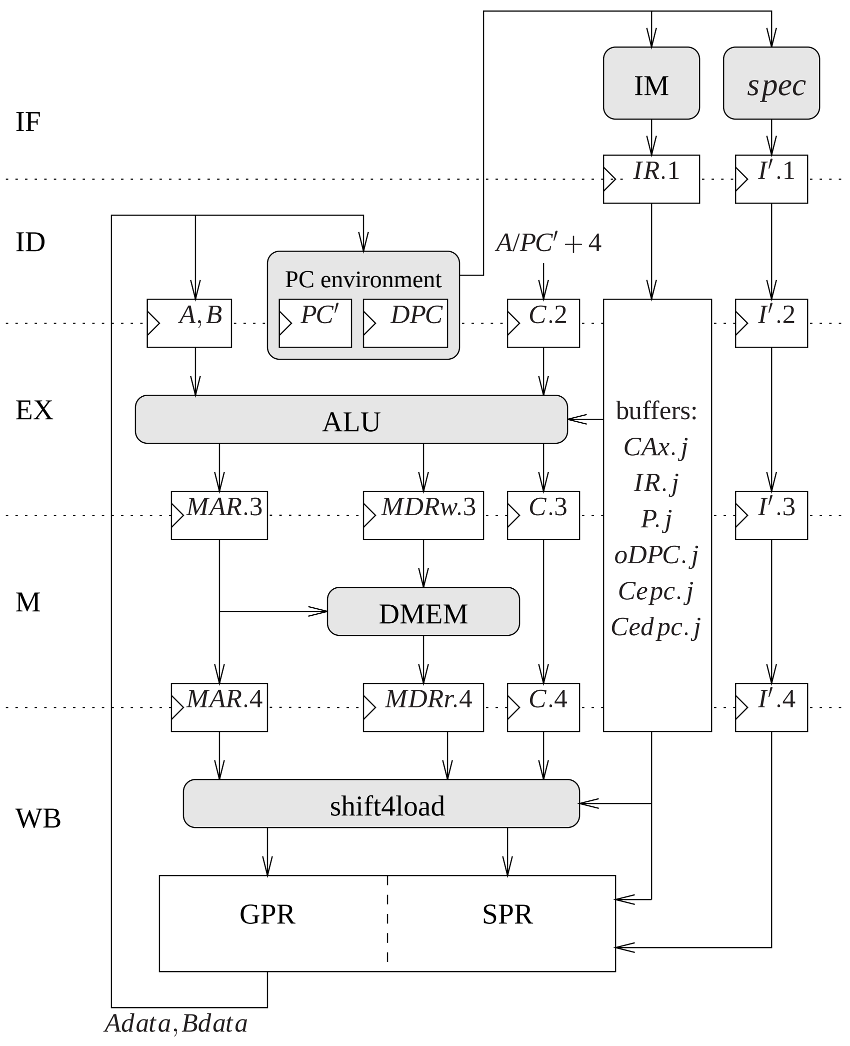


Figure 5.16 Overview of the DLX pipeline with precise interrupts. The registers I' are a shorthand for the speculative values $JISR$ and $repeat$. The $spec$ environment does the speculation.

the definition of $I(c)$ as given in the specification:

$$f_0IR(DPC,ipf) \quad = \quad \begin{cases} 0 & : \ imal(DPC) \vee ipf \\ IM[DPC] & : \ \text{otherwise} \end{cases}$$

Lemma 5.67 ▶ The calculation of the instruction word is correct:

$$\Omega_0IR(c_S^i) \quad = \quad I(c_S^i)$$

PROOF If one expands the left hand side of the claim, one gets:

$$f_0IR(c_S^i.DPC, ipf(c_S^i)) \quad \overset{!}{=} \quad I(c_S^i)$$

One easily asserts this claim by expanding f_0IR on the left hand side and I on the right hand side.
QED

Exceptions We collect the interrupt cause bits CA in separate implementation registers. In the register $CAimal$, we store whether we have an instruction word misalignment. The same applies for $CAipf$ and $CAex$. We assume an external signal ex that is a bitvector. The bits of the bitvector indicate the individual external interrupts. In contrast to [MP00], we detect the external interrupts in stage 0.

$$f_0CAimal(DPC) \quad = \quad imal(DPC) \tag{5.18}$$
$$f_0CAipf(ipf) \quad = \quad ipf \tag{5.19}$$
$$f_0CAex(ex) \quad = \quad ex \tag{5.20}$$

In addition to that, we speculate two values $JISR$ and $repeat$. We speculate that we have an interrupt if we have an instruction memory page fault, a trap instruction, a misaligned instruction word, or an external interrupt. We detect external interrupts using a zero tester. Remember that the function used for speculating R is called f_0Rs. Thus, the function for speculating $JISR$ is:

$$\begin{aligned} f_0JISRs(ipf,DPC,ex) \quad = \quad & ipf \vee I\_trap(f_0IR(DPC,ipf)) \vee \\ & imal(DPC) \vee \overline{zerotester(ex)} \end{aligned}$$

We speculate that we have an interrupt of type repeat if an instruction memory page fault is signaled and if there is no instruction word misalignment:

$$f_0repeats(ipf,DPC) \quad = \quad ipf \vee \overline{imal(DPC)}$$

This implementation differs from the implementation given in [MP00]: in [MP00], the execution is started always assuming that no interrupt happens. This includes the interrupts that can be detected in early stages.

As an example, consider a *trap* instruction. The machine in [MP00] executes the instructions followed by the *trap* instruction as if no interrupt happens. In stage 3, the misspeculation is detected and the instructions following *trap* are evicted from the pipeline. In contrast to that, the machine presented here never misspeculates on *trap* instructions. Thus, no rollback is necessary. Following the *trap* instruction, the instructions of the interrupt service routine are executed. We therefore waste no cycles in case of the interrupts given above.

Obviously, this speeds up execution. The price paid for this is extra complexity. In particular, we have to forward the effect of interrupts. This includes that interrupts modify all special purpose registers. In [MP00], the authors remark that "forwarding the effect of this looks like a nightmare". We will later on describe the forwarding hardware we use for this.

5.12.6 Transition Functions of Stage 1

In stage 1, we do the operand fetching, the calculation of the new PC registers, and the calculation of the precomputed control signals. As in chapter 3, let us define the precomputed control signals in the stages that use them.

PC' In order to calculate the new value of the PC' register, we implement the function $nextpc'$ as given by the specification as follows:

- In case of an rfe instruction, we take the value of the EPC input.
- In case of any other instruction, we use the value provided by the old $nextpc\_impl$ circuit, as defined in chapter 3.

Thus, $nextpc'\_impl$ is:

$$nextpc'\_impl(IR, GPRa, oldPC, EPC) :=$$
$$\begin{cases} EPC & : \quad I\_rfe(IR) \\ nextpc\_impl(IR, GPRa, oldPC, EPC) & : \quad \text{otherwise} \end{cases}$$

The following lemma asserts that the circuit $nextpc'\_impl$ complies with the specification $nextpc'$.

Lemma 5.68 ▶ The calculation of the new PC is correct:

$$nextpc'\_impl \quad = \quad nextpc'$$

PROOF If $I\_rfe(I)$ holds, the claim obviously holds. If not so, one asserts the claim using lemma 3.4.

Lemma 5.69 ▶ In the specification, we pass $op1(c_S^i)$ as parameter to $nextpc$. In the implementation, we pass $GPRa$ as input. Given that this input is correct, the $nextpc$ function returns the same value in both cases.

$$nextpc(I(c_S^i), G_1GPRa(c_S^i), c_S^i.PC') \quad = \quad nextpc(I(c_S^i), op1(c_S^i), c_S^i.PC')$$

PROOF One asserts this claim in analogy to the proof of lemma 3.15 (correctness of the transition functions of the sequential DLX).

For $nextpc'\_impl$, we need the value of EPC in case of an rfe instruction. We realize this by a conditional read access to EPC. The read enable function returns true iff we have an rfe instruction:

$$f_1EPCre(IR) \quad = \quad I\_rfe(IR) \tag{5.21}$$

This allows defining the register transition function for PC' in analogy to the specification:

$$f_1PC'(IR, JISR, PC', EPC, GPRa) = \\ \begin{cases} SISR + 4 & : \; JISR \\ nextpc'\_impl(IR, GPRa, PC', EPC) & : \; \text{otherwise} \end{cases}$$

Lemma 5.70 ▶ Assuming correct inputs, the calculation of the new value of PC' is correct:

$$c_S^{i+1}.PC' \quad = \quad f\Gamma_1PC'(c_S^i)$$

PROOF By expanding the definition of c_S^{i+1} on the left hand side, we get:

$$\delta.PC'(c_S^i) \quad \overset{!}{=} \quad f\Gamma_1PC'(c_S^i)$$

The function f_1PC' uses $JISR$ as input. The correct value of the $JISR$ input given configuration c_S^i is:

$$G_1JISR(c_S^i) \quad = \quad JISR(c_S^i)$$

Let $JISR(c_S^i)$ hold. In this case, both f_1PC' and $\delta.PC'$ return $SISR+4$ and the claim holds.

Let $JISR(c_S^i)$ not hold. In this case, we assert the correctness of the GPR operand as in the proof of lemma 3.15, which is the corresponding lemma for the machine without interrupts. We then apply lemma 5.68, which concludes the claim.

QED

DPC For defining the register transition function for DPC, we need the register $EDPC$ for rfe instructions. As above, we realize this by a conditional read access to $EDPC$. The read enable function returns true iff we have an rfe instruction:

$$f_1EDPCre(IR) \quad = \quad I\_rfe(IR) \tag{5.22}$$

This allows defining the register transition function for DPC in analogy to the specification:

$$f_1DPC(IR,JISR,PC',EDPC) \quad = \quad \begin{cases} SISR & : \quad JISR \\ EDPC & : \quad I\_rfe(IR) \\ PC' & : \quad \text{otherwise} \end{cases}$$

The following lemma asserts the correctness of this circuit.

◄ Lemma 5.71

Assuming correct inputs, the calculation of the new value of DPC is correct:

$$c_S^{i+1}.DPC \quad = \quad f\Gamma_1DPC(c_S^i)$$

PROOF

By expanding the definition of $c_S^{i+1}.DPC$ on the left hand side, we get:

$$\delta.DPC(c_S^i) \quad \overset{!}{=} \quad f\Gamma_1DPC(c_S^i)$$

By expanding the definition of $f\Gamma_1DPC$ on the right hand side, we get:

$$\delta.DPC(c_S^i) \quad \overset{!}{=} \quad f_1DPC(\Omega_0IR(c_S^i),JISR(c_S^i),c_S^i.PC',c_S^i.EDPC)$$

By applying lemma 5.67, we get:

$$\delta.DPC(c_S^i) \overset{!}{=} f_1DPC(I(c_S^i), JISR(c_S^i), c_S^i.PC', c_S^i.EDPC)$$

QED One easily asserts this by expanding the functions $\delta.DPC$ and f_1DPC.

We precompute the values to be written into the special purpose registers EPC and $EDPC$. This saves hardware cost, since this computation depends on many registers. Furthermore, it allows forwarding these registers. This includes the effect of interrupts. As the register C is responsible for forwarding GPR registers, the registers $Cepc$ and $Cedpc$ are responsible for forwarding EPC and $EDPC$. The new values are already available in the decode/issue stage. Thus, the write condition is always true.

The new value of EPC is precomputed as follows: In case of an interrupt of type repeat, we write PC'. In case of any other interrupt, we write the new value of PC' without interrupt, which is given by $nextpc_i'mpl$. In case there is no interrupt, we return $GPRa$ in order to handle $movi2s$ with EPC as destination.

$$f_1Cepc(IR, JISR, repeat, GPRa, PC', EPC) =$$

$$\begin{cases} PC' & : \; JISR \wedge repeat \\ nextpc_i'mpl(IR, GPRa, PC', EPC) & : \; JISR \wedge \overline{repeat} \\ GPRa & : \; \text{otherwise} \end{cases}$$

$$f_1Cepcwe(IR, JISR) = 1$$

The write enable signal of EPC is precomputed as follows: we write to EPC in case of a $movi2s$ with appropriate destination and in case of an interrupt.

$$f_1f_4EPCwe(IR, JISR) = JISR \vee (I\_movi2s(IR) \wedge \langle I\_immediate(IR)[4:0]\rangle = 3)$$

The new value of *EDPC* is precomputed as follows: in case of an interrupt of type repeat, we write *DPC*. In case of any other interrupt and an *rfe* instruction, we write *EDPC*. In case of any other interrupt and any other instruction, we write *PC'*. In case there is no interrupt, we return *GPRa* in order to handle *movi*2*s* with *EDPC* as destination.

$$f_1Cedpc(IR, JISR, repeat, GPRa, DPC, EDPC, PC') =$$

$$\begin{cases} DPC & : & JISR \wedge repeat \\ EDPC & : & JISR \wedge \overline{repeat} \wedge I\_rfe(IR) \\ PC' & : & JISR \wedge \overline{repeat} \wedge \overline{I\_rfe(IR)} \\ GPRa & : & \text{otherwise} \end{cases}$$

The write enable signal of *EDPC* is precomputed as follows:

$$f_1 f_4 EDPCwe(IR, JISR) \quad = \quad JISR \vee (I\_movi2s(IR) \wedge \langle I\_immediate(IR)[4:0] \rangle = 4)$$

We will show the correctness of these values when we describe the transition functions of stage 4.

Forwarding Logic for EPC/EDPC Using these precomputed values, we get the following forwarding hardware for reading *EPC* and *EDPC* in stage $k = 1$: We show this exemplary for *EPC*. The circuits for *EDPC* are identical. As before, we calculate hit signals $R_khit[j]$. Thus, the signals are named $EPC_1hit[j]$. The hit signal is active iff the full bit of stage j and the precomputed write enable signal of *EPC* in stage j are active:

$$EPC_1hit[j](c_I) \quad := \quad full_j(c_I) \wedge f_4EPCwe.j$$

Using the hit signals, we calculate the forwarded value. This is done using multiplexers, as illustrated in figure 5.17. The proof correctness of this logic is similar to the proof of correctness for forwarding *GPR* registers. However, we do not have to argue about an address.

Note that we need only very little effort in order to realize an instruction fetch with interrupts. In particular, we only need a few arguments in order to show correctness as we only instantiate the generic forwarding and speculation mechanisms.

In stage 1, we fetch the operands. This is done exactly as in chapter 3 with the exception that we need the first GPR operand also in case of a *movi*2*s* instruction.

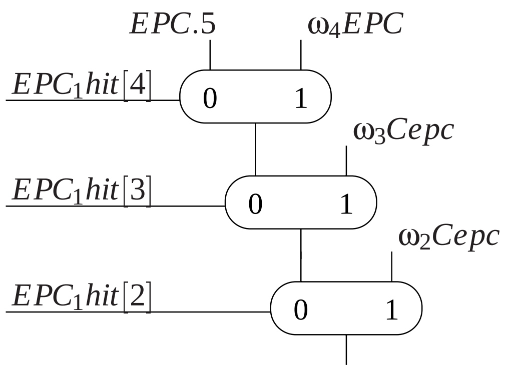


Figure 5.17 Implementation of EPC forwarding

Note that we do not fetch the source operand of *movs2i* instructions in stage 1 in contrast to the machine presented in [MP00]. We do so in order to illustrate read accesses to registers other than in the decode stage. This has both advantages and disadvantages: obviously, we save the forwarding logic. The disadvantage is that an instruction that follows the *movs2i* and uses the destination of the *movs2i* as source has to be stalled. However, we do not see a severe performance impact of doing so. Furthermore, if one desires forwarding, our generic forwarding approach will generate appropriate forwarding hardware.

Exceptions In stage 1, we decode the instruction word and signal an illegal instruction word exception if necessary. Given an instruction word IR, the function $ill(IR)$ indicates that it is illegal. Let $\mathtt{I}$ be the set of instructions. The function ill is defined using the predicates I_x as defined in chapter 2:

$$ill(IR) \quad := \quad \overline{\bigvee_{x \in \mathtt{I}} I\_x} \tag{5.23}$$

We store this bit in an implementation register $CAill$:

$$f_1 CAill(IR) \quad = \quad ill(IR) \tag{5.24}$$

We do not do this in stage 0 because the calculation of $ill(IR)$ might get slow in case that there are many instructions. Furthermore, we consider illegal instruction exceptions to be rare. Thus, the price for misspeculation is not often paid.

5.12.7 Transition Functions of Stage 2

In this stage, we do the ALU calculation. The transition functions from chapter 3 are taken without modification. We store a bit indicating an ALU overflow in a register $CAovf$:

$$f_2CAovf(IR,A,B) \quad = \quad ALU(A,aluop2(IR,B),aluf(IR)).ovf$$

The functions $aluop2$ and $aluf$ are taken from chapter 3.

5.12.8 Transition Functions of Stage 3

In this stage, we do the data memory access. Most transition functions from chapter 3 are taken without modification. We store a bit indicating a data memory page fault in a register $CAdpf$:

$$f_3CAdpf(IR,dpf) \quad = \quad dpf \wedge (I\_load(IR) \vee I\_store(IR))$$

We store a bit indicating a data memory misalignment exception:

$$f_3CAdmal(IR,MAR) \quad = \quad dmal(IR,MAR)$$

Furthermore, we do not enable the data memory write enable signal in case we have one of these exceptions.

Cause Collection In stage 3, all exceptions are now known. This allows us to calculate the MCA register: we do this by reading all CA registers and calculating CA. As a shorthand, let $CAargs$ denote the list of arguments used in order to calculate CA (in the PVS tree, we always use the expanded form). This is:

$$CAargs := (IR,CAill,CAimal,MAR,CAipf,dpf,CAtrap,CAovf,CAex)$$

The function *CA_impl* takes these inputs and provides *CA*:

$$CA\_impl(CAargs)[i] := \begin{cases} 0 & : \; i=0 \\ CAill & : \; i=1 \\ CAimal \vee dmal(IR, MAR) & : \; i=2 \\ CAipf & : \; i=3 \\ (I\_load(IR) \vee I\_store(IR)) \wedge dpf & : \; i=4 \\ CAtrap & : \; i=5 \\ CAovf & : \; i=6 \\ CAex(i-7) & : \; \text{otherwise} \end{cases}$$

Using the CA bits, we calculate *MCA* using the *MCA_impl* circuit:

$$f_3MCA(SR, CAargs) \quad = \quad MCA\_impl(CA\_impl(CAargs), SR)$$

Forwarding Logic for SR The register transition function f_3MCA depends on *SR*, i.e., we have a read access to *SR*.4 in stage 3. This requires forwarding. The forwarding mechanism described in the previous chapter (forwarding from the next stage, page 101) generates the following hardware (the definition of the function ω_4SR is expanded):

$$g_3(c_I) \quad = \quad \begin{cases} f\gamma_4SR(c_I) & : \; full_4(c_I) \wedge f\gamma_4SRwe(c_I) \\ c_I.SR.4 & : \; \text{otherwise} \end{cases}$$

Thus, in case stage 4 is full and the write enable signal of *SR*.4 is active, we use the value written into *SR*.4. This holds in particular if there is an instruction in stage 4 that causes an interrupt or is a *movi2s* instruction writing *SR*.

In any other case, we use the value in the register *SR*.4. The proof that this is the correct input is given in chapter 4 (lemma 4.7).

The following lemma asserts that the implementation register *MCA* contains the correct value, as defined using the configuration of the specification machine.

Lemma 5.72 ▶ The calculation of the next value of MCA is correct:

$$\Omega_3MCA(c_S^i) \quad = \quad MCA(c_S^i)$$

PROOF By expanding the function $\Omega_3 MCA(c_S^i)$ on the left hand side, we get (we omit the parameter list):

$$f_3 MCA(\ldots) \overset{!}{=} MCA(c_S^i)$$

By definition of $f_3 MCA$, we get:

$$MCA\_impl(CA\_impl(G_3(c_S^i, CAargs)), c_S^i.SR) \overset{!}{=} MCA(c_S^i)$$

By applying lemma 5.64, we get:

$$MCA\_impl(CA\_impl(\ldots), c_S^i.SR) \overset{!}{=} MCA\_impl(CA(c_S^i), c_S^i.SR)$$

Thus, the claim is shown if $CA\_impl(\ldots)$ is equal to $CA(c_S^i)$. We show this by a case split on the number of the exception, i.e., we show

$$CA\_impl(G_3(c_S^i, CAargs))[i] \overset{!}{=} CA(c_S^i)[i]$$

for all $i \in \{0,\ldots,31\}$. We show the claim exemplary for the external interrupts. The proofs for the other exceptions follow the same pattern.

For $i \geq 7$ (external interrupts), we have the following claim:

$$\Omega_2 CAex(c_S^i)[i] \overset{!}{=} ex_S(c_S^i)$$

By expanding the functions on the left hand side, we get:

$$\begin{aligned} \Omega_1 CAex(c_S^i)[i] &\overset{!}{=} ex_S(I(c_S^i)) \\ \Omega_0 CAex(c_S^i)[i] &\overset{!}{=} ex_S(I(c_S^i)) \\ f_0 CAex(ex_S(c_S^i))[i] &\overset{!}{=} ex_S(I(c_S^i)) \end{aligned}$$

This is concluded by expanding $f_0 CAex$. QED

Detecting Misspeculation Using MCA, we can also calculate the correct value of $JISR$ and $repeat$, thus, we can detect any misspeculation in stage 3. In case of $JISR$, we use the $JISR\_impl$ circuit as defined above.

◀ Lemma 5.73

The new value of $JISR$ is correct.

$$f\Gamma_3 JISR(c_S^i) = JISR(c_S^i)$$

One asserts this lemma using lemma 5.72 (correctness of *MCA*) and lemma 5.65.

In case of the *repeat* register, we calculate the correct value using the circuit *repeat_impl*.

Lemma 5.74 ▶ The new value of *repeat* is correct.

$$f\Gamma_3 repeat(c_S^i) \quad = \quad repeat(c_S^i)$$

We assert this lemma using lemma 5.72, which shows the correctness of *MCA*, and lemma 5.66.

5.12.9 Transition Functions of Stage 4

In analogy to lemma 5.67, one easily shows that *IR* read in stage 4 is the instruction word:

Lemma 5.75 ▶ The calculation of the instruction word is correct:

$$\Omega_3 IR(c_S^i) \quad = \quad I(c_S^i)$$

Write Access to GPR In this stage, the result of the instructions is written into the destination register. In case of ALU instructions or load instructions, we do this as in chapter 3. However, we have to change the transition function of *GPR* in order to realize *movs2i*.

As described above, we read the source operand in stage 4. We just pass the values of the special purpose registers as parameters to the transition function. After that, we use a decoder (definition 5.1) in order to generate select signals for multiplexers. Let *decoder_impl* be an implementation of a decoder according to the definition.

$$SAdec(IR) \quad = \quad decoder\_impl(I\_immediate(IR)[4:0])$$

We define a shorthand *SPRsrc*, which denotes the value of the SPR source operand:

$$SPRsrc(IR, SR, \ldots, EDATA) =$$

$$\begin{cases} SR & : SAdec(IR)[0] \\ ESR & : SAdec(IR)[1] \\ ECA & : SAdec(IR)[2] \\ EPC & : SAdec(IR)[3] \\ EDPC & : SAdec(IR)[4] \\ EDATA & : SAdec(IR)[5] \\ 0 & : \text{otherwise} \end{cases}$$

This allows defining the register transition function for *GPR*:

$$f_4GPR(C, IR, MAR, MDRr, SR, \ldots, EDATA) =$$

$$\begin{cases} shift4load(MAR, MDRr, IR) & : I\_load(IR) \\ SPRsrc(IR, SR, \ldots, EDATA) & : I\_movs2i(IR) \\ C & : \text{otherwise} \end{cases}$$

In addition to that, we modify the precomputed version of the write enable signal f_4GPRwe such that it is active in case of a *movs*2*i* instruction. Furthermore, we disable it in case of an interrupt of type repeat, as indicated by *repeat*:

$$\begin{aligned} f_1 f_4GPRwe(IR, repeat) \quad = \quad & (I\_ALU(IR) \vee I\_ALUi(IR) \vee I\_load(IR) \vee \\ & I\_shifti(IR) \vee I\_shift(IR) \vee I\_movs2i(IR) \\ & \vee((I\_j(IR) \vee I\_jr(IR)) \wedge I\_link(IR))) \\ & \wedge \overline{repeat} \end{aligned}$$

One easily asserts the correctness of the f_4GPR function in analogy to the proof of lemma 3.15.

In addition to the write access to *GPR*, we also have the write accesses to the special purpose registers in stage 4.

Write Access to SR We perform a conditional write access to *SR*: in case of an interrupt, as indicated by *JISR*, we write zero. In case of an *rfe* instruction, we write ESR. Otherwise, we have a $movi2s$ instruction and write the value in the C register, which is the GPR operand:

$$f_4SR(C,IR,JISR,ESR) = \begin{cases} 0 & : JISR \\ ESR & : I\_rfe(IR) \\ C & : \text{otherwise} \end{cases}$$

$$\begin{aligned} f_4SRwe(IR,JISR) &= JISR \lor I\_rfe(IR) \lor \\ &\quad (I\_movi2s(IR) \land SAdec(IR)[0]) \end{aligned}$$

Lemma 5.76 ▶ The correct value of C matches the GPR operand in case of a $movi2s$ instruction.

$$I\_movi2s(I(c_S^i)) \implies \Omega_3C(c_S^i) = op1(c_S^i)$$

PROOF Because we have a $movi2s$ instruction, we have $\Omega_3C(c_S^i) = G_1GPRa(c_S^i)$. This transforms the claim into:

$$I\_movi2s(I(c_S^i)) \implies G_1GPRa(c_S^i) = op1(c_S^i)$$

One concludes this claim by expanding the definition of G_1GPRa on the right hand side. QED

Lemma 5.77 ▶ The value written by f_4SR is correct.

$$c_S^{i+1}.SR = \begin{cases} f\Gamma_4SR(c_S^i) & : f\Gamma_4SRwe(c_S^i) \\ c_S^i.SR & : \text{otherwise} \end{cases}$$

PROOF Let us expand the definition of the write enable signal $f\Gamma_4SRwe$:

$$\begin{aligned} f\Gamma_4SRwe(c_S^i) &= c_S^i.JISR \lor I\_rfe(\Omega_3IR(c_S^i)) \lor \\ &\quad (I\_movi2s(\Omega_3IR(c_S^i)) \land SAdec(\Omega_3IR(c_S^i))[0]) \end{aligned}$$

By applying lemma 5.75, this is transformed into:

$$\begin{aligned} f\Gamma_4SRwe(c_S^i) &= c_S^i.JISR \lor I\_rfe(I(c_S^i)) \lor \\ &\quad (I\_movi2s(I(c_S^i)) \land SAdec(I(c_S^i))[0]) \end{aligned}$$

Using the correctness of the decoder circuit, this is transformed into:

$$\begin{aligned} f\Gamma_4 SRwe(c_S^i) &= c_S^i.JISR \vee I\_rfe(I(c_S^i)) \vee \\ &\quad (I\_movi2s(I(c_S^i)) \wedge \langle I\_immediate(I(c_S^i))[4:0]\rangle = 0 \end{aligned}$$

By expanding the definition of c_S^{i+1} on the left hand side of the claim (as given in lemma 5.77), we get:

$$\delta.SR(c_S^i) \overset{!}{=} \begin{cases} f\Gamma_4 SR(c_S^i) & : \quad f\Gamma_4 SRwe(c_S^i) \\ c_S^i.SR & : \quad \text{otherwise} \end{cases}$$

Let the write enable signal $f\Gamma_4 SRwe(c_S^i)$ be not active. In this case, one easily asserts the claim by expanding the definition of $\delta.SR$.

Let the write enable signal $f\Gamma_4 SRwe(c_S^i)$ be active. By expanding the definition of $f\Gamma_4 SR$, we get:

$$\delta.SR(c_S^i) \overset{!}{=} f_4 SR(\Omega_3 C(c_S^i), \Omega_3 IR(c_S^i), JISR(c_S^i), c_S^i.ESR)$$

Using lemma 5.75, we get:

$$\delta.SR(c_S^i) \overset{!}{=} f_4 SR(\Omega_3 C(c_S^i), I(c_S^i), JISR(c_S^i), c_S^i.ESR)$$

By expanding the definition of $f_4 SR$, we get:

$$\delta.SR(c_S^i) \overset{!}{=} \begin{cases} 0 & : \quad JISR(c_S^i) \\ c_S^i.ESR & : \quad I\_rfe(I(c_S^i)) \\ \Omega_3 C(c_S^i) & : \quad \text{otherwise} \end{cases}$$

In case of $JISR(c_S^i)$ or $I\_rfe(I(c_S^i))$ the claim holds by definition of $\delta.SR$. In any other case, we can conclude that we have a *movi2s* instruction because the write enable signal is active. This allows applying lemma 5.76 and we get:

$$\delta.SR(c_S^i) \overset{!}{=} op1(c_S^i)$$

This is concluded by expanding the definition of $\delta.SR$. QED

Write Access to ESR We perform a conditional write access to ESR: in case of an interrupt of type repeat, as indicated by $JISR$ and $repeat$, we write SR. In case of any other interrupt, we write C if we have a $movi2s$ instruction that uses SR as destination, and SR otherwise. In case there is no interrupt, we return C in order to handle $movi2s$ with ESR as destination.

$$f_4ESR(C,IR,JISR,SR,repeat) = \begin{cases} C & : \; sel \\ SR & : \; \text{otherwise} \end{cases}$$

with a signal sel in analogy to [MP00]:

$$sel = \overline{JISR} \vee (\overline{repeat} \wedge I\_movi2s(IR) \wedge SAdec(IR)[0])$$

The write enable signal is active in case of an interrupt or a $movi2s$ instruction with destination ESR.

$$\begin{aligned} f_4ESRwe(IR,JISR) &= JISR \vee \\ &\quad (I\_movi2s(IR) \wedge SAdec(IR)[1]) \end{aligned}$$

Lemma 5.78 ▶ The value written by f_4ESR is correct.

$$c_S^{i+1}.ESR = \begin{cases} f\Gamma_4ESR(c_S^i) & : \; f\Gamma_4ESRwe(c_S^i) \\ c_S^i.ESR & : \; \text{otherwise} \end{cases}$$

PROOF The proof proceeds in analogy to the proof of lemma 5.77.

Write Access to ECA We perform a conditional write access to ECA: in case of an interrupt, we write MCA. In case there is no interrupt, we return C in order to handle $movi2s$ with ECA as destination.

$$f_4ESR(C,IR,JISR,MCA) = \begin{cases} MCA & : \; JISR \\ C & : \; \text{otherwise} \end{cases}$$

$$\begin{aligned} f_4ECAwe(IR,JISR) &= JISR \vee \\ &\quad (I\_movi2s(IR) \wedge SAdec(IR)[2]) \end{aligned}$$

Lemma 5.79 ▶ The value written by f_4ECA is correct.

$$c_S^{i+1}.ECA = \begin{cases} f\Gamma_4ECA(c_S^i) & : \; f\Gamma_4ECAwe(c_S^i) \\ c_S^i.ECA & : \; \text{otherwise} \end{cases}$$

PROOF The proof proceeds in analogy to the proof of lemma 5.77. However, we need the correctness of the MCA input, which we assert using lemma 5.72.

Write Access to EPC We perform a conditional write access to EPC. We already precomputed the value to be written and the write enable signal in stage 1.

◀ Lemma 5.80

The value written by f_4EPC is correct.

$$c_S^{i+1}.EPC \quad = \quad \begin{cases} f\Gamma_4EPC(c_S^i) & : \quad f\Gamma_4EPCwe(c_S^i) \\ c_S^i.EPC & : \quad \text{otherwise} \end{cases}$$

PROOF

As in the proof of lemma 5.77, let us expand the definition of the write enable signal $f\Gamma_4EPCwe$ (including the functions used to pass the precomputed signals):

$$\begin{aligned} f\Gamma_4EPCwe(c_S^i) \quad = \quad & c_S^i.JISR \vee \\ & (I\_movi2s(\Omega_1IR(c_S^i)) \wedge \\ & \langle I\_immediate(\Omega_1IR(c_S^i))[4:0]\rangle = 3) \end{aligned}$$

One easily asserts $\Omega_1IR(c_S^i) = I(c_S^i)$, which transforms the last equation into:

$$\begin{aligned} f\Gamma_4EPCwe(c_S^i) \quad = \quad & c_S^i.JISR \vee \\ & (I\_movi2s(I(c_S^i)) \wedge \langle I\_immediate(I(c_S^i))[4:0]\rangle = 3) \end{aligned}$$

Using the correctness of the decoder circuit, this is transformed into:

$$\begin{aligned} f\Gamma_4EPCwe(c_S^i) \quad = \quad & c_S^i.JISR \vee \\ & (I\_movi2s(I(c_S^i)) \wedge \langle I\_immediate(I(c_S^i))[4:0]\rangle = 3) \end{aligned}$$

By expanding the definition of c_S^{i+1} on the left hand side of the claim (as given in lemma 5.80), we get:

$$\delta.EPC(c_S^i) \quad \overset{!}{=} \quad \begin{cases} f\Gamma_4EPC(c_S^i) & : \quad f\Gamma_4EPCwe(c_S^i) \\ c_S^i.EPC & : \quad \text{otherwise} \end{cases}$$

Let the write enable signal $f\Gamma_4EPCwe(c_S^i)$ be not active. In this case, one easily asserts the claim by expanding the definition of $\delta.EPC$.

Let the write enable signal $f\Gamma_4EPCwe(c_S^i)$ be active. By expanding the definition of $f\Gamma_4EPC$ (including the functions that pass the precomputed value), we get:

$$\begin{aligned} \delta.EPC(c_S^i) \quad \overset{!}{=} \quad & f_1Cepc(\Omega_1IR(c_S^i), JISR(c_S^i), repeat(c_S^i), \\ & G_1GPRa(c_S^i), c_S^i.PC', G_1EPC(c_S^i)) \end{aligned}$$

By expanding the definition of f_1Cepc, we get:

$$\delta.EPC(c_S^i) \overset{!}{=} \begin{cases} c_S^i.PC' & : \; JISR(c_S^i) \wedge repeat(c_S^i) \\ newpc'\_impl(\ldots) & : \; JISR(c_S^i) \wedge \overline{repeat(c_S^i)} \\ G_1GPRa(c_S^i) & : \; \text{otherwise} \end{cases}$$

We handle the three cases above separately:

1. In case of an interrupt of type repeat, one concludes the claim by expanding the definition of $\delta.SR$.
2. In case of any other interrupt, the claim is transformed into (we omit the parameter list):

   $$\delta.EPC(c_S^i) \overset{!}{=} newpc'\_impl(\ldots)$$

   By expanding the definition of $\delta.EPC(c_S^i)$ on the left hand side and by applying lemma 5.68, one gets:

   $$\begin{aligned} & nextpc'(I(c_S^i), op1(c_S^i), c_S^i.PC', c_S^i.EPC) \\ \overset{!}{=} \; & nextpc'(I(c_S^i), G_1GPRa(c_S^i), c_S^i.PC', G_1EPC(c_S^i)) \end{aligned}$$

   In case we have an rfe instruction, one asserts that $G_1EPC(c_S^i)$ (correct value if reading EPC) is equal to $c_S^i.EPC$ because the read enable function holds. The claim is then easily concluded by expanding the definition of $nextpc'$.

   In case we do not have an rfe instruction, we conclude the claim by expanding the definition of $nextpc'$ and by applying lemma 5.69.
3. In case we do not have an interrupt, we can conclude that we have a $movi2s$ instruction because the write enable signal is active. In this case, the claim is easily concluded by expanding G_1GPRa.

QED

Write Access to EDPC We perform a conditional write access to $EDPC$: We already precomputed the value to be written and the write enable signal in stage 1.

Lemma 5.81 ► The value written by f_4EDPC is correct.

$$c_S^{i+1}.EDPC = \begin{cases} f\Gamma_4EDPC(c_S^i) & : \; f\Gamma_4EDPCwe(c_S^i) \\ c_S^i.EDPC & : \; \text{otherwise} \end{cases}$$

The proof proceeds as the proof of lemma 5.80.

Write Access to EDATA In stage 4, we perform a conditional write access to *EDATA*: in case of a data memory page fault interrupt, we write *MAR*. In case of a trap instruction, we write the immediate constant. In case of any other interrupt, we write zero. In case there is no interrupt, we return *C* in order to handle *movi2s* with *EDATA* as destination.

$$f_4EDATA(C, IR, JISR, CAdpf, CAtrap, MAR) =$$

$$\begin{cases} MAR & : JISR \wedge CAdpf \\ I\_immediate(IR) & : JISR \wedge \overline{CAdpf} \wedge CAtrap \\ 0 & : JISR \wedge \overline{CAdpf} \wedge \overline{CAtrap} \\ C & : \text{otherwise} \end{cases}$$

$$f_4EDATAwe(IR, JISR) \quad = \quad JISR \vee (I\_movi2s(IR) \wedge SAdec(IR)[5])$$

The following lemma asserts the correctness of the transition function for *EDATA*.

The value written by f_4EDATA is correct. ◄ Lemma 5.82

$$c_S^{i+1}.EDATA \quad = \quad \begin{cases} f\Gamma_4EDPC(c_S^i) & : \ f\Gamma_4EDATAwe(c_S^i) \\ c_S^i.EDATA & : \ \text{otherwise} \end{cases}$$

The proof proceeds as the proof of lemma 5.77. However, we use *MAR* as input in case of a data memory page fault.

5.12.10 Data Consistency and Liveness

One concludes the data consistency and liveness of the pipelined machine with interrupts just as we concluded the data consistency of the pipelined machine with branch prediction.

Note that in particular PVS almost fully automates the proofs for the lemmas given above in order to show the pipelined machine with speculation.

5.13 Literature

In the open literature, speculation is a common approach for implementing processors without delay slot: Levitt et.al. use a predict-not-taken scheme [LO96] in a DLX implementation. Boerger and Mazzanti provide two DLX implementations [BM96]: the first assumes an empty instruction after jumps/branches. The second implementation stalls the instruction fetch for one cycle. Saxe et.al. [SGGH94] also use speculation.

In [VB00], Velev and Bryant extend Burch and Dill's pipeline flushing technique in order to automatically verify a dual-instruction issue, in-order DLX with five stages and branch prediction. Misspedicted branches are detected late, A generic speculation approach or a stall engine is not used.

Chapter 6

Out-of-Order Execution

6.1 Introduction

IN THE PREVIOUS SECTIONS, we presented various implementations of pipelined RISC processors. These implementations strictly processed the instructions in program order. However, the performance of these designs drops as soon as long latency instructions such as memory accesses are involved. For example, consider a load instruction with cache miss in the memory stage. Thus, the stall signal of the stage is activated and the instructions above the memory stage are stalled.

Furthermore, consider an ALU instruction that follows the load in the execute stage:

```
EX:      R3:=R1+R2
M:       R4:=Mem[R5]
```

If there is no data dependency, the result of the ALU instruction is already known in the execute stage and could be written into the register file. However, the in-order execution rule prohibits this and the ALU instruction has to wait for the load.

Thus, dropping this rule can result in better performance. This technique is called *out-of-order execution*. The most popular out-of-order execution

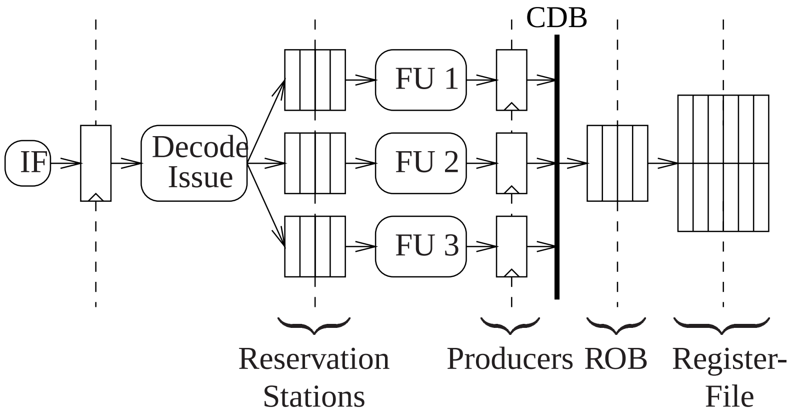


Figure 6.1 Basic structure of a microprocessor with Tomasulo Scheduler and reorder buffer

algorithms is the Tomasulo scheduling algorithm [Tom67]. It is one of the most competitive scheduling algorithms and provides CPI rates down to 1.1 on a single-instruction issue machine [Ger98, Del98, MLD$^+$99]. The algorithm is widely used, e.g., by IBM PowerPC, Intel Pentium-Pro or AMD K5 [Mot97, CS95]. The original Tomasulo scheduler uses out-of-order termination and therefore does not support precise interrupts without extra hardware. We support precise interrupts by adding a *reorder buffer*[SP88]. The reorder buffer sorts the instructions in program order before termination.

In this chapter, we describe the results of implementing and verifying a DLX with Tomasulo scheduler, precise interrupts and floating point unit using PVS. The designs, the scheduling protocols, and most proofs are taken from [KMP99, Krö99].

6.2 The Tomasulo Algorithm with Reorder Buffer

Figure 6.1 depicts the basic structure of a microprocessor with Tomasulo scheduler and reorder buffer. The execution begins with the instruction fetch, as in the in-order machine. The Tomasulo scheduling algorithm does not cover this phase; it is assumed that the instruction fetch is done in program order. We will use the very same instruction fetch mechanism as in the pipelined in-order machines described in the previous chapters.

In the next stage, the instruction is decoded. This includes fetching the operands if available. The instruction and the operands are then passed to a *reservation station* (RS). This is called *issue*. The reservation stations are the central data structure of the Tomasulo scheduling algorithm. The reservation stations act as queue for the instructions and are between the decode/issue stage and the functional units. Note that the instruction is passed to the reservation station even if forwarding fails. This is in contrast to the in-order machine, which stalls in this case.

As soon as all operands are available, the instruction is passed from the reservation station to the functional unit. This is called *dispatch*. This is done without obeying the program order of the instructions, i.e., the instructions can overtake each other at this point. After the function unit has finished the execution, the result of the instruction is passed to a special register, called *producer*.

In case the producer holds an instruction, it requests a result bus, called *common data bus* (CDB). As soon as the request is acknowledged, the result is put on this bus. This is called *completion*. In contrast to commerical designs such as the IBM's PowerPC, we support only one CDB. The bus is used for two purposes: 1) The instruction is passed to the reservation stations that wait for the result because of a data dependency, and 2) the result is passed to the reorder buffer.

The reorder buffer re-sorts the instructions back in program order. The benefit of this is that we can write the results into the register file in program order (in-order termination). This allows precise interruptions of the instruction stream.

In the following sections, we will describe the data structures and protocols used to realize this in detail.

6.3 Tomasulo Data Structures

6.3.1 Reorder Buffer

The reorder buffer [SP88] is a ring-buffer that serves two purposes in a machine with Tomasulo scheduler. The main purpose is to re-sort the instructions such that the instructions terminate in program order. For that purpose, each reorder buffer entry provides space to store the result of an

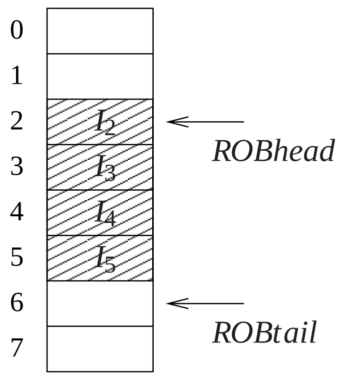


Figure 6.2 Illustration of the reorder buffer pointers

instruction. We support instructions that write multiple registers. This is useful for supporting double precision floating point instructions.

Furthermore, each reorder buffer entry has a *valid bit*. The bit indicates that the result of the instruction is in the reorder buffer entry. A reorder buffer entry with active valid bit is called valid reorder buffer entry.

The second purpose of the reorder buffer is to provide means to assign a *tag* to each instruction. The tag is assigned during instruction issue and stays unique until the instruction terminates. The tag is the address of the reorder buffer entry of the instruction. Let ϑ denote the number of tag (i.e., ROB address) bits. Thus, the reorder buffer has

$$\Theta \quad := \quad 2^{\vartheta}$$

entries. We denote the value of the ROB entry with address *tag* during cycle T with $ROB[tag]^T$.

The reorder buffer is accessed using to pointers, the head and tail pointers. These pointers are stored in ϑ-bit registers. We denote the value of the head pointer during cycle T by $ROBhead^T$, and the value of the tail pointer by $ROBtail^T$. Instructions are put in the ROB entry *ROBtail* points to, and removed from the entry *ROBhead* points to. After an instruction is put in the ROB, the *ROBtail* pointer is increased. After an instruction is removed from the ROB, the *ROBhead* pointer is increased. The pointers wrap-around if they reach the end of the ROB. This is illustrated in figure 6.2.

Let $issue(T)$ denote that we issue an instruction during cycle T. This allows defining the values of *ROBtail* recursively. We initialize the ROB pointers with zero. The *ROBtail* pointer is increased iff we issue an instruction.

$$\mathit{ROBtail}^T \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ \mathit{ROBtail}^{T-1} + 1 & : \quad \mathit{issue}(T-1) \\ \mathit{ROBtail}^{T-1} & : \quad \text{otherwise} \end{cases}$$

Note that the incrementation for the case $\mathit{issue}(T-1)$ holds is a bitvector operation as described in chapter 2. Thus, the *ROBtail* pointer wraps around.

In analogy to that, let $\mathit{writeback}(T)$ denote that we terminate an instruction during cycle T. This allows defining the values of *ROBhead* recursively.

$$\mathit{ROBhead}^T \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ \mathit{ROBhead}^{T-1} + 1 & : \quad \mathit{writeback}(T-1) \\ \mathit{ROBhead}^{T-1} & : \quad \text{otherwise} \end{cases}$$

As above, the incrementation for the case $\mathit{writeback}(T-1)$ holds is a bitvector operation as described in chapter 2. Thus, the *ROBhead* pointer wraps around.

6.3.2 Register File Extentions

As before, the register file holds the values of the specification registers of the machine. We still denote the set of registers by R (in PVS, we just number the registers). We denote the value of the register $r \in \mathsf{R}$ during cycle T by $R[r]^T.\mathit{data}$. We assume that all registers have a common width. We denote the set of possible values of a register by $\mathsf{W}(R)$.

The register file is extended with a *producer table*. The producer table records which instruction in the machine writes a given register. For that purpose, the producer table contains two data items for each register.

The first is a valid bit. We denote the value of the valid bit of register r during cycle T with $R[r]^T.\mathit{valid}$. If it is set, there is no instruction currently executing with the register as destination. If it is not set, there is such an instruction. In this case, the second item, a reorder buffer tag, points to the last instruction with the register as destination. We denote the value of this tag by $R[r]^T.\mathit{tag}$.

6.3.3 Reservation Stations

The reservation stations act as queue for the instructions and their source operands. We give each reservation station a number. We denote the values in reservation station number rs during cycle T by $RS[rs]^T$. Each reservation has a full bit $RS[rs].full$. It indicates that the reservation station is in use. In addition to that, we store the tag of the instruction in the reservation station in $RS[rs].tag$.

We support instructions with an arbitrary number of source operands. Let x denote the number of a source operand. For each source operand, we store a valid bit $RS[rs].op[x].valid$. If the bit is set, the value of the operand is stored in $RS[rs].op[x].data$. If it is not set, we store the tag of the instruction producing the value in $RS[rs].op[x].tag$.

6.3.4 Producers

The producers buffer the results from the function units until the CDB is available. We have a separate producer for each function unit. Each producer consists of a full bit, a tag, and the result. We denote these items of producer fu by $P[fu].full$, $P[fu].tag$, and $P[fu].result$.

6.3.5 Initial Configuration

We make the following assumptions about the initial values of those registers.

- The valid bits of the registers must be set in the initial configuration. We do not make an assumption on the values of the registers or the tags.
- The full bits of the reservation stations must not be set. We do not make any assumptions about the other values in the reservation stations.
- The full bits of the producers must not be set. We do not make any assumptions about the other values in the producers.

It is important that we do not make to many assumptions on initial values, since realizing fixed initial values in hardware is expensive regarding hardware cost. In particular, assuming initial values of a register usually prohibits implementing the register as RAM. In particular, note that we do not make any assumption about the initial values of the ROB entries.

6.4 Tomasulo Protocols

6.4.1 Formalization

In this section, we describe the protocols of the Tomasulo Scheduling algorithm. These protocols form the transition function of a generic and abstract microprocessor with Tomasulo scheduler. The configuration set of this machine comprises of the reservation stations, the reorder buffer including the pointers, the register files, the producers, and the producer tables.

We denote the configuration of this machine during cycle T by c_{aI}^{T} (abstract implementation).

The transition function of the machine is denoted by δ_{aI}. It maps the configuration of the machine during cycle T to the next configuration of the machine during cycle $T+1$. We will compose this function using functional specifications of the Tomasulo protocols, which are issue, CDB snooping, dispatch, completion, and writeback. We name the functions for these protocols *issue*, *snoop*, *dispatch*, *completion*, and *writeback*. These functions are called protocol functions.

$$\delta_{aI} \quad := \quad issue \circ snoop \circ dispatch \circ completion \circ writeback$$

Thus, the issue protocol has priority over CDB snooping and so on. This is important if two protocols change the same register value in the same cycle. The final value in the register is the value provided by the protocol with the higher priority. We omit the transition function for the ROB pointers, since we already specified the values of those pointers above.

Notation We specify the protocols using a notation similar to the notation used in [KMP99]. The notation is also very similar to the notation used in PVS. Consider the following example:

$$R[4].data \quad := \quad R[3].data$$

This is a shorthand for $R[4]^{T+1}.data = R[3]^{T}.data$.

As before, we consider a stream of instructions I_0, I_1, ... Each instruction has source and destination registers. By $S(i,x)$, we denote the number of register that is the source operand x. By $D(i,x)$, we denote the number of register that is the destination operand x.

By $dest(i,r)$, we denote the fact that instruction I_i has r as destination register, i.e., that there is a x with $D(i,x) = r$.

Embedding Convention In a machine with Tomasulo Scheduler and reorder buffer, there are different places where results are stored or propagated before writing the results into the register file. These are the producers, the CDB, and the ROB. We support multiple destination registers for a single instruction. By convention, each destination register is on a well-defined part of the result bus or registers. For example, consider the DLX with floating point instructions. That machine has a maximum of three results for each instruction. Thus, the result busses and registers have space for three 32-bit registers, $result[0]$, $result[1]$, and $result[2]$.

In case of the DLX, we embed the results as follows: By convention, all floating point registers with odd numbers are on $result[1]$, all other "normal" registers are on $result[0]$. In order to handle exceptions, we define a dummy register CA, which is on $result[2]$. This allows handling the IEEE flags register and exceptions.

For example, the result of a double precision floating point instruction with destination register FPR_0 is embedded as follows: The lower part of the result, i.e., the part that is written into FGR_0, is on $result[0]$. The higher part, i.e., the part that is written into FGR_1, is on $result[1]$. The exceptions/IEEE flags are on $result[2]$.

Formally, we define an embedding function. Let d denote the maximum number of destination operands. The embedding function e maps a register to a number in $\{0,\dots,d-1\}$. Thus, destination register r is on $result[e(r)]$.

6.4.2 Issue

Let I_i be the instruction to be issued during cycle T (figure 6.3). The first step is to invalidate the destination registers of instruction I_i. Thus, we clear the valid bit of all registers $R[r]$ with $dest(i,r)$ and set the tag of register r to $ROBtail^T$.

In contrast to the issue protocols given in [MPK00], we cover two different ways to issue an instruction: the first way is as described in [MPK00] and as done by the original Tomasulo scheduling algorithm. During issue, the instruction is stored in a reservation station along with the source operands that are available.

The second way is to skip the reservation stations and to store the result of the instruction in the reorder buffer directly. This speeds up the execution of simple instructions. Examples for this are branches, jumps, and the $trap$ instruction.

The result of these instructions is already known in the issue stage. We indicate these instructions by the predicate $issue\_with\_result(i)$. In case of such an instruction, the reservation stations are not modified by the issue protocol. However, we set the valid bit of the ROB entry $ROBtail$ points to and store the result in the $result$ data item. We denote this result by $issue\_result(i)$. For example, this could be the PC address in case of a jump-and-link instruction.

Machines that support instructions that are directly issued into the ROB are usually not covered in the open literature. The Tomasulo implementation in [Krö99] uses this feature. However, the proof does not cover it.

In case $issue\_with\_result(T)$ does not hold, we clear the valid bit of the ROB entry $ROBtail^T$. Let $issue\_rs(T, rs)$ hold iff reservation station rs is used for issue during cycle T. We initialize this reservation station as follows: we set the full bit of the reservation station and store the $ROBtail$ pointer in the tag data item. Besides the full bit and tag, the reservation station holds the source operands.

The Tomasulo scheduling algorithm with reorder buffer supports different places to forward the source operands from. For each operand of the instruction three sources have to be checked:

1. The operand might be in the register file. In this case, the valid bit of the register is set. If it is not in the register file, the producer table

provides the tag of the last instruction writing it.

2. The operand might be on the CDB. In order to determine which instruction is on the CDB, the result on the CDB comes with a valid bit and a tag. If the valid bit is set, the tag indicates the instruction on the CDB. Thus, we check the valid bit and compare the tag on the CDB with the tag from the producer table. If they match, we take the result on the CDB as source operand according to the embedding convention.

3. The operand might be in the reorder buffer. This is indicated by the valid bit of the reorder buffer entry that the tag in the producer table points to. If the bit is set, we take the result from the ROB according to the embedding convention.

If none of the three cases above applies, the source register is the destination of a preceding, incomplete instruction. The tag of this instruction is in the producer table, and instead of the operand, the tag of this instruction is stored in the reservation station.

6.4.3 CDB Snooping

During issue, the operands in the reservation station that are not available are marked as not valid. On completion, the result of an operation is put on the CDB. Instructions in the reservation stations, which depend on this result, read the operand data from the CDB (figure 6.4). The reservation stations identify the results by comparing the tag on the CDB with the tag in the reservation station.

6.4.4 Dispatch

During instruction dispatch (figure 6.5), an instruction moves from a reservation station entry into the actual function unit. We denote this fact by the predicate $dispatch(T, rs)$. If the predicate holds, the instruction in reservation station rs is dispatched during cycle T.

The reservation stations that are dispatched are determined by the hardware using a fair arbiter, which selects only full reservations with valid

```
if $issue(T)$ then
{
 $RS[rs].full := 1$;
 $RS[rs].tag := ROBtail$;

 For all source operands $x$ of $I_i$, let $r$ be $S(i,x)$:
    if $R[r].valid$ then
        $RS[rs].op[x] := R[r]$;
    elsif $CDB.tag = R[r].tag \wedge CDB.valid$ then
        $RS.op[x].valid := 1$;
        $RS.op[x].data := CDB.result[e(r)]$;
    elsif $ROB[R[r].tag].valid$ then
        $RS.op[x].valid := 1$;
        $RS.op[x].data := ROB[R[r].tag].result[e(r)]$;
    else
        $RS.op[x].valid := 0$;
        $RS.op[x].tag := R[r].tag$;
    endif

 For all registers $r$ with $dest(i,r)$:
    $R[r].tag := ROBtail$;
    $R[r].valid := 0$;
}
```

Figure 6.3 Issue protocol for issuing instruction I_i during cycle T.

```
$\forall$ operands $x$ of instruction $I_i$
    if $RS[rs].full \wedge /RS[rs].op[x].valid \wedge$
          $(RS[rs].op[x].tag = CDB.tag)$
      {
        $RS[rs].op[x].valid := 1$;
        $RS[rs].op[x].data := CDB.result[e(S(i,x))]$;
      }
```

Figure 6.4 CDB snooping protocol for instruction I_i in reservation station rs

```
if $dispatch(T,rs)$ then
 {
  Pass instruction, operands,
  and tag to FU

  $RS.full := 0$;
 }
```

Figure 6.5 Dispatch protocol

operands. Thus, we can assume that the reservation stations *rs* that are dispatched are full and have valid operands:

$$\begin{aligned} dispatch(T,rs) \quad \Longrightarrow \quad & RS[rs]^T.full \wedge \\ & \forall x : RS[rs]^T.op[x].valid \end{aligned}$$

In addition to passing the instruction to the function unit, the reservation station is freed during dispatch. Note that clearing the full bit may conflict with setting the full bit as done by the issue protocol. Since the issue protocol has priority, the full bit is set in this case.

6.4.5 Completion

During completion (figure 6.6), the result and the ROB tag in a producer $P[fu]$ are put on the CDB. Let the predicate $completion(T)$ hold iff the machine completes an instruction. Let $fu = compl\_p(T)$ denote the number of the producer that holds that instruction. That number is determined by the hardware among the full producers using a fair arbiter. Thus, we can assume that the producer is full:

$$completion(T) \quad \Longrightarrow \quad P[compl\_p(T)]^T.full$$

During completion, the according reorder buffer entry is filled with the result and the valid bit is set. Let $FU[fu]^T.valid$ denote that the function unit provides a result. Let $FU[fu]^T.result$ denote that result. Let $FU[fu]^T.tag$ denote the tag that accompanies the result.

If the function unit provides a new result, this result is stored in the producer. If not so, the full bit of the producer is cleared.

```
if $completion(T)$ then
  {
    $CDB^T.valid = 1$;
    $CDB^T.result = P[compl\_p(T)].result$;
    $CDB^T.tag = P[compl\_p(T)].tag$;

    $ROB[CDB^T.tag].valid := 1$;
    $ROB[CDB^T.tag].result := CDB^T.result$;
  }

$\forall$ function units $fu$:
    if $FU[fu]^T.valid$ then
      {
        $P[fu].full := 1$;
        $P[fu].result := FU[fu]^T.result$;
        $P[fu].tag := FU[fu]^T.tag$;
      }
    elsif $completion(T) \wedge compl\_p(T) = fu$ then
        $P[fu].full := 0$;
    endif
```

Figure 6.6 Completion protocol

```
if $writeback(T)$
  for all registers $r$ with $dest(i,r)$:
   {
    $R[r].data := ROB[ROBhead].result[e(r)]$;
    if $ROBhead = R[r].tag$ then
      $R[r].valid := 1$;
   }
```

Figure 6.7 Retirement / writeback protocol for instruction I_i.

6.4.6 Writeback

During writeback (figure 6.7), a result of the instruction in the ROB entry that *ROBhead* points to is written into the register file. As introduced above, we denote this fact by the predicate $writeback(T)$. We assume that writeback is done iff the ROB entry is valid and the ROB is not empty. Let $ROBempty(T)$ denote that the ROB is empty during cycle T. We will later on define it.

$$writeback(T) \quad \Longleftrightarrow \quad \overline{ROBempty(T)} \wedge ROB[ROBhead(T)]^T.valid$$

During writeback, we store the result in the ROB in the registers. Furthermore, we set the valid bit of the register if the tag of the instruction matches the tag in the producer table.

Note that setting the valid bit may conflict with clearing the valid bit during issue. As described above, the issue protocol has priority over the writeback protocol, i.e., the setting of the valid bit is suppressed.

6.5 Data Consistency

6.5.1 Scheduling Functions

We need a formal way to state that "instruction I_i is being issued during cycle T" or "instruction I_i is being dispatched during cycle T". We do this in analogy to the previous chapters using a *scheduling function*. While this concept was introduced for in-order machines by [MP00], we extend it to out-of-order machines in the obvious way.

Issue We recursively define a function *sIissue* that maps a cycle T to the number of the instruction that is in the issue stage. Since we issue in program order, that number increases by one in case that $issue(T)$ holds and stays unmodified otherwise. We start with instruction I_0.

$$sIissue(T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sIissue(T-1)+1 & : \quad issue(T-1) \\ sIissue(T-1) & : \quad \text{otherwise} \end{cases}$$

Reservation Stations We also desire a way to define the instruction in a given reservation station *rs* during a given cycle T. We do this by defining a schedule function $sIRS(rs,T)$ for reservation stations. Instructions are put in a reservation station during issue. In case an instruction is issued into reservation station *rs*, we take the value of $sIissue(T-1)$. Otherwise, the value of $sIRS(rs,T)$ remains unchanged.

$$sIRS(rs,T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sIissue(T-1) & : \quad issue(T-1) \\ sIRS(rs,T-1) & : \quad \text{otherwise} \end{cases}$$

Note that the only point we put an instruction into a reservation station is during issue. This is in contrast to the implementation given [Krö99], which moves the instructions from one reservation station into the next.

Reorder Buffer In analogy to the schedule of the reservation stations, we can provide a schedule for the ROB. The function $sIROB(tag,T)$ denotes the instruction that is in the ROB entry with tag *tag* during cycle T. We start with -1, which denotes that no instruction is in the ROB entry. We need this special value because the ROB entries have no such thing like a full bit.

$$sIROB(tag,T) \quad := \quad \begin{cases} -1 & : \quad T = 0 \\ sIissue(T-1) & : \quad issue(T-1) \wedge \\ & \quad\;\; tag = ROBtail^{T-1} \\ sIROB(tag,T-1) & : \quad \text{otherwise} \end{cases}$$

Function Units Let $dispatch\_fu(T,fu)$ denote the number of the reservation station that is used for dispatching an instruction to function unit fu during cycle T. In hardware, this number is represented unary using $dispatch(T,rs)$.

Let $sIdispatch(fu,T)$ denote the number of the instruction passed to function unit fu during cycle T. This is defined using the schedule of the reservation station.

$$sIdispatch(fu,T) \quad := \quad sIRS(dispatch\_fu(T,fu),T)$$

We also define schedules for the functional units. Let $sIfu(fu,T)$ denote the number of the instruction that **leaves** function unit fu during cycle T. The most simple functional unit is a combinatorial functional unit that calculates its result within the same cycle the arguments are passed. The 32-bit ALU presented in chapter 2 is an example. For such a function unit, $sIfu(fu,T)$ just is:

$$sIfu(fu,T) \quad := \quad sIdispatch(fu,T)$$

In case of more complex function units such as floating point dividers, one has to construct a scheduling function. There are two ways to do so: 1) one constructs the function such that it matches the pipeline structure of the functional unit, and 2) one defines the schedule using the tags the function unit provides.

As an example for the first method, consider a function unit with four stages and a cycle that allows iterating the instruction in stage 2 (figure 6.8). We denote the instruction in stage k of the function unit fu during cycle T by $sI_{fu}(k,T)$. The instruction in stage 0 of the function unit is the instruction that is dispatched:

$$sI_{fu}(0,T) \quad := \quad sIdispatch(fu,T)$$

This instruction proceeds into stage 1 iff the update enable signal $ue_{fu,0}$ is active. This update enable signal is local to the function unit fu.

$$sI_{fu}(1,T) \quad := \quad \begin{cases} 0 & : \quad T=0 \\ sI_{fu}(0,T-1) & : \quad ue^{T-1}_{fu,0}=1 \\ sI_{fu}(1,T-1) & : \quad \text{otherwise} \end{cases}$$

This must be changed for stage 2, the stage with the back-cycle.

$$sI_{fu}(2,T) \quad := \quad \begin{cases} 0 & : \quad T=0 \\ sI_{fu}(1,T-1) & : \quad ue^{T-1}_{fu,1}=1 \wedge sel^{T-1}_{1}=0 \\ sI_{fu}(2,T-1) & : \quad ue^{T-1}_{fu,1}=1 \wedge sel^{T-1}_{1}=1 \\ sI_{fu}(2,T-1) & : \quad \text{otherwise} \end{cases}$$

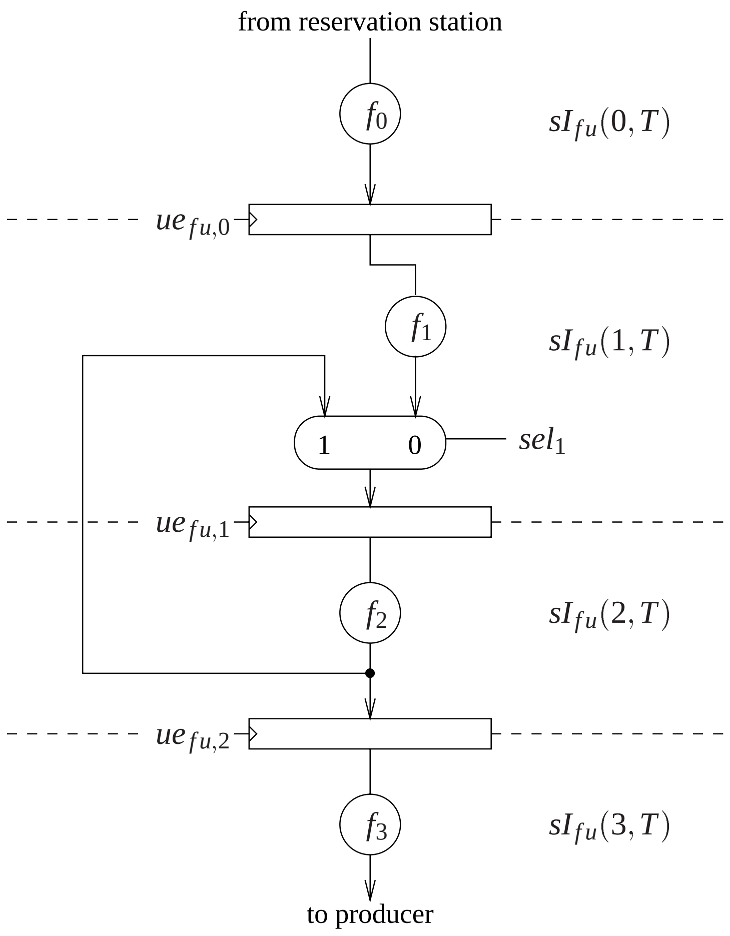


Figure 6.8 Construction of the scheduling function for a function unit with cycles

For stage 3 of the function unit, the scheduling function is defined in analogy to the scheduling function of stage 1:

$$sI_{fu}(3,T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sI_{fu}(2,T-1) & : \quad ue^{T-1}_{fu,2} = 1 \\ sI_{fu}(3,T-1) & : \quad \text{otherwise} \end{cases}$$

Since this is also the last stage of the function unit fu, we have

$$sIfu(fu,T) \quad := \quad sI_{fu}(3,T)$$

Producers In analogy to the scheduling function of the reservation stations, we define the scheduling function of the producer registers. We denote the number of the instruction in producer number fu during cycle T by $sIP(fu,T)$. In case the function unit provides a result, we take the value from the schedule of the function unit as defined above. If not so, the value of $sIP(fu,T)$ does not change.

$$sIP(fu,T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sIfu(fu,T-1) & : \quad FU[fu]^{T-1}.valid = 1 \\ sIP(fu,T-1) & : \quad \text{otherwise} \end{cases}$$

As described above, the instruction in producer with the number given by $compl\_p(T)$ is put on the CDB during completion. We therefore define the following shorthand for the instruction on the CDB during cycle T:

$$sICDB(T) \quad := \quad sIP(compl\_p(T),T)$$

Writeback In analogy to *sIissue*, we recursively define a scheduling function *sIwriteback* that maps a cycle T to the number of the instruction that is in the writeback stage. Since we writeback in program order, that number increases by one in case that $writeback(T)$ holds and stays unmodified otherwise. We start with instruction I_0.

$$sIwriteback(T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sIwriteback(T-1)+1 & : \quad writeback(T-1) \\ sIwriteback(T-1) & : \quad \text{otherwise} \end{cases}$$

6.5.2 Function Unit Axioms

In this section, we describe the assumptions we make regarding data consistency properties of the functional units. We consider the functional units as a "black box". In particular, we do not provide implementations for data memory or floating point function units. The design and verification of a data memory function unit including virtual memory is subject of the thesis of Sven Beyer [Bey01]. The design and verification of an IEEE compliant floating unit including a divider is subject of the thesis of Christian Jacobi [Jac01].

Inputs and Outputs As described above, $FU[fu]^T.valid$ indicates that function unit fu provides a result during cycle T. $FU[fu]^T.tag$ denotes the tag the function unit provides, and $FU[fu]^T.result$ denotes the result the function unit provides.

Let $fuins(fu,T)$ denote the inputs of function unit fu during cycle T. This is a defined as follows: Let rs be a shorthand for $dispatch\_rs(T, fu)$. This is the reservation station that is used for dispatching to function unit fu.

$$\begin{aligned} fuins(fu,T).valid &:= dispatch\_rs(T,rs) \\ fuins(fu,T).tag &:= RS[rs]^T.tag \\ fuins(fu,T).source[x] &:= RS[rs]^T.op[x].data \end{aligned}$$

Tag Consistency Given that the function unit gets correct tags as inputs upto cycle T, we assume that the function unit provides the correct tag of the instruction as output during cycle T.

We formalize "gets correct tags as inputs upto cycle T" as follows:

$$\begin{aligned} & \forall T' \leq T : fuins(fu,T').valid \\ \Longrightarrow \quad & fuins(fu,T').tag = I\_tag(sIdispatch(fu,T')) \end{aligned}$$

We formalize "provides the correct tag of the instruction" as follows:

$$FU[fu]^T.valid \quad \Longrightarrow \quad FU[fu]^T.tag = I\_tag(sIfu(fu,T))$$

Operand Consistency Given that the function unit gets correct source operands as inputs upto cycle T, we assume that the function unit provides the correct results of the instruction as output during cycle T.

We formalize "gets correct source operand as inputs upto cycle T" as follows:

$$\begin{aligned} & \forall T' \leq T : fuins(fu, T').valid \\ \Longrightarrow \quad & fuins(fu, T').source = source(sIdispatch(fu, T')) \end{aligned}$$

We formalize "provides the correct results of the instruction" as follows:

$$FU[fu]^T.valid \quad \Longrightarrow \quad FU[fu]^T.result = result(sIfu(fu, T))$$

Phase Consistency In order to show data consistency, we have to argue that the function units does not generate "garbage output". We assume two things: 1) If an instruction leaves the function unit, it entered it before, and 2) if instructions upto cycle T enter the function unit at most one, the instructions leave the function unit at most once.

We formalize this as follows: Let $in(i, T, fu)$ denote that instruction I_i enters the function unit fu during cycle T.

$$in(i, T, fu) \quad :\Longleftrightarrow \quad fuins(fu, T).valid \wedge sIdispatch(fu, T) = i$$

In analogy to that, let $out(i, T, fu)$ denote that instruction I_i leaves the function unit fu during cycle T.

$$out(i, T, fu) \quad :\Longleftrightarrow \quad FU[fu]^T.valid \wedge sIfu(fu, T) = i$$

If instruction I_i leaves function unit fu during cycle T, there must be a cycle $T' \leq T$ such that it entered the function unit:

$$out(i, T, fu) \quad \Longrightarrow \quad \exists T' \leq T : in(i, T', fu)$$

If the cycle $T' \leq T$ such that instruction I_i enters the function unit during cycle T' is unique, then the cycle $T'' \leq T$ such that instruction I_i leaves the function unit during cycle T'' is unique.

$$\left|\{T' \leq T \mid in(i, T', fu)\}\right| = 1 \quad \Longrightarrow \quad \left|\{T'' \leq T \mid out(i, T'', fu)\}\right| = 1$$

We do not make further assumptions regarding data consistency. In particular, this allows that the latency of the function unit is variable and that the instructions leave the dispatch order within the function unit.

We make further assumptions on the function units in order to show liveness. We will later on describe these assumptions.

6.5.3 ROB Flags

We need means to determine wether the reorder buffer is full or not. For this purpose, we take the circuit from [Lei99]. It uses a $\vartheta+1$ bit counter register. The counter is incremented if we issue and instruction and do not writeback one simulataneously. This is indicated by $ROBinc(T)$.

$$ROBinc(T) \quad = \quad issue(T) \wedge \overline{writeback(T)}$$

In analogy to that, $ROBdec(T)$ indicates that we decrement the counter. This is done if we writeback an instruction but do not issue one simultaneously.

$$ROBdec(T) \quad = \quad \overline{issue(T)} \wedge writeback(T)$$

Thus, the value of the counter register during cycle T is defined as follows:

$$ROBcount(T) \quad := \quad \begin{cases} 0^{\vartheta} & : \quad T=0 \\ ROBcount(T-1)+1 & : \quad ROBinc(T-1) \\ ROBcount(T-1)-1 & : \quad ROBdec(T-1) \\ ROBcount(T-1) & : \quad \text{otherwise} \end{cases}$$

The ROB is empty iff the counter is zero:

$$ROBempty(T) \quad = \quad (ROBcount(T) = 0^{\vartheta+1})$$

The ROB is full iff the counter is the number of ROB entries Θ. We use the binary encoding of Θ.

$$ROBempty(T) \quad = \quad (ROBcount(T) = 10^{\vartheta})$$

We make the following assumptions:

- If we issue an instruction without simultaneous writeback, the ROB must not be full.

$$ROBinc(T) \quad \Longrightarrow \quad \overline{ROBfull(T)}$$

- If we writeback an instruction, the ROB must not be empty.

$$writeback(T) \quad \Longrightarrow \quad \overline{ROBempty(T)}$$

6.5.4 ROB Properties

Definition 6.1 ▶ $tag \oplus i$ Let $tag \oplus i$ be a shorthand for a tag that is incremented i times. Formally, this is defined using a recursion and the bit-vector incrementation as defined in chapter 2:

$$tag \oplus i \quad := \quad \begin{cases} tag & : \quad i = 0 \\ (tag \oplus (i-1)) + 1 & : \quad \text{otherwise} \end{cases}$$

Note that we increment a bit vector with limited range. Thus, it will wrap-around. One easily verifies the following properties of the ROB pointers:

Lemma 6.1 ▶ Let i be the number of the instruction in the issue stage. The ROB tail pointer has been increased i times.

$$ROBtail^T \quad = \quad 0^{\vartheta} \oplus sIissue(T)$$

Lemma 6.2 ▶ Let i be the number of the instruction in the writeback stage. The ROB head pointer has been increased i times.

$$ROBhead^T \quad = \quad 0^{\vartheta} \oplus sIwriteback(T)$$

The proof for both lemmas is easily done using induction on T.

Lemma 6.3 ▶ The value in the *ROBcount* register is smaller or equal than the number of ROB entries.

$$\langle ROBcount(T) \rangle \quad \leq \quad \Theta$$

PROOF One verifies this claim by induction on T. For $T = 0$, we have

$$\langle ROBcount(T) \rangle \quad = \quad 0.$$

For $T+1$, we show the claim by a full case split on the values of $ROBinc(T)$ and $ROBdec(T)$.

- If neither $ROBinc(T)$ or $ROBdec(T)$ holds, the value of $ROBcount$ does not change and the claim is concluded using the induction premise.
- If $ROBinc(T)$ holds, we assert the claim as follows: in case

  $$\langle ROBcount(T) \rangle \quad < \quad \Theta$$

  holds, the claim is easily concluded. Assume

  $$\langle ROBcount(T) \rangle \quad = \quad \Theta$$

  holds. In this case, we have a contradiction to the assumption above since $ROBinc(T)$ holds and the ROB is full.
- If $ROBdec(T)$ holds, we assert the claim as follows: in case

  $$\langle ROBcount(T) \rangle \quad \neq \quad 0$$

  holds, the claim is easily concluded. Assume

  $$\langle ROBcount(T) \rangle \quad = \quad 0$$

  holds. In this case, we have a contradiction to the assumption above since $ROBdec(T)$ holds and the ROB is empty. QED

◄ Lemma 6.4

Let

$$instr\_in\_rob(T) = sIissue(T) - sIwriteback(T)$$

denote the difference between the number of issued and terminated instructions, i.e., the number of instructions in the reorder buffer. We claim that this number is equal to the binary number interpretation of the value of $ROBcount(T)$:

$$instr\_in\_rob(T) \quad = \quad \langle ROBcount(T) \rangle$$

PROOF This claim is asserted by induction on T. For $T = 0$ we have

$$\begin{array}{rcl} instr\_in\_rob(T) & = & \langle ROBcount(T)\rangle \\ sIissue(T) - sIwriteback(T) & = & \langle 0^{\vartheta+1}\rangle \\ 0 - 0 & = & \langle 0^{\vartheta+1}\rangle. \end{array}$$

For $T+1$, we do a full case split on the values of the signals $issue(T)$ and $writeback(T)$.

- If neither $issue(T)$ nor $writeback(T)$ holds, both the values of the scheduling functions and the ROB counter do not change from cycle T to $T+1$. Thus, the claim is concluded by the induction premise.
- If both $issue(T)$ and $writeback(T)$ hold, both scheduling functions are incremented by one. Thus, the difference stays the same. The ROB counter does not change from cycle T to $T+1$. Thus, the claim is concluded by the induction premise.
- In case $issue(T)$ holds and $writeback(T)$ does not hold, the difference is increased by one. The ROB couter is also increased by one. One asserts that the ROB counter does not wrap around by lemma 6.3.
- In case $issue(T)$ doe not hold and $writeback(T)$ holds, the difference is decreased by one. The ROB couter is also decreased by one. One asserts that the ROB counter does not wrap around using the assumption that we do not writeback in case of an empty ROB. 6.3.

QED

Lemma 6.5 ► The number of instructions in the ROB is greater or equal than zero.

$$instr\_in\_rob(T) \geq 0$$

One easily asserts this using lemma 6.4.

Lemma 6.6 ► The number of instructions in the ROB is smaller or equal than the number of ROB entries.

$$instr\_in\_rob(T) \leq \Theta$$

This is easily shown using lemma 6.4 and lemma 6.3.

The following lemma is easily concluded using lemma 6.5:

◀ Lemma 6.7

The number of issued instructions is greater or equal than the number of terminated instructions.

$$sIissue(T) \geq sIwriteback(T)$$

◀ Lemma 6.8

If we terminate an instruction using cycle T, the number of issued instructions is greater than the number of terminated instructions.

$$writeback(T) \quad \Longrightarrow \quad sIissue(T) > sIwriteback(T)$$

One easily shows this using lemma 6.7, and lemma 6.4, and the fact that we only writeback if the ROB is not empty.

◀ Lemma 6.9

The number of issued instructions upto cycle T is greater or equal than the number of terminated instructions upto cycle $T+1$.

$$sIissue(T) \quad \geq \quad sIwriteback(T+1)$$

One easily verifies this claim using lemma 6.8 for the case $writeback(T)$ and using lemma 6.7 otherwise.

◀ Definition 6.2
$I\_tag(i)$

As described above, we assign a tag to each instruction during issue. This is the value of the ROB tail pointer. This pointer is increased by one each time we issue an instruction. Thus, we define a function $I\_tag(i)$, which denotes the tag of instruction I_i, as follows:

$$I\_tag(i) \quad := \quad 0^{\vartheta} \oplus i$$

◀ Lemma 6.10

$I\_tag(i)$ is the value of the ROB tail pointer during issue of instruction I_i.

$$ROBtail^{T} \quad = \quad I\_tag(sIissue(T))$$

This claim is easily concluded using lemma 6.1 and the definition of I_tag.

Lemma 6.11 ► If an instruction is in ROB entry tag, then the tag of that instruction is tag.

$$sIROB(tag,T) = i \quad \Longrightarrow \quad tag = I\_tag(i)$$

PROOF One shows this claim by induction on T. For $T = 0$, there is nothing to show since there is no instruction in the ROB (formally, $sIROB(tag,0)$ is -1, and there is no instruction I_{-1}).

For $T+1$, the claim is concluded by expanding the definition of $sIROB$. If

$$issue(T) \wedge tag = ROBtail^T$$

holds, we have $sIROB(tag,T+1) = sIissue(T)$. The claim is then concluded using lemma 6.10.

If not so, we have $sIROB(tag,T+1) = sIROB(tag,T)$. The claim is then concluded using the induction premise. QED

We will now show that this tag is unique beginning with the cycle the instruction is issued until the instruction terminates. Formally, this means that we can assign a single, unique instruction to each such tag.

Let $issued(i,T)$ hold iff instruction I_i is already issued during cycle T. We define this predicate using the scheduling function $sIissue$:

$$issued(i,T) \quad :\Longleftrightarrow \quad sIissue(T) > i$$

However, it is not obvious that instruction I_i was issued before cycle T if $sIissue(T) > i$ and vice-versa. It is an implication of in-order issue. The following lemma asserts one direction.

Lemma 6.12 ► If $issued(i,T)$ holds, there is a cycle $T' < T$ such that I_i is issued during cycle T'.

$$issued(i,T) \quad \Longrightarrow \quad \exists T' < T : sIissue(T') = i \wedge issue(T')$$

PROOF The claim is shown by induction on T. For $T = 0$, we have $sIissue(0) = 0$. Thus, $sIissue(0) > i$ cannot hold and there is nothing to show.

For $T+1$, we show the claim using a case split on $issue(T)$.

- If $issue(T)$ holds, we have

  $$sIissue(T+1) = sIissue(T) + 1$$

  and therefore $sIissue(T)+1 > i$. Let $sIissue(T) > i$ hold. In this case, we can apply the induction premise and the claim holds. Thus, let $sIissue(T) = i$ hold. In this case, cycle T satisfies the claim.

- If $issue(T)$ does not hold, we have $sIissue(T+1) = sIissue(T)$ and we can apply the induction premise to show the claim.

QED

In analogy to $issued(i,T)$, we define a predicate $terminated(i,T)$ that holds iff instruction I_i already terminated before cycle T.

$$terminated(i,T) \quad :\Longleftrightarrow \quad sIwriteback(T) > i$$

Let the predicate $\tau(i,T)$ be a shorthand for the fact that instruction I_i is already issued during cycle T but has not yet terminated.

$$\tau(i,T) \quad :\Longleftrightarrow \quad issued(i,T) \wedge \overline{terminated(i,T)}$$

The following lemma will be used in order to show that issue is done in program order.

◄ Lemma 6.13

Consider the instruction in the issue stage during cycle T. During cycle $T+1$, there is the same or a later instruction in the issue stage.

$$sIissue(T+1) \quad \geq \quad sIissue(T)$$

The proof of lemma 6.13 is easily done by expanding the definition of the scheduling function $sIissue(T+1)$.

◄ Lemma 6.14

The instructions are issued in order, i.e., during cycle $T' \leq T$ there is the same or an earlier instruction in the issue stage.

$$\forall T' \leq T : sIissue(T') \leq sIissue(T)$$

This lemma is easily shown using induction on T and lemma 6.13 as induction step.

Lemma 6.15 ▶ Let $i \geq 0$ and $j \geq 0$ hold. If one increments a tag i times and after that j times, this is equivalent to incrementing the tag $i+j$ times.

$$(tag \oplus i) \oplus j \quad = \quad tag \oplus (i+j)$$

This is easily shown by induction on j.

Lemma 6.16 ▶ Let T and $T' \geq T$ be cycles. $ROBtail^{T'}$ is equal to $ROBtail^{T}$ incremented $sIissue(T') - sIissue(T)$ times.

$$\forall T' \geq T : \qquad ROBtail^{T'} = ROBtail^{T} \oplus (sIissue(T') - sIissue(T))$$

PROOF By applying lemma 6.1 twice, the claim is transformed into:

$$0^{\vartheta} \oplus sIissue(T') \quad \overset{!}{=} \quad (0^{\vartheta} \oplus sIissue(T)) \oplus (sIissue(T') - sIissue(T))$$

One shows $sIissue(T') - sIissue(T) \geq 0$ using lemma 6.14. This allows concluding the claim using lemma 6.15.
QED

One easily verifies the following property of tag arithmetic (i.e., bit-vector arithmetic). It applies for incrementing tags as done for *ROBhead* and *ROBtail*.

Lemma 6.17 ▶ If one increments a tag i times, the value of this tag is the value of the old tag plus i modulo Θ (number of ROB entries).

$$\langle tag \oplus i \rangle \quad = \quad \langle tag \rangle + i \bmod \Theta$$

The following lemma will be used in order to argue that certain entries in the ROB are not overwritten.

Lemma 6.18 ▶ If one increments a tag at least once and less than Θ times, the incremented tag is different from the old tag.

$$0 < j < \Theta \quad \Longrightarrow \quad (tag \oplus j) \neq tag$$

PROOF According to lemma 6.17, we have

$$\langle tag \oplus j \rangle \quad = \quad \langle tag \rangle + j \bmod \Theta$$

Assume $(tag \oplus j) = tag$ holds. In this case, the equation above transforms into:

$$\langle tag \rangle \quad = \quad \langle tag \rangle + j \bmod \Theta$$

This only holds if j is a multiple of Θ (this property of mod is shown in the PVS libraries). This is a contradiction to the premise of the lemma and we therefore have $(tag \oplus j) \neq tag$. QED

Entries in the ROB are overwritten if the ROB tail pointer wraps around. This happens each Θ (number of ROB entries) instructions. The following lemma asserts the fact that instruction I_i in the ROB is overwritten only in this case.

◄ Lemma 6.19

Let instruction I_i be issued during cycle T'. Consider cycles $T > T'$. As long as no more than Θ instructions are issued from cycle T' to T, the instruction in the ROB entry during cycle T that $ROBtail^{T'}$ points to is instruction i.

$$\begin{aligned} & issue(T') \wedge sIissue(T') = i \wedge sIissue(T) \leq (i + \Theta) \\ \Longrightarrow \quad & sIROB(ROBtail^{T'}, T) = i \end{aligned}$$

PROOF

The proof proceeds by induction on T. For $T = 0$, there is nothing to show since there is no cycle $T' \geq 0$ with $T > T'$.

For $T + 1$, let us consider the case $T = T'$. In this case, the claim holds by definition of *sIROB*.

The claim for the case $T > T'$ is (we swap left hand side and right and side):

$$\begin{aligned} i \quad & \overset{!}{=} \quad sIROB(ROBtail^{T'}, T) \\ & \overset{!}{=} \quad \begin{cases} sIissue(T) & : \quad issue(T) \wedge \\ & \quad\;\; ROBtail^{T'} = ROBtail^{T} \\ sIROB(ROBtail^{T'}, T) & : \quad \text{otherwise} \end{cases} \end{aligned}$$

We argue the two cases above separately. Assume

$$issue(T) \; \wedge \; ROBtail^{T'} = ROBtail^{T}$$

holds. This implies that $sIissue(T+1) = sIissue(T)+1$ holds because $issue(T)$ holds. This allows concluding that

$$sIissue(T)+1 \leq i+\Theta$$

holds. This allows applying lemma 6.18 with $j = sIissue(T)-i$, which states:

$$ROBtail^{T'} \quad \neq \quad ROBtail(T') \oplus (sIissue(T)-i)$$

According to lemma 6.16 for cycles T' and T, we have

$$ROBtail^{T} \quad = \quad ROBtail^{T'} \oplus (sIissue(T)-i).$$

Thus, this is a contradiction to $ROBtail^T = ROBtail^{T'}$. Thus,

$$issue(T) \wedge ROBtail^{T'} = ROBtail^T$$

cannot hold. We therefore only have to show $sIROB(ROBtail^{T'}, T) = i$. This is done using the induction premise. QED

Lemma 6.20 ▶ If instruction I_i has been issued but has not not yet terminated, less than Θ (number of ROB entries) instructions have been issued since I_i was issued.

$$\tau(i,T) \quad \Longrightarrow \quad sIissue(T) \leq i+\Theta$$

This claim is easily concluded using lemma 6.6.

The following theorem provides the unique mapping from tags to instructions: we just use the ROB schedule. The tag of an instruction is unique, if the instruction in the ROB.

Theorem 6.21 ▶ If instruction I_i has been issued but has not not yet terminated, the instruction in ROB entry $I\_tag(i)$ is instruction i.

$$\tau(i,T) \quad \Longrightarrow \quad sIROB(I\_tag(i),T) = i$$

PROOF According to lemma 6.12, there is a cycle $T' < T$ such that instruction I_i is issued during cycle T'. According to lemma 6.19 for cycle T' and T and instruction i, we have:

$$sIissue(T) \leq i+\Theta \quad \Longrightarrow \quad sIROB(ROBtail^{T'},T) = i$$

We assert the left hand side of the implication using lemma 6.20. Thus, we have:

$$sIROB(ROBtail^{T'}, T) \quad = \quad i$$

It is therefore left to show that $ROBtail^{T'}$ is equal to $I\_tag(i)$. This is done using lemma 6.10.

QED

From lemma 6.21, one easily concludes the following claim:

◀ Lemma 6.22

Let I_i and I_j be instructions. If the tags of the instructions are equal and both unique, instruction i is instruction j.

$$I\_tag(i) = I\_tag(j) \wedge \tau(i,T) \wedge \tau(j,T) \quad \Longrightarrow \quad i = j$$

In analogy to lemma 6.10, we show:

◀ Lemma 6.23

The *ROBhead* pointer during cycle T is the tag of the instruction in write-back stage.

$$ROBhead(T) \quad = \quad I\_tag(sIwriteback(T))$$

One easily concludes this claim using lemma 6.2

◀ Lemma 6.24

If we writeback an instruction during cycle T, that instruction is in the ROB entry that *ROBhead* points to.

$$writeback(T) \quad \Longrightarrow \quad sIwriteback(T) = sIROB(ROBhead(T), T)$$

PROOF

Using lemma 6.23, we transform the claim into:

$$writeback(T) \quad \Longrightarrow \quad sIwriteback(T) = sIROB(I\_tag(sIwriteback(T)), T)$$

The claim is concluded using lemma 6.21. It is left to show that the premise of lemma 6.21 holds, i.e., we have to show that

$$\tau(sIwriteback(T), T)$$

holds. We show that the instruction is already issued using lemma 6.8. Furthermore, the instruction is obviously not terminated yet.

QED

6.5.5 Instruction Phases

We distinguish the following phases of executing instruction I_i:

- **Not issued:** Before an instruction is issued, the instruction is in the ”not issued” phase. Formally, this holds if $\overline{issued(i,T)}$ holds.
- **In RS:** During issue, the instruction is stored in a reservation station unless $issue\_with\_result(i)$ holds. Formally, instruction I_i is in a reservation station during cycle T iff

  $$\exists rs: \ RS[rs]^T.full \wedge sIRS(rs,T) = i$$

  holds.
- **In FU:** During dispatch, the instruction is passed from the reservation station to a function unit. Formally, we say an instruction is dispatched during cycle T iff there is a cycle $T' < T$ and a reservation station rs such that instruction I_i is in reservation station rs and the instruction in that reservation station is dispatched.

  $$\begin{aligned} & dispatched(i,T) \\ :\Longleftrightarrow \quad & \exists T' < T, rs: \ dispatch\_rs(T',rs) \wedge sIRS(rs,T') = i \end{aligned}$$

  The instruction leaves the function unit if it is passed to a producer. Formally, an instruction is executed iff there is a cycle $T' \leq T$ and a producer fu such that instruction I_i is in the producer fu and that producer is full.

  $$\begin{aligned} & executed(i,T) \\ :\Longleftrightarrow \quad & \exists T' < T, fu: \ FU[fu]^{T'}.valid \wedge sIfu(fu,T') = i \end{aligned}$$

  Formally, instruction I_i is in a function unit during cycle T iff

  $$dispatched(i,T) \wedge \overline{executed(i,T)}$$

  holds. Note that there are function units (ALU, for example), that return the result in the same cycle they get it. In this case, the condition above never holds, although the function unit is not bypassed.
- **In producer:** After leaving the function unit, the result of the instruction is stored in a producer. Formally, an instruction is in a producer iff there is a producer fu such that instruction I_i is in the producer fu and the producer is full.

  $$\exists fu: \ P[fu]^T.full \wedge sIP(fu,T) = i$$

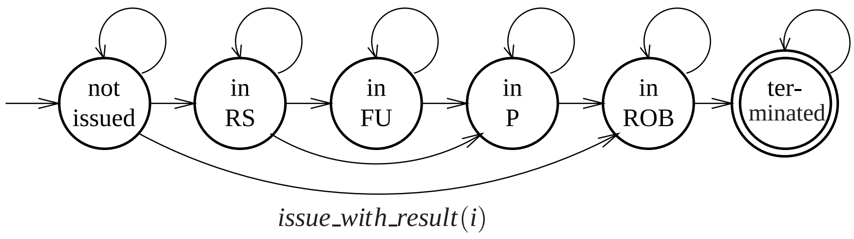


Figure 6.9 Instruction phase state diagram

- **In ROB:** As soon as the producer gets the CDB, the result in the producer is stored in the ROB. Formally, an instruction is in the ROB during cycle T iff there is a ROB entry tag such that the instruction in that entry is I_i and the entry is valid and the instruction has not terminated yet.

$$\exists\, tag:\ ROB[tag]^T.valid \wedge sIROB(tag,T) = i \wedge \overline{terminated(i,T)}$$

The phases of "normal" instructions, i.e., instructions I_i that are not issued with result, are processed in the order above. Instructions with $issue\_with\_result(i)$ skip the phases "in RS", "in FU", and "in producer". This is illustrated in figure 6.9. The figure shows the different phases and the transitions between the phases. However, one has to assert this property of the machine. This is done by the following lemmas.

Let $p(i,T)$ denote that instruction I_i is in phase p during cycle T.

Let $pred(p)$ denote the set of predecessor phases of phase p according to figure 6.9. For example, the "not issued" phase only has itself as predecessor. The "in ROB" phase has three predecessor phases: "in ROB", "not issued", and "in producer".

In analogy to $pred(p)$, let $succ(p)$ denote the set of successor phases of phase p according to figure 6.9. For example, the "not issued" phase has two successor phases: "in RS" and "in ROB".

◄ Lemma 6.25

If instruction I_i is in a given phase during cycle T, and not in any other phase, we show that the instruction is in at most one successor phase during cycle $T+1$, i.e., the sucessor phases mutually exclude each other.

PROOF

For most phases, the claim is trivial, because they only thave themselves and another state as successors. The only exception is the "not issued" phase, which has three successors. We therefore show the claim exemplary for the "not issued" phase.

- If $issue(T)$ and $sIissue(T) = i$ does not hold, one easily concludes that instruction I_i stays in "not issued" phase during cycle $T+1$. Thus, we have to show that it is not in a reservation station or in the ROB. According to the premise of the lemma, the phases of I_i are unique during cycle T. Thus, I_i is not in the ROB or in a reservation station during cycle T. Since I_i is also not issued, one easily verifies that it does not move into the ROB or into a reservation station.
- If $issue(T)$ and $sIissue(T) = i$ holds, one easily concludes that instruction I_i either enteres the ROB or a reservation station, depending on $issue\_with\_result(i)$. If $issue\_with\_result(i)$ holds, one verifies that the instruction cannot be in a reservation station. If not so, one verifies that the instruction cannot be in the ROB.

QED

Lemma 6.26 ▶ If instruction I_i is in a given phase during cycle $T+1$, we show that it must have been in one of the predecessor phases as given in figure 6.9 during cycle T:

$$p(i,T+1) \quad \Longrightarrow \quad \bigvee_{p' \in pred(p)} p'(i,T)$$

For example, if instruction I_i is in phase "not issued" during cycle $T+1$, this implies that it must be in phase "not issued" during cycle T.

PROOF In PVS, we split this claim into 6 lemmas, one for each phase. We show the claim for the "not issued" phase and the "in RS" phase here exemplary.

- The claim for the "not issued" phase is easily asserted by expanding the definition of "not issued" and by applying lemma 6.13.
- The claim for the "in RS" phase is asserted as follows: according to the premise, there is a reservation station rs such that

$$RS[rs]^{T+1}.full \wedge sIRS(rs,T+1) = i$$

holds. Let $issue\_rs(T,rs)$ hold. In this case, we have

$$sIRS(rs,T+1) \quad = \quad sIissue(T)$$

Thus, the instruction I_i is in issue stage during cycle T. Thus, it is in "not issued" phase during cycle T, which concludes the claim.

Let $issue\_rs(T,rs)$ not hold. In this case, one easily asserts that the full bit $RS[rs]^T.full$ is active and $sIRS(rs,T) = i$ holds. Thus, the instruction is in "in RS" phase during cycle T, which concludes the claim.

QED

◀ Lemma 6.27

The phase of instruction I_i during cycle T is unique, i.e., the phases above exclude each other mutually.

PROOF

One easily shows this claim by induction on T. For $T = 0$, one asserts that all instructions are in "not issued" phase only.

For $T+1$, one shows the claim as follows: according to the induction premise, instruction I_i is in at most one phase during cycle T. One applies lemma 6.25, which shows that the successor states mutually exclude each other.

Furthermore, the instruction I_i cannot be in a phase that is not a successor phase during cycle $T+1$, which is asserted by lemma 6.26.

QED

6.5.6 Tag Consistency

We will now show that the tags transported in the machine are consistent with the scheduling functions, i.e., we will show that the tag stored together with instruction I_i is $I\_tag(i)$.

◀ Lemma 6.28

If a reservation station is full, the tag in that reservation station is the tag of the instruction in the reservation station.

$$RS[rs]^T.full \quad \Longrightarrow \quad RS[rs]^T.tag = I\_tag(sIRS(rs,T))$$

PROOF

The claim is shown using induction on T. For $T = 0$ there is nothing to show because the reservation stations are not full in the initial configuration.

For $T+1$, we show the claim as follows: If an instruction I_i is issued into reservation station rs during cycle T, the value of the tag in reservation station is defined by the issue protocol:

$$RS[rs]^{T+1}.tag \quad = \quad ROBtail^{T}$$

According to lemma 6.1, this is equivalent to $0^{\vartheta} \oplus sIissue(T)$. This is the definition of $I\_tag(i)$.

If no instruction is issued into reservation station rs during cycle T, we apply the induction premise. QED

Lemma 6.29 ► If there is an instruction in a producer, the tag in the producer matches the tag of the instruction.

$$P[fu]^{T}.full \quad \Longrightarrow \quad P[fu]^{T}.tag = I\_tag(sIP(fu,T))$$

PROOF We show this claim by induction on T. For $T = 0$, there is nothing to show because the producer is not full in the initial configuration.

For $T+1$, we show the claim as follows: For the case that the instruction in the producer did not change from cycle T to $T+1$, we apply the induction premise.

If a new instruction moved into the producer, we conclude the claim by making the following assumption: if the function unit gets correct tags as inputs for cycles T' with $T' \leq T$, this implies that the function unit passes the correct tag during cycle T. We will later on describe how to verify that property of the function units. We show that the function units get correct tags for T' with $T' \leq T$ using lemma 6.28. QED

Lemma 6.30 ► The tag on the CDB matches the tag of the instruction on the CDB.

$$CDB^{T}.valid \quad \Longrightarrow \quad CDB^{T}.tag = I\_tag(sICDB(T))$$

PROOF We assume that we only complete instructions from producers that are full. Thus, we can apply lemma 6.29. The tag on the CDB matches the tag from the producer. Furthermore, the instruction on the CDB matches the instruction in the producer, by definition of *sICDB*. QED

6.5.7 Data Consistency Criterion

In this section we describe our data consistency criterion for the Tomasulo protocols. We define a formal notion for the correct input and output values of an instruction. We do this by defining an abstract machine that processes an instruction with each transition. We call this machine abstract specification machine (aS). The configuration set of this machine consists of the registers.

Given an instruction (configuration of this machine), we define the correct value of a source register r to be the value of the register r if $r \neq 0$ and to be zero if $r = 0$:

$$source(i,r) \quad := \quad \begin{cases} 0 & : \quad r = 0 \\ c^i_{aS}.R & : \quad \text{otherwise} \end{cases}$$

The function $source(i)$ maps an instruction to the values of all source operands. Remember that $S(i,x)$ denotes the number of the register of source operand x. Let s denote the number of source registers.

$$source: \mathbb{N} \longrightarrow \mathtt{W}\;(R)^s$$

$$source(i)(x) \quad := \quad source(i, S(i,x))$$

Let f_i be the function that maps the values of the source operands of instruction I_i to the values of the destination operands unless we have $issue\_with\_result(i)$. Let d denote the number of destination registers.

$$f_i : \mathtt{W}\;(R)^s \longrightarrow \mathtt{W}\;(R)^d$$

Thus, the result of instruction I_i is:

$$result(i,r) \quad := \quad \begin{cases} issue\_result(i) & : \quad issue\_with\_result(i) \\ f_i(source(i)) & : \quad \text{otherwise} \end{cases}$$

This allows defining the configurations of the abstract specification machine. We start with an initial configuration c^0_{aS} and proceed using f. If instruction $i-1$ has register r as destination register, then we take the the new value of $R[r]$ from the result of I_{i-1}. If not so, we take the value from the old configuration.

$$c^i_{aS}.R[r] \quad := \quad \begin{cases} c^0_{aS}.R[r] & : \quad i = 0 \\ result(i-1)[e(r)] & : \quad i \neq 0 \wedge dest(i-1,r) \\ c^{i-1}_{aS} & : \quad \text{otherwise} \end{cases}$$

Proof Strategy We will show the correctness of a DLX implementation with Tomasulo scheduler as follows:

- We will show that a machine implementing the Tomasulo protocols given in the previous sections simulates the abstract machine *aS*. This is the hardest part of the proof.
- We will show that the DLX implementation with Tomasulo scheduler implements the Tomasulo protocols.

We will now conclude several trivial properties of the abstract specification machine *aS*.

Lemma 6.31 ▶ If instruction I_i has no destination register $R[r]$, then $R[r]$ is not changed by instruction I_i.

$$\overline{dest(i,r)} \quad \Longrightarrow \quad R[r]_{aS}^{i+1} = R[r]_{aS}^{i}$$

The proof is done by expanding the definition of $R[r]_{aS}^{i+1}$.

Definition 6.3 ▶ $\mathsf{L}(i,r)$ Let the predicate $\mathsf{L}(i,r)$ hold iff there is an instruction $j < i$ such that instruction I_j has destination register r.

$$\mathsf{L}(i,r) \quad :\Longleftrightarrow \quad \exists j < i : dest(j,r)$$

Lemma 6.32 ▶ Let i and $j \leq i$ be instructions. If $\mathsf{L}(j,r)$ holds, so does $\mathsf{L}(i,r)$.

$$j \leq i \wedge \mathsf{L}(j,r)) \quad \Longrightarrow \quad \mathsf{L}(i,r)$$

This holds by definition of the predicates.

Definition 6.4 ▶ $last(i,r)$ Let $\mathsf{L}(i,r)$ hold. Let $last(i,r)$ denote the number of the last instruction with destination register r prior to instruction I_i. Formally, this is the maximum of the set of instructions I_j with $j < i$ and $dest(j,r)$.

$$last(i,r) \quad := \quad max\{j \mid j < i \wedge dest(j,r)\}$$

This set is always non-empty because of $\mathsf{L}(i,r)$. Furthermore, the set is finite and has an upper bound. Thus, the maximum is defined if $\mathsf{L}(i,r)$ holds.

The following property is easily shown using the definition of *last* and the definition of max.

◀ Lemma 6.33

If $\mathsf{L}(i,r)$ holds, the instruction $I_{last(i,r)}$ has destination register r.

$$\mathsf{L}(i,r) \quad\Longrightarrow\quad dest(last(i,r),r)$$

◀ Lemma 6.34

Let $\mathsf{L}(i,r)$ and $i \geq 1$ hold. If instruction I_{i-1} does not have a destination register r, $\mathsf{L}(i-1,r)$ holds.

$$i \geq 1 \wedge \mathsf{L}(i,r) \wedge \overline{dest(i-1,r)} \quad\Longrightarrow\quad \mathsf{L}(i-1,r)$$

PROOF

Because $\mathsf{L}(i,r)$ holds, there must be an instruction I_j with $j < i$ and $dest(j,r)$. Since this is not instruction $i-1$, it must be an instruction with $j < i-1$. Thus, $\mathsf{L}(i-1,r)$ holds.

◀ Lemma 6.35

Let $i \geq 1$ and $\mathsf{L}(i,r)$ hold. If instruction I_{i-1} does not have a destination register r, then $last(i,r)$ is equal to $last(i-1,r)$.

$$i \geq 1 \wedge \mathsf{L}(i,r) \wedge \overline{dest(i-1,r)} \quad\Longrightarrow\quad last(i,r) = last(i-1,r)$$

PROOF

Because of $\mathsf{L}(i,r)$, $last(i,r)$ is defined. According to lemma 6.34, $\mathsf{L}(i-1,r)$ holds. Thus, $last(i-1,r)$ is defined.

Let j be $last(i,r)$. By definition of max, this number is element of $\{0,\ldots,i-1\}$. Because of $\overline{dest(i-1,r)}$, j cannot be $i-1$. Thus, j is equal to $last(i-1,r)$.

QED

◀ Lemma 6.36

Let $i \geq 1$ hold. If instruction I_{i-1} has destination register r, $last(i,r)$ is equal to $i-1$.

$$i \geq 1 \wedge dest(i-1,r) \quad\Longrightarrow\quad last(i,r) = i-1$$

This is easily shown by using the definition of max.

◀ Lemma 6.37

Let I_i and I_j with $j \leq i$ be instructions. If all instructions $I_{j'}$ with $j \leq j' < i$ do not have a destination register r, the value of $R[r]$ does not change from configuration c^i_{aS} to c^j_{aS}.

$$j \leq i \wedge (\forall j \leq j' < i : \overline{dest(j',r)} \quad\Longrightarrow\quad R[r]^i_{aS} = R[r]^j_{aS}$$

One easily concludes this using induction on i and the transition function of $R[r]$.

Lemma 6.38 ► Let $R[r]$ with $r \neq 0$ be a register and let $\mathsf{L}(i,r)$ hold. In this case, the correct source register of I_i is the result of the last instruction writing $R[r]$.

$$r \neq 0 \wedge \mathsf{L}(i,r) \quad \Longrightarrow \quad source(i,r) = result(last(i,r))[e(r)]$$

PROOF By definition of $last(i,r)$, the instructions I_j with $last(i,r) < j < i$ do not have destination register r. According to lemma 6.37, we have

$$R[r]^{i}_{aS} \quad = \quad R[r]^{last(i,r)+1}_{aS}$$

The left hand side is $source(i,r)$ by definition, and the right hand side is $result(last(i,r))[e(r)]$ by definition of $R[r]^{last(i,r)+1}_{aS}$. QED

Lemma 6.39 ► Let there not be an instruction that is issued during cycle T with destination $R[r]$. This implies that the value of source register r of instruction $I_{issue(T)}$ matches the value of source register r of instruction $I_{issue(T+1)}$.

$$\begin{aligned} & \overline{issue(T) \wedge dest(sIissue(T),r)} \\ \Longrightarrow \quad & source(sIissue(T),r) = source(sIissue(T+1),r) \end{aligned}$$

PROOF If $issue(T)$ does not hold, we have $sIissue(T) = sIissue(T+1)$ and the claim obviously holds.

If $issue(T)$ holds, we apply lemma 6.37 and expand the definition of *source*. QED

6.5.8 Forwarding Tags Consistency

The Tomasulo scheduling algorithm does forwarding at two places: 1) during issue, we forward from the CDB and from the ROB, 2) while in a reservation station, we forward from the CDB.

Both forwarding from the ROB and from the CDB is done using the tag. We will now show that the tags used for forwarding are correct.

◄ Lemma 6.40

Let I_i be the instruction in issue stage during cycle T. If a register $R[r]$ is marked as "not valid" during cycle T in the producer table, there is an instruction prior to instruction I_i that writes $R[r]$ and the tag of the register in the producer table is the tag of the last instruction prior instruction $I_{sIissue(T)}$ writing $R[r]$.

$$\begin{aligned} & sIissue(T) = i \wedge \overline{R[r]^T.valid} \\ \Longrightarrow\quad & \mathsf{L}(i,r) \wedge R[r]^T.tag = I\_tag(last(i,r)) \end{aligned}$$

PROOF

We verify that claim by induction on T. For $T = 0$, there is nothing to show because we make the valid bits of the registers active in the initial configuration.

For $T+1$, we conclude the claim as follows: In case $R[r]^{T+1}.valid$ holds, there is nothing to show. Thus, let $R[r]^{T+1}.valid$ not hold. We distinguish three cases:

- If an instruction with destination register $R[r]$ is issued during cycle T, we easily assert $\mathsf{L}(i,r)$, since instruction $sIissue(T)$ satisfies the claim.

  We assert $R[r]^T.tag = I\_tag(last(i,r))$ as follows: we apply lemma 6.36, which states:

  $$last(i,r) \quad = \quad i-1$$

  Thus, we have to show:

  $$R[r]^{T+1}.tag \quad \overset{!}{=} \quad I\_tag(i-1)$$

  During issue, the ROB tail pointer is stored in $R[r].tag$. Thus, the claim is equivalent to:

  $$ROBtail^T \quad \overset{!}{=} \quad I\_tag(i-1)$$

  According to the definition of I_tag and lemma 6.1, this is equivalent to:

  $$\begin{aligned} 0^\theta \oplus sIissue(T) \quad &\overset{!}{=} \quad 0^\theta \oplus (i-1) \\ sIissue(T) \quad &\overset{!}{=} \quad i-1 \end{aligned}$$

This is concluded using the fact that $i = issue(T+1)$ holds, and by expanding the definition of $issue(T+1)$, and the fact that $issue(T)$ holds.

- If an instruction with no destination register $R[r]$ is issued during cycle T, consider $R[r]^T.valid$. If $R[r]^T.valid$ holds, this implies that $R[r]^{T+1}.valid$, which is a contradiction.

  Thus, $R[r]^T.valid$ does not hold. This allows applying the induction premise for instruction I_{i-1} and we get:

  $$\mathsf{L}(i-1,r) \wedge R[r]^T.tag = I\_tag(last(i-1,r))$$

  We conclude $\mathsf{L}(i,r)$ from $\mathsf{L}(i-1,r)$ using lemma 6.32.

  As the instruction that is issued during cycle T does not have a destination register $R[r]$, we have $R[r]^{T+1}.tag = R[r]^T.tag$, which transforms the claim into:

  $$R[r]^T.tag \quad \overset{!}{=} \quad I\_tag(last(i,r))$$

  Thus, it is left to show that $last(i-1,r) = last(i,r)$ holds. This is concluded using lemma 6.35.

- If no instruction is issued during cycle T, we assert that $R[r]^T.valid$ does not hold as in the case above. This allows applying the induction premise, which concludes the claim.

QED

The following lemma will be used for the induction step for the proof of lemma 6.42.

Lemma 6.41 ▶ Let reservation station rs be full during cycle $T+1$ and let the operand x be not valid. There are two possible reasons for this: 1) this was already true during cycle T, and 2) an instruction was issued into the reservation station during cycle T.

$$\begin{aligned} & RS[rs]^{T+1}.full \wedge \overline{RS[rs]^{T+1}.op[x].valid} \\ \Longrightarrow \quad & (RS[rs]^T.full \wedge \overline{RS[rs]^T.op[x].valid}) \vee \\ & (issue(T) \wedge issue\_rs(T,rs)) \end{aligned}$$

One easily asserts this claim by applying the definition of the issue protocol. Full bits of reservation stations are only set by the issue protocol, the valid bit of the operand is only cleared by the issue protocol.

The following lemma will be used to argue the correctness of data that is forwarded into a reservation station.

Lemma 6.42 Let reservation station rs be full and let instruction I_i be in this reservation station. Let operand x be not valid, and let r be $S(i,x)$. This implies that r is not zero, and that there is an instruction prior to instruction I_i with destination $R[r]$ and the tag of operand x is the tag of the last instruction prior to I_i with destination $R[r]$.

$$\begin{aligned} & RS[rs]^T.full \wedge sIRS(rs,T) = i \wedge \overline{RS[rs]^T.op[x].valid} \\ \implies\ & r \neq 0 \wedge \mathsf{L}(i,r) \wedge RS[rs]^T.op[x].tag = I\_tag(last(i,r))) \end{aligned}$$

PROOF One asserts this claim by induction on T. For $T = 0$, there is nothing to show since the full bits of the reservation stations are not set in the initial configuration.

For $T+1$, we show the claim by applying lemma 6.41. Consider the case that an instruction is issued into the reservation station during cycle T. In this case, the claim is easily concluded using lemma 6.40 (correctness of the tags in the producer tables).

If no instruction is issued into the reservation station during cycle T, the tag in the reservation station does not change and we have

$$RS[rs]^T.full \wedge \overline{RS[rs]^T.op[x].valid}$$

according to lemma 6.41. This allows applying the induction premise, which concludes the claim. QED

6.5.9 Tag Uniqueness

We will now show the tag uniqueness properties for the different places tags are used in the Tomasulo machine.

Recall that this property was shown in lemma 6.21. This lemma uses $\tau(i,T)$ as premise. Thus, we use "tag is unique" and $\tau(i,T)$ synonymously.

Lemma 6.43 Let I_i be the instruction in issue stage and let the valid bit of register $R[r]$ be not set. This implies that there is an instruction prior to I_i writing $R[r]$ and the tag of the last such instruction is unique.

$$sIissue(T) = i \wedge \overline{R[r]^T.valid} \quad \implies \quad \mathsf{L}(i,r) \wedge \tau(last(i,r),T)$$

PROOF This claim is concluded by induction on T. For $T = 0$, there is nothing to show since we make the valid bits of all registers set in the initial configuration.

For $T+1$, we apply lemma 6.40, which states that there is an instruction prior to I_i writing $R[r]$ and that the tag in the producer table is the tag of instruction $j := last(i,r)$. In order to show the uniqueness of the tag, we have to assert that instruction I_j is already issued but not yet terminated.

One easily asserts that instruction I_j is already issued by definition of $last(i,r)$.

We show that instruction I_j is not yet terminated by distinguishing two cases:

1. If an instruction with destination $R[r]$ is issued during cycle T, we show that $j = i-1$ holds using lemma 6.36. This instruction cannot be terminated in cycle $T+1$, because this is a contradiction to lemma 6.9.

2. If no instruction with destination $R[r]$ is issued during cycle T, we assert that the valid bit of register $R[r]$ is not set during cycle T:

   $$\overline{R[r]^T.valid}$$

   This allows applying the induction premise for the instruction issued during cycle T (instruction $sIissue(T)$). Thus, we have:

   $$\tau(last(sIissue(T),r),T)$$

   If $issue(T)$ does not hold, we have $sIissue(T) = i$ and the claim is concluded. Thus, let $issue(T)$ hold. We already showed the claim for the case that instruction $sIissue(T)$ has destination register $R[r]$. For the case it does not have such a destination register, we apply lemma 6.35, which states that

   $$last(i,r) \quad = \quad last(sIissue(T),r)$$

   holds. Thus, we have:

   $$\tau(j,T)$$

   We therefore know that instruction I_j did not terminate before cycle T. It is left show show that it does not terminate during cycle T. Assume it does terminate during cycle T. One easily asserts that the tag in the producer table of register $R[r]$ is the tag of instruction I_j since it is unique according to the induction premise.

Thus, according to the writeback protocol, the valid bit of $R[r]$ is set during cycle T. This is a contradiction to the fact that $R[r]^{T+1}.valid$ does not hold.

QED

◄ Lemma 6.44

Let I_i be in reservation station rs and let that reservation station be full. This implies that the tag of instruction I_i is unique.

$$RS[rs]^T.full \implies \tau(sIRS(rs,T),T)$$

PROOF

One easily concludes that instruction I_i is in phase "in RS", as formally defined above. According to lemma 6.27, the instruction cannot be in two different phases during cycle T. Thus, it cannot be in "not issued" phase, which allows concluding that it is already issued.

Furthermore, it cannot be in "terminated" phase. Thus, $\tau(i,T)$ holds.

QED

◄ Lemma 6.45

Let I_i be an instruction in a full reservation station. Let x be a source operand that is not valid, and $r := S(i,x)$ be the source register. There is an instruction prior to I_i writing $R[r]$. Let I_j be the last instruction prior to instruction I_i that writes $R[r]$.

We claim that instruction I_j is in one of the following phases: 1) it is in a reservation station, 2) it is in a function unit, or 3) it is in a producer.

PROOF

This claim is shown by induction on T. For $T = 0$, there is nothing to show since the reservation stations are not full in the initial configuration.

For $T+1$, we conclude the clain as follows: According to lemma 6.41, there are two cases: an instruction is issued into reservation station rs during cycle T or the instruction already was in the reservation station during cycle T.

- If an instruction is issued into the reservation station during cycle T, one easily asserts that the valid bit of the source register cannot be active (otherwise, the valid bit of the reservation station source operand is set and we have nothing to show). This allows applying lemma 6.40, which states that the tag of the last instruction writing the register is in the producer table. According to lemma 6.43, the tag is unique, i.e., instruction I_j is already issued and has not yet terminated. Futhermore, the instruction is not in the "in ROB" phase during cycle T and not on the CDB (otherwise, the valid bit of the reservation station source operand is set and we have nothing to show). Thus, it must be in a reservation station, function unit or producer during cycle T.

We conclude the claim as follows: if the instruction is in a reservation station, we use lemma 6.59 in order to conclude that it either stays in that phase or enters a function unit. This concludes the claim.

If the instruction is in a function unit, we use lemma 6.59 in order to conclude that it either stays in that phase or enters a producer. This concludes the claim.

If the instruction is in a producer, we use 6.59 in order to conclude that it either stays in that phase or moves into the ROB. The last case cannot happen, since this is a contradiction to the fact that the valid bit of the operand is not active. This is easily concluded since the tag of I_j is valid because the instruction is in the "in producer" phase.

- If no instruction is issued into the reservation station during cycle T, one applies the induction premise. The induction premise states that instruction I_j is in a reservation station, a function unit, or in a producer. After that, the claim is concluded as in the case above.

QED

Lemma 6.46 ► Let I_i be an instruction in a full reservation station. Let x be a source operand that is not valid, and $r := S(i,x)$ be the source register. There is an instruction prior to I_i writing $R[r]$. Let I_j be the last instruction prior to instruction I_i that writes $R[r]$. The tag of that instruction is unique.

$$\begin{aligned} & RS[rs]^T.full \wedge sIRS(rs,T) = i \wedge \overline{RS[rs]^T.op[x].valid} \\ \Longrightarrow \quad & \mathsf{L}(i,r) \wedge \tau(j,T) \end{aligned}$$

PROOF One easily asserts this lemma by applying lemma 6.45. According to lemma 6.27, the phases exclude each other. Thus, I_j cannot be in "not issued" or "terminated" phase, which concludes the claim.

QED

Lemma 6.47 ► Let I_i be in producer fu and let that producer be full. This implies that the tag of instruction I_i is unique.

$$P[fu]^T.full \quad \Longrightarrow \quad \tau(sIP(fu,T),T)$$

PROOF The instruction in the producer is in the "in producer" phase. According to lemma 6.27, the phases exclude each other. Thus, the instruction cannot be in "not issued" or "terminated" phase, which concludes the claim.

◄ Lemma 6.48

The tag of the instruction on the CDB is unique.

$$CDB^T.valid \implies \tau(sICDB(T), T)$$

One easily asserts this lemma by expanding the definition of $sICDB(T)$ and by applying lemma 6.47.

6.5.10 Data Consistency Invariants

In order to show data consistency, we claim a set of invariants. As done in the previous chapters, we will show that all these invariants hold by induction on T. The invariants are taken from [MPK00].

◄ Invariant 6.1

Let instruction I_i be in the issue stage. Let $r \neq 0$ be a register. Let the valid bit of register $R[r]$ be set. In this case, the register data is correct.

$$sIissue(T) = i \wedge r \neq 0 \wedge R[r]^T.valid \implies R[r]^T.data = source(i, r)$$

◄ Invariant 6.2

Let reservation station rs be full and let instruction I_i be in reservation station rs. If an input operand of the reservation station is valid, the value in the operand registers is the correct source operand of instruction I_i.

$$\begin{aligned} & sIRS(rs, T) = i \wedge RS[rs]^T.full \wedge RS[rs]^T.op[x].valid \\ \implies \; & RS[rs]^T.op[x].data = source(i)(x) \end{aligned}$$

After all operands are valid, the instruction is passed to the function unit. Once the instruction leaves the function unit, the result is stored in a producer. The following invariant asserts that the producer holds the correct result.

◄ Invariant 6.3

Let producer p be full and let instruction I_i be in producer fu. The result in this producer is the result of instruction I_i.

$$sIP(fu, T) = i \wedge P[fu]^T.full \implies P[fu]^T.result = result(i)$$

Once there is an instruction in a producer, the producer requests the CDB. After the request is acknowledged, the result is put on the CDB.

Invariant 6.4 ► Let I_i be on the CDB. The result on the CDB is the result of I_i.

$$sICDB(T) = i \wedge CDB^T.valid \quad \Longrightarrow \quad CDB^T.result = result(i)$$

While on the CDB, the results are written into the ROB. The following invariant asserts that the results in the ROB are correct.

Invariant 6.5 ► Let I_i be in ROB entry *tag* and let that entry be valid. This implies that the result in the ROB entry is the result of instruction I_i.

$$\begin{aligned} & sIROB(tag,T) = i \wedge ROB[tag]^T.valid \\ \Longrightarrow \quad & ROB[tag]^T.result = result(i) \end{aligned}$$

We now show lemmas that form the induction step of the invariant proof.

Lemma 6.49 ► Let invariant 6.3 (producer data consistency) hold during cycle T. This implies that invariant 6.4 (CDB data consistency) holds during cycle T.

PROOF By definition, $CDB^T.valid$ only holds iff we complete an instruction, i.e., iff $completion(T)$ holds. The producer the instruction we complete is in, is denoted by $compl\_p(T)$. We assume that we only complete an instruction in a producer, if that producer is full. Thus,

$$P[compl\_p(T)]^T.full$$

holds. This allows applying invariant 6.3, which states that the result in the producer is correct:

$$P[compl\_p(T)]^T).result = result(sIP(compl\_p(T),T))$$

The term on the left hand side is the result on the CDB by definition.

$$CDB^T.result = result(sIP(T))$$

By definition of $sICDB(T)$, we have $sICDB(T) = sIP(compl\_p(T),T)$. This concludes the claim.

◀ Lemma 6.50

Let invariant 6.5 (ROB data consistency) and invariant 6.4 (CDB data consistency) hold during cycle T. This implies that invariant 6.5 (ROB data consistency) holds during cycle $T+1$.

PROOF

In order to show the claim, we distinguish three cases:

1. Consider the case that an instruction is issued into ROB entry *tag* during cycle T, i.e., we have:

$$issue(T) \wedge ROBtail^T = tag$$

In this case, the ROB entry *tag* is valid iff we have the result of the instruction available during issue, i.e., if $issue\_with\_result(T)$ holds. Thus, there is nothing to show unless $issue\_with\_result(T)$ holds. We easily conclude that $sIROB(tag,T+1)$ is equal to $sIissue(T)$. Thus, the result in the ROB is correct by definition.

2. Consider the case that we do not issue an instruction into ROB entry *tag* during cycle T and that we receive a result from the CDB during cycle T, i.e.:

$$CDB^T.valid \wedge CDB^T.tag = tag$$

In this case, the result on the CDB is stored in the ROB and we have to argue its correctness:

$$\begin{aligned} result(sIROB(tag,T+1)) &\overset{!}{=} ROB[tag]^{T+1}.result \\ &\overset{!}{=} CDB^T.result \end{aligned}$$

According to invariant 6.4 (CDB data consistency), we have:

$$CDB^T.result \quad = \quad result(sICDB(T))$$

Thus, the claim holds if we show $sIROB(tag,T+1) = sICDB(T)$, i.e., it is left to show that the tag maps to the correct instruction. These arguments are weak in [MPK00].

We show this formally using lemma 6.48. Lemma 6.48 states that

$$\tau(sICDB(T),T)$$

holds. This allows applying theorem 6.21, which states:

$$sIROB(I\_tag(sICDB(T)),T) \quad = \quad sICDB(T)$$

Thus, it is left to show:

$$sIROB(tag,T+1) \quad \overset{!}{=} \quad sIROB(I\_tag(sICDB(T)),T)$$

According to lemma 6.30, we have $tag = I\_tag(sICDB(T))$. This transforms the claim into:

$$sIROB(tag,T+1) \quad \overset{!}{=} \quad sIROB(tag,T)$$

This is concluded by expanding the definition of $sIROB(tag,T+1)$.

3. Consider the case that no instruction is issued in ROB entry *tag* and that no result for ROB entry *tag* is on the CDB. We assert this case using invariant 6.5 for cycle T.

QED

Lemma 6.51 ▶ Let invariant 6.5 (ROB data consistency) and invariant 6.1 (register file data consistency) hold during cycle T. This implies that invariant 6.1 (register file data consistency) holds during cycle $T+1$.

PROOF We distinguish three cases:

1. Consider the case that we issue an instruction with destination r during cycle T. In this case, the valid bit $R[r]^{T+1}.valid$ cannot hold and there is nothing to show.

2. Consider the case that we writeback an instruction with destination r during cycle T and let the valid bit of $R[r]$ be not active during cycle T. We only do this writeback if the ROB entry that the ROB head pointer points to is valid. According to invariant 6.5, this implies that the result in the rob entry is the result of the instruction. This transforms the claim into:

$$result(sIROB(ROBhead^T,T))[e(r)] \quad \overset{!}{=} \quad source(i,r)$$

The tag of $R[r]$ matches the the ROB head pointer, since otherwise $R[r]^{T+1}.valid$ cannot hold and there is nothing to show.

According to lemma 6.40, that tag is equal to the tag of the last instruction prior to instruction $issue(T)$ that writes $R[r]$. This transforms the claim into:

$$result(sIROB(I\_tag(last(sIissue(T),r),T))[e(r)] \overset{!}{=} source(i,r)$$

According to lemma 6.43, that tag is unique. This allows applying lemma 6.21, which transforms the claim into:

$$result(last(sIissue(T),r))[e(r)] \overset{!}{=} source(i,r)$$

According to lemma 6.39, we have:

$$source(sIissue(T),r) = source(sIissue(T+1),r)$$

This transforms the claim into:

$$result(last(sIissue(T),r))[e(r)] \overset{!}{=} source(sIissue(T),r)$$

This is concluded using lemma 6.38.

3. If we neither issue an instruction with destination $R[r]$ nor write-back an instruction with destination $R[r]$ with $\overline{R[r]^T.valid}$, assume $R[r]^T.valid$ does not hold. In this case, valid bit $R[r]^{T+1}.valid$ cannot hold and there is nothing to show.

   Thus, $R[r]^{T+1}.valid$ holds. The claim is:

$$R[r]^T.data \overset{!}{=} source(i,r)$$

   After applying the induction premise, this is transformed into:

$$source(sIissue(T),r) \overset{!}{=} source(i,r)$$

   We assert this using lemma 6.39. QED

◄ Lemma 6.52

Let invariant 6.3 (producer data consistency) hold during cycle T and invariant 6.2 (reservation station data consistency) hold during cycles T' with $T' \leq T$. This implies that invariant 6.3 (producer data consistency) holds during cycle $T+1$.

PROOF One concludes this claim as follows: if an instruction moves into the producer during cycle T, we make the assumption that the function unit delivers a correct result given that it got correct inputs during all cycles $T' \leq T$. This is easily asserted using invariant 6.2 (reservation station data consistency). For this, we have to assume that we only dispatch instructions with valid operands.

If no instruction moves into the producer during cycle T, we conclude

$$sIP(fu,T) \quad = \quad sIP(fu,T+1).$$

Furthermore, we conclude that $P[fu]^T.full$ holds and that the value in $P[fu].result$ does not change from cycle T to cycle $T+1$. This allows concluding the claim from invariant 6.3 (producer data consistency) for cycle T.

QED

Lemma 6.53 ► If the tag on the CDB matches the tag of an instruction I_i and the tag of that instruction is unique, then the instruction on the CDB is instruction I_i.

$$CDB^T.valid \wedge CDB^T.tag = I\_tag(i) \wedge \tau(i,T) \quad \Longrightarrow \quad sICDB(T) = i$$

This is easily shown using lemma 6.30 (uniqueness of CDB tag) and 6.22.

The following two lemmas are used to argue the data consistency of the reservation stations (invariant 6.2). Since this is where all forwarding is done, this is the most complicated part of the proof. We therefore split the proof of invariant 6.2 into two lemmas.

The first lemma shows the claim for the case the operand reading is done in the issue stage. The second lemma shows the claim for the case the operand reading is done in the reservation station. The same case split is also done in [MPK00].

Lemma 6.54 ► Let invariant 6.2 (reservation station data consistency) and invariant 6.1 (register file data consistency) and invariant 6.4 (CDB data consistency) and invariant 6.5 (ROB data consistency) hold during cycle T.

If an instruction is issued into reservation station rs, invariant 6.2 for reservation station rs holds during cycle $T+1$.

PROOF We show this claim by a case split on the location the operand x is read from. Let I_i be the instruction in the issue stage and let $r = S(i,x)$ be a shorthand for the number of the register we read.

- If $r = 0$ holds, we read zero and the claim holds by definition of $source(i,0)$.
- **Reading from the register file:** This is done only iff $R[r]^T.valid$ holds. This allows applying invariant 6.1. This concludes the claim.
- **Reading from the CDB:** This is done only iff $R[r]^T.valid$ does not hold. This allows applying lemma 6.40, which states that the tag in the producer table is the tag of the last instruction writing $R[r]$. According to lemma 6.43, that tag is unique. This allows applying lemma 6.53, which states that the last instruction writing $R[r]$ is on the CDB. According to lemma 6.4, the result on the ROB is the result of that instruction.

  Thus, it is left to show:

  $$result(last(i,r))[e(r)] \overset{!}{=} source(i)(x)$$

  We assert this using lemma 6.38.
- **Reading from the ROB:** We repeat the arguments from the case above in order to show that the tag in the producer table is the tag of the last instruction writing $R[r]$. Let tag denote the tag. This tag is unique, and we therefore know that the instruction in ROB entry tag is the last instruction writing $R[r]$ (lemma 6.21). According to invariant 6.5, the result in the ROB is the result of this instruction. As before, we conclude the claim using lemma 6.38. QED

◀ Lemma 6.55 Let invariant 6.2 (reservation station data consistency) and invariant 6.4 (CDB data consistency) hold during cycle T.

If no instruction is issued into reservation station rs, invariant 6.2 for reservation station rs holds during cycle $T+1$.

PROOF Let x be a source operand number. If the valid bit of operand x holds during cycle T, one just applies invariant 6.2 for cycle T.

If not so, we snoop an operand from the CDB or we have nothing to show. The argue the correctness of CDB snooping as follows: Let i be the number of the instruction in reservation station rs during cycle $T+1$. The claim of invariant 6.2 is:

$$RS[rs]^{T+1}.op[x].data \overset{!}{=} source(i)(x)$$

By expanding the definition of $RS[rs]^{T+1}.op[x].data$ on the left hand side, this is transformed into:

$$CDB^{T}.result[e(S(i,x))] \overset{!}{=} source(i)(x)$$

Invariant 6.4 states:

$$CDB^{T}.result = result(sICDB(T))$$

Thus, the claim is transformed into:

$$result(sICDB(T))[e(S(i,x))] \overset{!}{=} source(i)(x)$$

Thus, it is left to show that the result of the instruction on the CDB is the source operand of the instruction in the reservation station. This is argued as follows: According to lemma 6.38 with instructions I_i and $I_{sICDB(T)}$, the claim above holds if we show the premises of the lemma. These premises are:

$$S(i,x) \neq 0 \wedge \mathsf{L}(i,S(i,x)) \wedge last(i,S(i,x)) = sICDB(T))$$

Thus, we have to show that the source register is not register 0 and that there is an instruction before I_i that writes the register. One easily argues this using invariant 6.42.

Furthermore, one has to show that the last instruction before I_i writing the register is the instruction on the CDB. We argue this using the fact that the tag on the CDB matches the tag stored in the reservation station for the operand. According to invariant 6.42, that tag is the tag of the last instruction writing the register.

Lemma 6.44 states that the tags in the reservation stations are unique. This allows applying lemma 6.53, which concludes the claim. QED

The following lemma combines the claims of lemma 6.54 and lemma 6.55.

Lemma 6.56

Let invariant 6.2 (reservation station data consistency) and invariant 6.1 (register file data consistency) and invariant 6.4 (CDB data consistency) and invariant 6.5 (ROB data consistency) hold during cycle T. This implies that invariant 6.2 for reservation station rs holds during cycle $T+1$.

This claim is shown using lemma 6.54 and lemma 6.55.

Theorem 6.57

The invariants 6.1 to 6.5 hold.

PROOF

We show this claim by induction on T. We omit the simple arguments for cycle $T = 0$.

The claim for $T+1$ is shown by applying lemma 6.50, 6.51, 6.52, and 6.56 for cycle T and lemma 6.49 for cycle $T+1$.

QED

Theorem 6.58

A machine implementing the Tomasulo protocols above, satisfies the following data consistency criterion:

$$R[r]_{aI}^{T}.data \quad = \quad R[r]_{aS}^{sIwriteback(T)}$$

Since all speculation registers are output of the writeback stage, this criterion exactly matches the data consistency criterion as proposed for the in-order pipelined machine.

PROOF

Given the data consistency invariants above, one easily shows this claim by induction on T. For $T = 0$, we have $sIwriteback(T) = 0$ and we therefore have the claim that the registers are in the initial configuration. We assume this.

For $T+1$, we show the claim as follows: In case $writeback(T)$ does not hold, one easily asserts that

$$sIwriteback(T) \quad = \quad sIwriteback(T+1)$$

holds and that the registers do not change from cycle T to $T+1$. Thus, the claim is concluded using the induction premise. Let i be a shorthand for $sIwriteback(T)$.

In case $writeback(T)$ holds, we do a case split on $dest(i,r)$. If $dest(i,r)$ does not hold, we easily assert the claim using the induction premise.

If $dest(i,r)$ and $writeback(T)$ hold, we have the following claim:

$$ROB^T[ROBhead(T)].result[e(r)] \quad \overset{!}{=} \quad R[r]^{i+1}_{aS}$$

The register on the right hand side expands to the result of instruction I_i:

$$ROB^T[ROBhead(T)].result[e(r)] \quad \overset{!}{=} \quad result(i)[e(r)]$$

We assert this using invariant 6.5 for instruction I_i and tag $ROBhead(T)$, which holds according to theorem 6.57.

The claim of invariant 6.5 concludes the claim above. It is left to show the premises of invariant 6.5, which are:

$$sIROB(ROBhead(T),T) = i \wedge ROB[ROBhead(T)]^T.valid$$

We assert the first part of this claim using lemma 6.24. The valid bit of the ROB entry holds since we assume that we only writeback if the valid bit holds. QED

6.6 Liveness

We propose the following liveness criterion for the Tomasulo machine with reorder buffer: we will show that all instructions will eventually be in the terminated phase.

We use a similar liveness proof strategy as employed in chapter 4. We show our claim by induction on T. Thus, the induction step is: given all instructions up to instruction I_{i-1} terminated, instruction I_i eventually terminates.

Informally, we show this as follows: We will show that instruction I_i must be in a phase. According to lemma 6.27, that phase is unique. We do a case split on the phase of instruction I_i. If instruction I_i is in “in ROB” phase, we easily assert that it eventually terminates. If instruction I_i is in a producer, we assert that it will move into “in ROB” phase. We then conclude the claim as before. These arguments are continued until all phases are covered.

We will now formalize this proof.

◀ Lemma 6.59

If instruction I_i is in phase p during cycle T, this implies that it is in one of the successor phases of phase p during cycle $T+1$.

$$p(i,T) \quad \Longrightarrow \quad \bigvee_{p' \in succ(p)} p'(i,T+1)$$

PROOF

We show this claim exemplary for phase "not issued". Thus, we have to show that instruction I_i is still not issued, in a reservation station, or in the ROB during cycle $T+1$.

- If $issue(T)$ and $sIissue(T)=i$ does not hold, one easily concludes that instruction I_i stays in "not issued" phase.
- If $issue(T)$ and $sIissue(T)=i$ holds and $issue\_with\_result(i)$ holds, one easily shows that instruction I_i is in the reorder buffer during cycle $T+1$.
- Otherwise, we assume that there is a reservation station rs such that $issue\_rs(T,rs)$ holds. One easily verifies that instruction I_i is in that reservation station during cycle $T+1$.

QED

◀ Lemma 6.60

Instruction I_i is in at least one phase during cycle T.

PROOF

The claim is concluded by induction on T. For cycle $T=0$, we conclude the claim easily since all instructions are in the "not issued" phase.

For $T+1$, we conclude as follows: According to the induction premise, instruction I_i is in at least one phase during cycle T. This allows applying lemma 6.59, which states that instruction I_i is in one of the successor phases of that phase. This concludes the claim.

QED

The following lemmas form the induction step for the liveness proof.

◀ Lemma 6.61

If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in "in ROB" phase, instruction I_i will eventually terminate.

PROOF Let T be the cycle given by the premise. According to the premise, instruction I_i is in "in ROB" phase during cycle T. This implies that it is not terminated yet. Since we either have $i = 0$ or the previous instruction is terminated, we have

$$i \quad = \quad sIwriteback(T)$$

We show that instruction I_i terminates during cycle T, i.e., it is left to show that $writeback(T)$ holds. As described above, we assume that we always terminate if the ROB is not empty and the ROB entry that *ROBhead* points to is valid. One easily asserts that the ROB is not empty during cycle T using that instruction I_i is in "in ROB" phase during cycle T.

According to the premise, there is a ROB entry *tag* that is valid and such that

$$sIROB(tag, T) \quad = \quad i$$

holds. Using lemma 6.11, we assert that *tag* is the tag of instruction I_i. Using lemma 6.23, we assert that entry *tag* is the entry $ROBhead(T)$ points to. Thus, the ROB entry $ROBhead(T)$ points to is valid and we writeback.

QED

Lemma 6.62 ► If producer fu is full during cycle T, then there is a cycle $T' \geq T$ such that the instruction is put on the CDB.

$$\begin{aligned} P[fu]^T.full \quad \Longrightarrow \quad & \exists T' \geq T : completion(T') \wedge \\ & compl\_p(T') = fu \wedge \\ & sIP(fu, T') = sIP(fu, T) \end{aligned}$$

PROOF In order to show this claim, we make the assumption that the CDB requests are served using a fair arbiter. One has to show that instruction I_i stays in the producer fu until the request is served using induction. For this purpose, we have to assume that the function unit does not overwrite an instruction in its producer. This is illustrated in figure 6.10. Formally, the function unit fu provides a result during cycle T iff $FU[fu]^T.valid$ holds.

The producer generates a stall signal if it is full and does not get the CDB. Let $fuins(fu, T).stall$ denote the value of this signal during cycle T.

$$fuins(fu, T).stall \quad := \quad P[fu]^T.full \wedge \overline{(completion(T) \wedge compl\_p(T) = fu)}$$

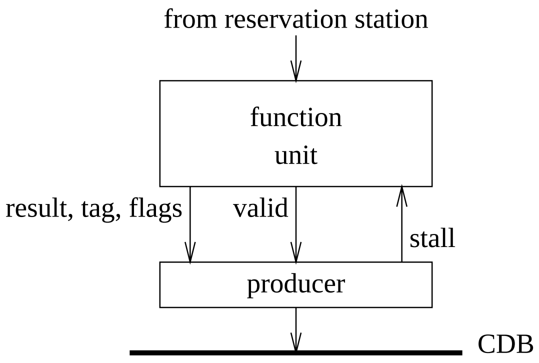


Figure 6.10 Interface between function unit and producer

We assume that the function unit does not provide a result if it gets a stall signal.

$$fuins(fu,T).stall \implies \overline{FU[fu]^T.valid}$$

Since the CDB is assigned using a fair arbiter, there is a cycle T' such that the request is acknownledged. Using the assumption on the function unit above, one easily shows by induction that the instruction stays in the producer until this happens and is not overwritten. QED

◀ Lemma 6.63

If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in "in producer" phase, instruction I_i will eventually terminate.

PROOF

Let T be the cycle from the premise of the lemma. Thus, instruction I_i is in a producer during cycle T. Let this be producer fu. We will show that this instruction eventually moves into the reorder buffer. Although we assume that all instructions prior to instruction I_i already terminated, this is not obvious. In particular, there might be instructions *later* than instruction I_i that block the CDB.

According to lemma 6.62, there is a cycle $T' \geq T$ such that the request is served and the instruction is still in the producer. Formally, we have:

$$completion(T') \wedge compl\_p(T') = fu \wedge sIP(fu,T') = sIP(fu,T)$$

One easily concludes that instruction I_i is in ROB entry $I\_tag(i)$ during cycle $T+1$. This allows applying lemma 6.61, which shows that the instruction eventually terminates. QED

Note that assuming that the CDB is allocated using a fair arbiter is not necessary for liveness, we do it for sake of simplicity only. If the CDB is not allocated using a fair arbiter, we can argue as follows: Informally, assume instruction I_i is blocked in a producer by instructions later than I_i. Since we terminate in-order, there is an upper bound for the number of these instructions, which is the number of ROB entries. Thus, instruction I_i will eventually get the CDB.

Lemma 6.64 ▶ If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in "in FU" phase, instruction I_i will eventually terminate.

PROOF Let T be the cycle from the premise of the lemma. Thus, instruction I_i is in a function unit during cycle T. Let this be function unit fu. We will show that this instruction eventually moves into the producer P. Although we assume that all instructions prior to instruction I_i already terminated, this is not obvious. In particular, there might be instructions *later* than instruction I_i that block the function unit or the producer.

In order to show this claim, we have to make the following assumption on the functional units: Given that the signal $fuins(fu,T).stall$ is finite true and that instruction I_i entered the function, there is a later cycle such that the instruction leaves the unit.

$$\begin{aligned} & \forall T' \exists T'' \geq T' : \overline{fuins(fu,T'').stall} \wedge in(i,T,fu) \\ \Longrightarrow \quad & \exists T''' \geq Tout(i,T''',fu) \end{aligned}$$

One easily asserts that the signal $fuins(fu,T).stall$ is finite true using the fact that the CDB is allocated using a fair arbiter. Thus, we have a cycle T''' such that the instruction leaves the function unit. One easily asserts that this instruction moves into the producer during that cycle. We then apply lemma 6.63 in order to conclude the claim. QED

In analogy to lemma 6.62, we show:

Lemma 6.65 ▶ If a reservation station is full during cycle T, there is a cycle $T' \leq T$ such that this reservation station is dispatched during cycle T'. Furthermore, the instruction in the RS during cycle T' is the same as during cycle T.

$$\begin{aligned} RS[rs]^T.full \quad \Longrightarrow \quad & \exists T' \geq T : dispatch\_rs(T',rs) \wedge \\ & sIRS(rs,T') = sIRS(rs,T) \end{aligned}$$

PROOF As described above, dispatching is done using a fair arbiter. The arbiter selects among the reservation stations that are full and valid. The first thing to assert is that the reservation station is valid. Assume it is not. In this case, one can apply lemma 6.45, which states that there is an instruction I_j with $j = last(i,r)$ that is in a reservation station, in a function unit, or in a producer. This is a contradition to the premise that all instructions I_j with $j < i$ are already terminated.

The function unit provides a stall singal. We denote this stall signal by $FU[fu]^T.stall$. Dispatching is only done if the function unit is not stalled. We assert this using the following assumption on function units: If the stall singal that is input of the function unit is finite true, then the stall signal that is output of the function unit is finite true.

$$\begin{aligned} & \left(\forall T' \exists T'' \geq T' : \overline{fuins(fu,T'').stall}\right) \\ \Longrightarrow \quad & \left(\forall T' \exists T'' \geq T' : \overline{fuins(fu,T'').stall}\right) \end{aligned}$$

One shows that the stall singal that is input of the function unit is finite true using that the CDB is assigned using a fair arbiter, as above. This concludes the claim. QED

◀ Lemma 6.66 If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in “in RS” phase, instruction I_i will eventually terminate.

PROOF Let T be the cycle from the premise of the lemma. We conclude this claim easily using lemma 6.65. According to this lemma, there is a cycle $T' \geq T$ such that the instruction is dispatched. There are two cases:

- The funcition unit returns the result of instruction I_i in the same cycle. In this case, one shows that the instruction moves into the “in producer” phase and uses lemma 6.63 in order to conclude the claim.
- The funcition unit does not return the result of instruction I_i in the same cycle. In this case, one shows that the instruction is in “in FU” phase during cycle $T+1$ and uses lemma 6.64 in order to conclude the claim. QED

◀ Lemma 6.67 If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in “not issued” phase, instruction I_i will eventually terminate.

PROOF We will show that the instruction eventually either moves into the ROB or into a reservation station, depending on $issue\_with\_result(i)$. This happens if the instruction is issued. We then conclude the claim using lemma 6.61 or 6.66, respectively.

Thus, it is left to show that the instruction is eventually issued. The issue stage belongs to the in-order part of the machine. As done in the previous chapters, one easily concludes that this happens if the stall signal of the stage is finite true. The issue stage is stalled if one of the following conditions hold [Krö99]:

- The ROB is full. One argues that this cannot be the case since all instructions I_j prior to I_i terminated. Thus, we have

  $$sIissue(T) = sIwriteback(T),$$

  which implies that the ROB is empty (lemma 6.4).
- There is no reservation station available. One easily concludes that all reservation stations are empty because all instructions are either in “not issued” or “terminated” phase during cycle T. Thus, they cannot be in “in RS” phase according to lemma 6.27.
- In case of the DLX, there are some instructions that require stalling issue because they depend on registers that the Tomasulo scheduler cannot forward. In case of a conditional branches or jump register instruction, one has to wait until the source register is valid. Assume it is not. In this case, we can apply lemma 6.43, which states that there is an instruction I_j with $j = last(i,r)$ that is already issued but not yet terminated. This is a contradiction.
- In case the instruction is a *movs2i* and the source register is $IEEEf$, we have to stall issue until the ROB is empty. This arises from the fact that the Tomasulo scheduling algorithm is not able to forward this register. As above, one easily concludes that the ROB is empty.
- The desings we verify are based on the designs presented in [Krö99]. The machine stalls issue until the ROB is empty in case the instruction is an rfe instruction. This arises from the hardware cost constraints. We do not have enough read ports for the SPR producer table to forward *ESR*, *EPC*, and *EDPC*. As above, one easily concludes that the ROB is empty.

QED Thus, the instruction is issued eventually, which concludes the claim.

Note that in contrast to the machine given in [Krö99], we do not have to stall issue because of busy instruction memory. This arises from the fact that our stall engine allows stalling stages indepandantly.

The following lemma forms both the induction step and induction base for the main liveness claim.

◀ Lemma 6.68

If there is a cycle such that instruction I_{i-1} either not exists or terminated, instruction I_i will eventually terminate.

PROOF

Let T be the cycle from the premise. According to lemma 6.60, instruction I_i is in a phase. If this is “not issued”, we conclude the claim using lemma 6.67. If it is “in RS”, we conclude the claim using lemma 6.66. If it is “in FU”, we conclude the claim using lemma 6.64. If it is “in producer”, we conclude the claim using lemma 6.63. If it is “in ROB”, , we conclude the claim using lemma 6.61. If it is “terminated”, the claim obviously holds.

QED

◀ Lemma 6.69

Instruction I_i eventually terminates.

PROOF

We show this claim by induction on i. For $i = 0$, we apply lemma 6.68. This is also done for the induction step.

6.7 Verifying the DLX Implementation

In this section, we show that the implementation machine I with configurations $c_I^0, \ldots$ complies with the specification.

6.7.1 Implementation Differences

We do not describe the implementation of the DLX with Tomasulo scheduler and reorder buffer, since this design is already presented in [Krö99] in detail including cost and cycle time analysis.

In this section, we describe the differences between the implementation given in [Krö99] and the implementation used for this thesis. Figure 6.11 shows an overview of the hardware.

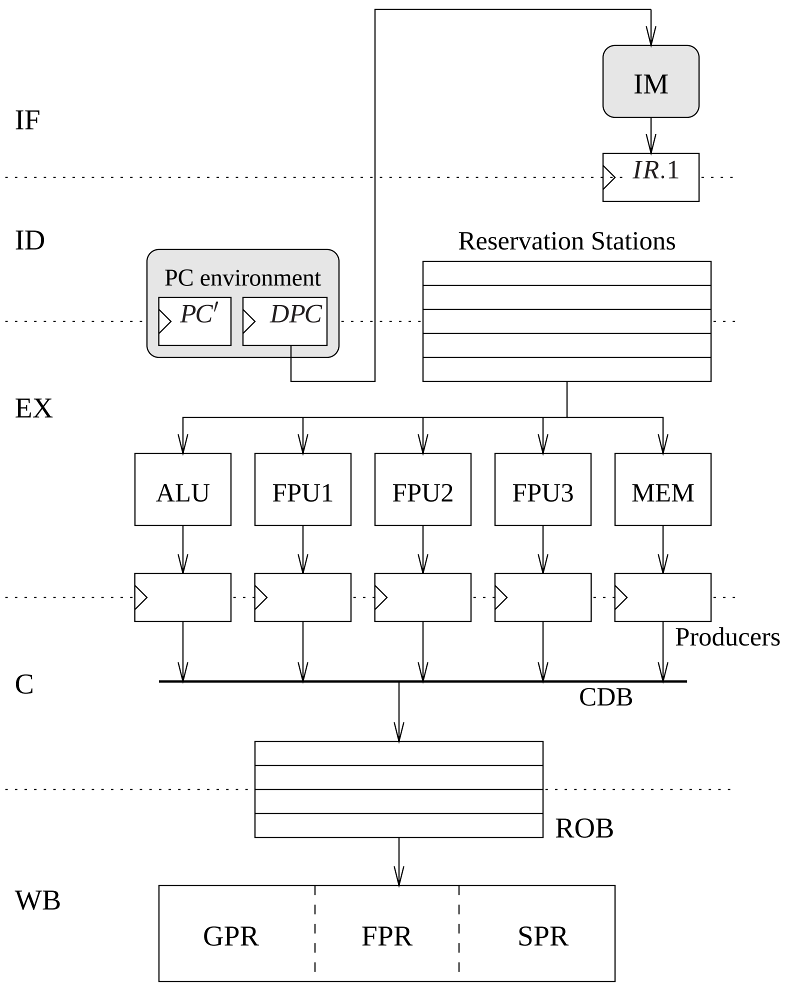


Figure 6.11 Overview of the Tomasulo Hardware

Instruction Fetch In [Krö99], the PC environment from [Lei99] is used. In order to prevent the destruction of the PC registers, stage 0 and 1 are always clocked simultaneously. We remove this limitation by using the PC environment and the stall engine described in chapter 5 (in-order machine with Delayed PC and speculation) instead.

Issue As described above, we no longer need an issue stall because of instruction memory stalls. This is a feature of the new stall engine.

Dispatch In contrast to [Krö99], the instructions do not move from one RS into another. This implementation in [Krö99] is motivated by the liveness proof, which uses the fact that one selects the oldest instruction for dispatch. We use a fair arbiter instead.

Function Units In contrast to [Krö99], we do not implement out-of-order dispatch for the memory unit. This simplifies implementing paging. As an example, consider two store instructions. The first one modifies the page table and the second one modifies a memory cell in a page that is affected. Passing the instructions in program order to the memory function unit significantly simplifies the task of building such a functional unit.

CDB In [Krö99], we allocated the CDB round-robin. We use a fair arbiter instead (this is weaker than round-robin).

6.7.2 Verifying the Instruction Fetch

In the proofs above, we assumed that the instruction fetch is correctly done. The instruction fetch mechanism in the stages 0 and 1 operates like the in-order pipelined machine as described in section 5. The verification of the forwarding of *DPC* for the instruction fetch uses the very same arguments as before.

One combines the two machines as follows: we define that we issue an instruction if the output registers of the decode/issue stages are clocked. This happens iff ue_1^T is active, as described in the previous chapters.

$$issue(T) \quad := \quad ue_1^T$$

For the correctness proof, we argue on the schedules of both parts of the machine. We argue that the schedule of the issue stage of the Tomasulo part matches the schedule of the issue stage of the in-order pipeline.

$$issue(T) \overset{!}{=} sI(1,T)$$

We show this claim by inducition on T. For $T = 0$, we have $issue(T) = 0$ and $sI(1,T) = 0$.

For $T+1$, we show the claim by a case-split on ue_1^T. If ue_1^T does not hold, the value of both scheduling functions does not change from cycle T to $T+1$ by definition. Thus, the claim is concluded using the induction premise.

If ue_1^T holds, we have

$$sI(1,T+1) = sI(1,T)+1$$

according to invariant 5.1.

By definition, $issue(T)$ holds if ue_1^T holds. Thus, we have

$$issue(T+1) = issue(T)+1$$

by definition of $issue(T+1)$. This allows concluding the claim using the induction premise.

6.7.3 Verifying IEEEf

The $IEEEf$ (IEEE flags) register is a special case for the correctness proof of the machine, since the IEEE standard [IEE85] requires that the bits in this register are sticky. Thus, if a floating point instruction generates a masked IEEE exception, the bit of this exception is set in the $IEEEf$ register. The bits that were set previously are maintained. However, in case of a *movi*2*s* instruction with destination $IEEEf$, all bits are overwritten.

One argues the data consistency of the register by induction. As induction claim we show the data consistency of the complete machine. For $T = 0$, we show the correctness of the initialization. For $T+1$, we have the data consistency upto cycle T as premise. The first thing is to argue the correctness of the interrupt mask in SR_I^T. This holds according to the induction premise. Let i be a shorthand for $sIwriteback(T)$. We distinguish three cases:

- If we do not writeback an instruction, we have

  $$sIwriteback(T) \quad = \quad sIwriteback(T+1).$$

  The registers also do not change. Thus, the claim holds.

- If we writeback an instruction that is *movi2s* with destination register $IEEEf$, the correctness is shown as above.

- If we writeback an instruction which sets IEEE flags, we have:

  $$sIwriteback(T+1) \quad = \quad i+1$$

  We assert the correctness of the flags as above using invariant 6.5. Let $ieeeflags(i)$ denote the IEEE flags generated by instruction I_i:

  $$ROB[ROBhead]^T.result[2] \quad = \quad ieeeflags(i)$$

  We assert the correctness of the old value in the IEEE flags register using the induction premise:

  $$IEEEf_I^T \quad = \quad IEEEf_S^i$$

  The new value written into the IEEE flags register is the old value *OR* the masked new one.

  $$IEEEf_I^{T+1} \quad = \quad IEEEf_I^T \vee (ROB[ROBhead]^T.result[2] \wedge SR_I^T)$$

  The claim is that this the correct value:

  $$IEEEf_I^{T+1} \quad \overset{!}{=} \quad IEEEf_I^{i+1}$$

  One expands the transition function of the specification machine on the right hand side:

  $$IEEEf_I^{T+1} \quad \overset{!}{=} \quad IEEEf_I^i \vee (ieeeflags(i+1) \wedge CA_S^i)$$

  This is easily concluded using the the equations above.

One cannot forward the $IEEEf$ register using the mechanisms described above. We therefore stall the issue stage if we read this register until the ROB is empty. As soon as the ROB is empty, we have

$$sIissue(T) \quad = \quad sIwriteback(T).$$

In this case, one easily concludes the correctness of the value in the register using the data consistency criterion above.

6.7.4 Verifying Interrupts

In this section, we describe how to verify a machine that generates interrupts. The proof method is taken from [MP00]. We show the data consistency by induction on T. For $T = 0$, we have the correctness of the initialization of the machine. Note that we do not process an interrupt during cycle T. We realize the *reset* interrupt by adjusing the initial configuration accordingly, as done in chapter 5.

Let $lastint(T)$ denote the number of the last cycle before cycle T in which we processed an interrupt **plus one** (i.e., the maximum value of $lastint(T)$ is T). In case no such cycle exists, we define $lastint(T)$ to be zero.

In order to show the claim for $T+1$, we distinguish two cases:

- If we have an interrupt during cycle T, we argue as follows: according to the induction premise, the data consistency for cycle T holds. The modifications made by an interrupt on the configuration are easy to verify using this fact.

- If we do not have an interrupt during cycle T, we argue as follows: We claim that the machine works as the abstract implementation machine without interrupts above from cycle $lastint(T)$ to cycle $T+1$. We initialize the abstract machine without interrupts using the configuration $c_I^{lastint(T)}$:

  $$c_{aI}^0 \quad := \quad c_I^{lastint(T)}$$

  We then show that the transitions made by both machines are equal from cycle $lastint(T)$ to cycle $T+1$ using induction on the cycle number. For this one uses the fact that there are no interrupts from cycle $lastint(T)$ to cycle $T+1$ by definition of $lastint(T)$.

Liveness Note that the liveness of the machine with interrupts does not require extra arguments as required in chapter 5. This arises from the fact that the instruction that generates the interrupt retires as usual and is not executed a second time. This is in contrast to the implementation of interrupts given in chapter 5.

6.8 Literature

In this chapter, we formally verify the Tomasulo scheduling algorithm with reorder buffer as presented in [MPK00]. In contrast to [MPK00], we verify the correctness using PVS and argue the uniqueness of the tags.

The parts of the hardware are based on machines described in [Lei99]. The correctness of the designs presented in [Lei99] is not verified by means of machine.

Hosabettu et.al. verify implementations using a Tomasulo scheduler both with and without reorder buffer [HGS99, HGS00, Hos00] using the completion functions approach. The verification is done using PVS at a very high level of abstraction. Gate-level designs are not verified. The functional units are very simple and do not contain cycles. Despite that, the size of the PVS proofs in [Hos00] is four times the size of the proofs for this chapter of this thesis. However, [Hos00] makes extensive use of proof strategies, which enlarges the PVS proofs significantly.

In [BBCZ98], Clarke et.al. verify out-of-order processors by combining symbolic model-checking with uninterpreded functions. In [BCRZ99], Clarke et.al. verify safety properties of a PowerPC, which implements out-of-order execution and precise interrupts.

Sawada and Hunt [SH99] verify the FM9801, which also features a reorder buffer, using the theorem proving system ACL2. The number of lemmas is enormous (nearly 4000).

Henzinger et al. [HQR98] verify a simple out-of-order processor using a model checker. McMillan [McM98] partly automates the proof by refinement of Tomasulo's algorithm presented in [DP97] with the help of compositional model checking. This technique is improved in [McM99b] by theorem proving methods to support an arbitrary register size and number of function units. In [McM99a], McMillan verifies the liveness of a machine with Tomasulo scheduler using SMV.

Arvind and Shen [AS99] describe how to apply term rewriting systems in order to model microprocessors. The authors give a simple out-of-order RISC machine with reorder buffer as an example. The authors suggest the use of tools such as PVS for verifying large, realistic machines.

Chapter 7

Perspective

This thesis covers the verification of in-order and out-of-order microprocessor designs. We develop generic theories for forwarding and speculation and demonstrate how they can be applied to DLX-like RISC processors. However, several aspects are not covered by this thesis.

7.1 Functional Units

Despite of a simple ALU, the correctness of the functional units is not covered by this thesis. This ALU needs further enhancements. For example, the ALU verified in this thesis lacks an integer multiplier. Furthermore, all ALU instructions assume signed operands. Commercial microprocessors, such as the MIPS series or the i860 support unsigned operantions, too.

For example, this affects overflow detection. The 29K has three variants for addition/subtraction operations:

1. Suppress interrupts,
2. signed (interrupt if the result is not in the range of the two's complement numbers),

3. unsigned (interrupt if the result is not in the range of the binary numbers).

This also affects test/set operations. The design presented here offers both $\leq$ and $\geq$ tests, which is superflous since one can get the desired operations by implementing one and swapping operands if necessary. The test/set instructions implemented in this tesis assume that the operands are two's complement numbers. Processors such as the MIPS RISC series also implement ALU test/set instructions that assume that the operands are unsigned binaries.

Furthermore, modern microprocessors implement instructions with saturation, i.e., if an overflow occurs, the result is set to the edge of the number range.

Floating point units are not covered at all by this thesis. The formal verification of a complete floating unit is subject of the PhD thesis of Christian Jacobi [Jac01]. The adder is verified by Christoph Berg [Ber01]. The proofs and designs are taken from [MP00] and verified using the theorem system PVS. This includes a formalization of the IEEE standard and a proof that the designs comply with this standard.

The architecture used in this thesis lacks SIMD (single instruction multiple data) instructions. For example, one can process two single precision floating point operands within a 64-bit word simultaneously with litte extra hardware cost.

Furthermore, we do not cover how to build and verify memory interfaces. The verification of a memory interface including first level on-chip cache is subject of the PhD thesis of Sven Beyer [Bey01]. This includes support for virtual memory, which is implemented using a TLB. The correctness is verified formally using PVS.

7.2 In-Order Scheduling and Forwarding

Besides the schedulers covered by this thesis, there are more scheduling methods in use in commercial microprocessors. As for in-order machines, this includes multiple instruction issue machines, i.e., pipelined in-order machines with two or more parallel pipelines. These machines are able to issue multiple instructions within the same cycle. Furthermore, we did not verify in-order schedulers for functional units with variable latency, such as result shift registers, as used in [MP00].

7.3 Speculation

The generic speculation mechanism presented in chapter 5 assumes that we have a guarantee that an instruction never rollbacks twice. However, one might want to build machines that require this feature. Note that the hardware presented in chapter 5 supports it; it is left to show its correctness for this case.

7.4 Out-of-Order Execution

The out-of-order machine we present uses a reorder buffer and therefore in-order termination. Machines without reorder buffer and out-of-order termination are not verified. Furthermore, multiple instruction issue machines with Tomasulo scheduler are not covered. Furthermore, commercial designs feature two or more CDBs, which is also not covered. A machine with Tomasulo scheduler, multiple instruction issue and reorder buffer is described in [Hil00]. However, the designs are not verified by machine.

7.5 Synthesizing Hardware

Subject of the master's thesis of Dirk Leinenbach is converting the PVS hardware specification into synthesizable Verilog HDL. This allows building hardware implementations of the designs using ASICs or FPGAs. This allows realistic cost and performance measuring. In particular, it allows evaluating the real hardware cost in chip area rather than gate count. This includes that one can take the hardware cost of wiring in account.

The Tomasulo scheduler uses several large bus structures. It is of interest whether these bus structures have significant impact on the hardware cost and cycle time of the design. The evaluation in [Krö99] does not cover this, since the hardware model presented in [MP95] is used. This hardware model does not take wiring in account.

Another approach of interest is automated conversion from Verilog or other hardware description langues into PVS for formal verification. This approach is used by Russinov in order to verify AMD's floating point units, for example. He converts an in-house, synthesizable HDL into ACL2 language and verifies the correctness using ACL2. The benefit of this approach is that it permits verifying existing desings in HDL.

Appendix A

Theorem Index

A.1 The PVS Proof Tree

In this chapter, we provide a mapping from the theorems in this thesis to the theorems in the PVS proof tree. This mapping is limited, however. For sake of simplicity, we sometimes present multiple lemmas of the PVS proof tree as single one in this thesis. For example, we have a single lemma that states that the initialization of the machine is correct. In the PVS proof tree, we use a separate lemma for each stage.

The following tables provide the number of the lemma or theorem, the page number, the file name of the file the lemma is to be found in, and the lemma name.

A.2 Basic Concepts

Th.	Page	File	Name
2.1	15	btree	btree_lem
2.2	16	zerotester	zerotester_correct
2.3	16	tester	equality_tester_correct
2.4	17	pp	pp_correct2
2.5	18	pp	pp_spec_equiv_lem
2.6	18	pp	pp_Xp_lem
2.7	19	pp	pp_correct1_lem
2.8	20	bvhelp	bv_adder_cin_is_add
2.9	21	cla	cla_cout_lemma
2.10	25	alu_addsub	alu_bv_unary_minus
2.11	26	alu_addsub	alu_addsub_result_correct
2.12	26	alu_addsub	alu_addsub_ovf_correct
2.13	26	alu_addsub	alu_addsub_neg_correct
2.14	26	dlxalu_imp	alu_correct

A.3 A Sequential Implementation Machine

Th.	Page	File	Name
Conv. 3.1	39	pipetheory	pipe_stall_correct
3.2	39	pipetheory	pipe_sequential_full
3.3	57	bjtaken_impl	bjtaken_imp_correct
3.4	58	nextpc_impl	nextpc_imp_correct
3.5	61	pipetheory	sched_sequential_lemma1
3.6	62	pipetheory	sched_sequential_lemma2
3.7	62	pipetheory	sched_sequential_lemma3
3.8	62	pipetheory	sched_sequential_lemma4
Inv. 3.1	65	pipetheory	sched_lemma1
Inv. 3.2	65	pipetheory	sched_lemma2
Inv. 3.3	65	pipetheory	sched_lemma3
3.9	69	pipetheory	full_bit_lemma
3.10	69	pipetheory	sched_sequential_lemma
3.11	71	pipetheory	sched_pipe_start
3.18	85	dlxs_lemmas	dlxs_correct
3.19	86	live_calculus	weakEafter_is_strongEafter
3.20	87	pipetheory	ue_is_live_IS_lem
3.21	87	pipetheory	ue_is_live_IS2
3.22	88	pipetheory	ue_is_live_seq
3.23	88	pipetheory	ue_sI_lemma
3.24	88	pipetheory	Machine_is_live

A.4 Pipelined Machines

Th.	Page	File	Name
4.1	95	pipetheory	pipe_full_def
4.3	96	pipetheory	sched_overwrite
4.4	96	pipetheory	sched_full_bits_save
4.5	97	pipetheory	sched_clear_full_bits
4.6	97	pipetheory	sched_pipe_start
4.15	131	live_calculus2	stays_until_gt
4.16	132	live_calculus2	stays_until_impl_lem
4.17	132	live_calculus2	AND_stays_until
4.18	133	live_calculus2	weakEafter_and_stays_until_ IMPLIES_weakEafter
4.19	134	live_calculus2	finite_false_and_stays_until_ IMPLIES_finite_false
4.20	134	live_calculus2	finite_true_and_stays_until_ IMPLIES_finite_true
4.21	134	live_calculus2	AND_weakEafter_and_stays_until
4.22	136	live_calculus2	AND_finite_false_and_stays_until
4.23	136	live_calculus2	OR_finite_true_and_stays_until
4.25	136	live_calculus2	never_is_finite_true_and_stays_until
4.24	136	live_calculus2	always_is_finite_false_and_stays_until
4.27	137	live_calculus	OR_finite_false
4.28	137	live_calculus	AND_finite_true
4.29	137	live_calculus2	finite_true_OR_finite_true_ and_stays_until_lem
4.30	138	pipetheory	pipe_drain_lem
4.35	144	pipetheory	pipe_stall_is_finite_true

A.5 Speculative Execution

Th.	Page	File	Name
5.1	152	pipetheory_spec	pipe_full_def
5.2	152	pipetheory_spec	sched_overwrite
5.3	153	pipetheory_spec	sched_full_bits_save
5.4	153	pipetheory_spec	sched_clear_full_bits
5.5	153	pipetheory_spec	sched_pipe_start
5.11	181	spec_theory	spec_premise1
5.12	181	spec_theory	spec_premise2
5.13	181	spec_theory	spec_premise3
5.14	182	spec_theory	spec_premise4
5.15	183	spec_theory	spec_premise5
5.16	183	spec_theory	spec_premise6
5.17	183	spec_theory	spec_premise7
5.18	183	spec_theory	rollback_stage_exists
5.19	185	spec_theory	spec_max_lemma
5.20	185	spec_theory	spec_full_lemma
5.21	186	spec_theory	stage_spec_correct_lem
5.22	186	spec_theory	spec_correct_inputs_lemma
5.23	186	spec_theory	spec_correct_spec_inputs_lemma
5.24	187	spec_theory	spec_misspec_step
5.25	188	spec_theory	spec_data_consistency_lemma2
5.26	188	spec_theory	spec_data_consistency_lemma1
5.27	189	spec_theory	spec_data_consistency1
5.28	189	spec_theory	spec_data_consistency2
5.29	190	spec_theory	spec_data_consistency3
5.30	190	spec_theory	spec_inv_hold
5.31	192	spec_theory	M_max_exists
5.32	192	spec_theory	live_M0_lem
5.33	193	spec_theory	M_is_full
5.34	193	spec_theory	g_M_not_full
5.35	193	spec_theory	M_implies_below_empty
5.36	193	spec_theory	M_lemma0
5.37	193	spec_theory	M_lemma1
5.38	195	spec_theory	rollback_correct
5.39	196	spec_theory	M_rollback
5.40	196	spec_theory	sched_rollback_lem4
5.41	197	spec_theory	spec_premise8
5.42	198	spec_theory	spec_premise9

Th.	Page	File	Name
5.43	200	spec_theory	ue_M_lemma
5.44	200	spec_theory	M_stall_is_finite_true_lem
5.45	201	spec_theory	spec_ue_is_live_IS_lem
5.46	201	spec_theory	stage_M_is_live_lem
5.47	202	spec_theory	spec_le_M_lem
5.48	202	spec_theory	M_correct_inputs_lemma
5.49	202	spec_theory	mc_premise_spec
5.50	203	spec_theory	mc_premise
5.51	204	spec_theory	live_Mc_lem1
5.52	204	spec_theory	spec_ue_sI_lemma
5.53	204	spec_theory	live_Mc_lem2
5.54	205	spec_theory	live_Mc_lem3
5.55	206	spec_theory	live_Mc_lem4
5.56	206	spec_theory	live_no_Mc_lem2
5.57	207	spec_theory	live_no_Mc_lem3
5.58	208	spec_theory	live_is_M_lem2
5.59	208	spec_theory	live_sI_M_is_laststage_lem
5.60	209	spec_theory	live_is_M_lem1
5.61	210	spec_theory	live_sI_exists_IS
5.62	210	spec_theory	live_sI_exists_lem
5.64	218	interrupts	dlx_MCA_correct
5.65	218	interrupts	dlx_JISR_correct
5.66	219	interrupts	dlx_repeat_correct
5.67	222	dlxP_f_correct	dlxP_c0_IR_correct
5.68	224	nextpc_impl	nextpci_imp_correct
5.69	224	dlxP_f_correct	dlxP_c1_nextpc_correct
5.70	224	dlxP_f_correct	dlxP_f1_PCp_correct
5.71	225	dlxP_f_correct	dlxP_f1_DPC_correct
5.72	230	dlxP_f_correct	dlxP_c3_MCA_correct
5.73	231	dlxP_f_correct	dlxP_f3_JISR_correct
5.74	232	dlxP_f_correct	dlxP_f3_repeat_correct
5.75	232	dlxP_f_correct	dlxP_c3_IR_correct
5.76	234	dlxP_f_correct	dlxP_c3_C_correct
5.77	234	dlxP_f_correct	dlxP_f4_SR_correct
5.78	236	dlxP_f_correct	dlxP_f4_ESR_correct
5.79	236	dlxP_f_correct	dlxP_f4_ECA_correct
5.80	237	dlxP_f_correct	dlxP_f4_EPC_correct
5.81	238	dlxP_f_correct	dlxP_f4_EDPC_correct
5.82	239	dlxP_f_correct	dlxP_f4_EDATA_correct

A.6 Out-of-Order Execution

Th.	Page	File	Name
6.1	262	robtheory	ROBtail_inv
6.2	262	robtheory	ROBhead_inv
6.3	262	robtheory	ROBcount_inv
6.4	263	robtheory	instr_in_rob_lemma
6.5	264	robtheory	min_ROB
6.6	264	robtheory	max_ROB
6.7	265	robtheory	sI_issue_ge_sI_writeback
6.8	265	robtheory	sI_issue_gt_sI_writeback
6.9	265	robtheory	sI_issue_ge_sI_writeback2
6.10	265	robtheory	I_tag_issue_lemma
6.11	266	robtheory	ROB_lemma
6.12	266	robtheory	issued_correct
6.13	267	robtheory	in_order_issue_aux0
6.14	267	robtheory	in_order_issue
6.15	268	robtheory	tag_inc_sum
6.16	268	robtheory	ROBtail_diff_lemma
6.17	268	robtheory	tag_inc_lemma_aux
6.18	268	robtheory	tag_inc_lemma
6.19	269	robtheory	ROB_invariant
6.20	270	robtheory	ROB_count_lemma
6.21	270	robtheory	tag_unique_lemma
6.22	271	robtheory	tag_unique_lemma2
6.23	271	robtheory	I_tag_writeback_lemma
6.24	271	robtheory	sI_writeback_lem
6.27	275	tomistate	state_unique
6.28	275	tomcorrect	tag_RS_correct
6.29	276	tomcorrect	tag_P_correct
6.30	276	tomcorrect	tag_CDB_correct
6.31	278	tomspec	not_instr_has_dest_lemma
6.32	278	tomspec	ldef_before
6.33	279	tomspec	last_has_dest_lemma
6.34	279	tomspec	last_has_not_dest_lemma_aux
6.35	279	tomspec	last_prev_has_not_dest_lemma
6.36	279	tomspec	last_prev_has_dest_lemma
6.37	279	tomspec	last_lemma_aux
6.38	280	tomspec	last_lemma

Th.	Page	File	Name
6.39	280	tomcorrect	issue_lemma1
6.40	281	tomcorrect	prod_tag_inv
6.41	282	tomcorrect	rs_tag_inv_aux1
6.42	283	tomcorrect	rs_tag_inv
6.43	283	tomcorrect	tag_unique_R
6.44	285	tomcorrect	tag_unique_RS
6.45	285	tomcorrect	RS_op_lemma
6.46	286	tomcorrect	tag_unique_RS_op
6.47	286	tomcorrect	tag_unique_P
6.48	287	tomcorrect	tag_unique_CDB
6.49	288	tomcorrect	inv_P_data_IMPL_inv_CDB_data
6.50	289	tomcorrect	inv_CDB_data_IMPL_inv_ROB_valid
6.51	290	tomcorrect	inv_ROB_valid_IMPL_inv_R_valid
6.52	291	tomcorrect	inv_RS_valid_IMPL_inv_P_valid
6.53	292	tomcorrect	CDB_tag_lemma
6.54	292	tomcorrect	inv_RS_valid_proof_read_issue
6.55	293	tomcorrect	inv_RS_valid_proof_read_snoop
6.56	295	tomcorrect	inv_RS_valid_proof
6.57	295	tomcorrect	tom_inv
6.58	295	tomcorrect	data_consistency
6.60	297	tomlive	has_state
6.61	297	tomlive	liveness_step_in_ROB
6.62	298	tomlive	stays_in_P
6.63	299	tomlive	liveness_step_in_P
6.64	300	tomlive	liveness_step_in_FU
6.65	300	tomlive	stays_in_RS
6.66	301	tomlive	liveness_step_in_RS
6.67	301	tomlive	liveness_step_not_issued
6.68	303	tomlive	liveness_step
6.69	303	tomlive	liveness

Appendix

B

DLX Instruction Set

This instruction set is taken from [MP95, MP00] with minimal modifications. The architecture was defined in [HP96]. A reference for the instruction formats and mnemonics is also [SK96].

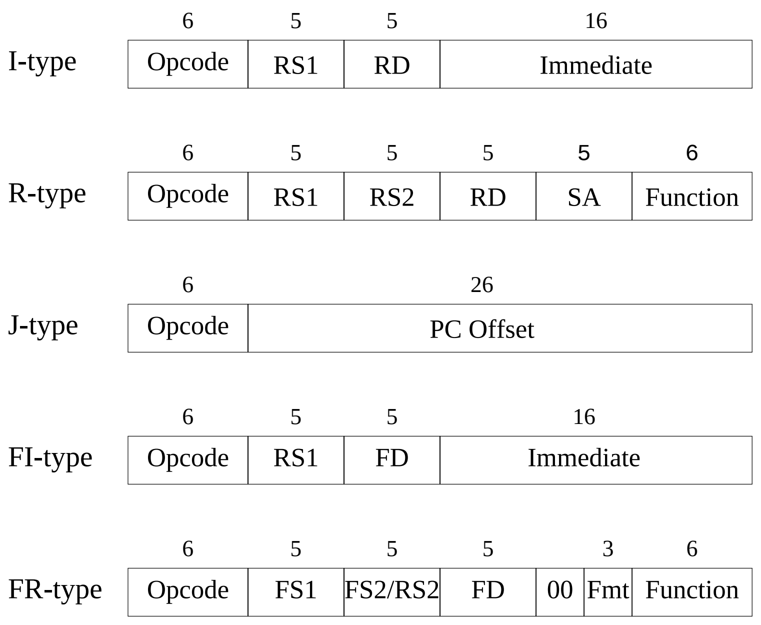


Figure B.1 Instruction formats of the DLX

IR[31 : 26]		Mnem.	d	Effect
Data Transfer, mem = M[RS1 + imm]				
100000	0x20	lb	1	RD=Sext(mem)
100001	0x21	lh	2	RD=Sext(mem)
100011	0x23	lw	4	RD=mem
100100	0x24	lbu	1	RD=0^{24}mem
100101	0x25	lhu	2	RD=0^{16}mem
101000	0x28	sb	1	mem=RD[7 : 0]
101001	0x29	sh	2	mem=RD[15 : 0]
101011	0x2b	sw	4	mem=RD
Arithmetic, Logical Operation				
001000	0x08	addi		RD=RS1 + imm
001001	0x09	addiu		RD=RS1 + imm (no overflow)
001010	0x10	subi		RD=RS1 - imm
001011	0x11	subiu		RD=RS1 - imm (no overflow)
001100	0x12	andi		RD=RS1 $\wedge$ imm
001101	0x13	ori		RD=RS1 $\vee$ imm
001110	0x14	xori		RD=RS1 $\oplus$ imm
001111	0x15	lhgi		RD=imm 0^{16}
Test Set Operation				
011000	0x18	clri		RD=(false ? 1 : 0)
011001	0x19	sgri		RD=(RS1 $>$ imm ? 1 : 0)
011010	0x1a	seqi		RD=(RS1 $=$ imm ? 1 : 0)
011011	0x1b	sgei		RD=(RS1 $\geq$ imm ? 1 : 0)
011100	0x1c	slsi		RD=(RS1 $<$ imm ? 1 : 0)
011101	0x1d	snei		RD=(RS1 $\neq$ imm ? 1 : 0)
011110	0x1e	slei		RD=(RS1 $\leq$ imm ? 1 : 0)
011111	0x1f	seti		RD=(true ? 1 : 0)
Control Operation				
000100	0x04	beqz		PC=PC+4+(RS1 $=$ 0 ? imm: 0)
000101	0x05	bnez		PC=PC+4+(RS1 $\neq$ 0 ? imm: 0)
000110	0x16	jr		PC=RS1
000111	0x17	jalr		R31=PC+4; PC = RS1

Table B.1 I-type instruction layout

IR[31 : 26]		IR[5 : 0]		Mnem.	Effect
Shift Operation					
000000	0x00	000000	0x00	slli	RD=RS1$\ll$SA
000000	0x00	000001	0x01	slai	RD=RS1$\ll$SA (arith.)
000000	0x00	000010	0x02	srli	RD=RS1$\gg$SA
000000	0x00	000011	0x03	srai	RD=RS1$\gg$SA (arith.)
000000	0x00	000100	0x04	sll	RD=RS1$\ll$RS2[4:0]
000000	0x00	000101	0x05	sla	RD=RS1$\ll$RS2[4:0] (ar.)
000000	0x00	000110	0x06	srl	RD=RS1$\gg$RS2[4:0]
000000	0x00	000111	0x07	sra	RD=RS1$\gg$RS2[4:0] (ar.)
Data Transfer					
000000	0x00	010000	0x10	movs2i	RD=SA
000000	0x00	010001	0x11	movi2s	SA=RS1
Arithmetic, Logical Operation					
000000	0x00	100000	0x20	add	RD=RS1+RS2
000000	0x00	100001	0x21	addu	RD=RS1+RS2 (no overfl.)
000000	0x00	100010	0x22	sub	RD=RS1-RS2
000000	0x00	100011	0x23	subu	RD=RS1-RS2 (no overfl.)
000000	0x00	100100	0x24	and	RD=RS1 $\wedge$ RS2
000000	0x00	100101	0x25	or	RD=RS1 $\vee$ RS2
000000	0x00	100110	0x26	xor	RD=RS1 $\oplus$ RS2
000000	0x00	100111	0x27	lhg	RD=RS2[15:0] 0^{16}
Test Set Operation					
000000	0x00	101000	0x28	clr	RD=(false ? 1 : 0)
000000	0x00	101001	0x29	sgr	RD=(RS1 $>$ RS2 ? 1 : 0)
000000	0x00	101010	0x2a	seq	RD=(RS1 $=$ RS2 ? 1 : 0)
000000	0x00	101011	0x2b	sge	RD=(RS1 $\geq$ RS2 ? 1 : 0)
000000	0x00	101100	0x2c	sls	RD=(RS1 $<$ RS2 ? 1 : 0)
000000	0x00	101101	0x2d	sne	RD=(RS1 $\neq$ RS2 ? 1 : 0)
000000	0x00	101110	0x2e	sle	RD=(RS1 $\leq$ RS2 ? 1 : 0)
000000	0x00	101111	0x2f	set	RD=(true ? 1 : 0)

Table B.2 R-type instruction layout

IR[31 : 26]		Mnem.	Effect
Control Operation			
000010	0x02	j	PC = PC + 4 + imm
000011	0x03	jal	R31 = PC + 4; PC = PC + 4 + imm
111110	0x3e	trap	trap = 1; EDATA = imm;
111111	0x3f	rfe	SR = ESR; PC' = EPC; DPC = EDPC

Table B.3 J-type instruction layout

IR[31 : 26]		Mnem.	d	Effect
Load, Store				
110001	0x31	load.s	4	FD[31 : 0] = mem
110101	0x35	load.d	8	FD[63 : 0] = mem
111001	0x39	store.s	4	m = FD[31 : 0]
111101	0x3d	store.d	8	m = FD[63 : 0]
Control Operation				
000110	0x06	fbeqz		PC=PC+4+(FCC $= 0$? imm: 0)
000111	0x07	fbnez		PC=PC+4+(FCC $\neq 0$? imm: 0)

Table B.4 FI-type instruction layout

IR[31 : 26]		IR[5 : 0]		Fmt	Mnem.	Effect
Arithmetic and Compare Operations						
010001	0x11	000000	0x00		fadd	FD = FS1 + FS2
010001	0x11	000001	0x01		fsub	FD = FS1 - FS2
010001	0x11	000010	0x02		fmul	FD = FS1 * FS2
010001	0x11	000011	0x03		fdiv	FD = FS1 / FS2
010001	0x11	000100	0x04		fneg	FD = - FS1
010001	0x11	000101	0x05		fabs	FD = abs(FS1)
010001	0x11	000110	0x06		fsqt	FD = sqrt(FS1)
010001	0x11	000111	0x07		frem	FD = rem(FS1, FS2)
010001	0x11	$11c_3c_2c_1c_0$			fc.cond	FCC=(FS1 *co* FS2)
Data Transfer						
010001	0x11	001000	0x08	000	fmov.s	FD[31:0]=FS1[31:0]
010001	0x11	001000	0x08	001	fmov.d	FD[63:0]=FS1[63:0]
010001	0x11	001001	0x09		mf2i	RS = FS1[31:0]
010001	0x11	001010	0x0a		mi2f	FD[31:0] = RS
Conversion						
010001	0x11	100000	0x20	001	cvt.s.d	FD = cvt(FS1, s, d)
010001	0x11	100000	0x20	100	cvt.s.i	FD = cvt(FS1, s, i)
010001	0x11	100001	0x21	000	cvt.d.s	FD = cvt(FS1, d, s)
010001	0x11	100001	0x21	100	cvt.d.i	FD = cvt(FS1, d, i)
010001	0x11	100100	0x24	000	cvt.i.s	FD = cvt(FS1, i, s)
010001	0x11	100100	0x24	001	cvt.i.d	FD = cvt(FS1, i, d)

Table B.5 FR-type instruction layout. Fmt=IR[8:6]

RM	Symbol	Rounding
00	RZ	toward zero
01	RNE	to next even
10	RPI	toward $+\infty$
11	RMI	toward $-\infty$

Bit	Symbol	Purpose
0	OVF	overflow
1	UNF	underflow
2	INX	inexact result
3	DBZ	divide by zero
4	INV	invalid operation

Table B.6 Coding of the rounding mode RM and the interrupt flags IEEEf

IR[31 : 26]		IR[5 : 0]		Predicate	Effect
100***	0x04			*I_load*	load instructions
***000	0x00			*I_lb*	byte signed
***001	0x00			*I_lh*	halfword signed
***011	0x00			*I_lw*	full word
***100	0x00			*I_lbu*	byte unsigned
***101	0x00			*I_lhu*	halfword unsigned
1010**	0x0a			*I_store*	store instructions
0*1***	0x00			*I_ALUi*	i-type ALU instr.
0001**	0x01			*I_branch*	conditional branch
****1*	0x00			*I_branch_fcc*	test FCC instead of RS1
*****0	0x00			*I_branch_eq*	branch if equal
01011*	0x0b			*I_jr*	jump register instr.
*****1	0x00			*I_link*	j/jr is a link instr.
00001*	0x01			*I_j*	jump instructions
111110	0x3e			*I_trap*	trap instruction
111111	0x3f			*I_rfe*	return from exception
000000	0x00	0000**	0x00	*I_shifti*	shift instr. with SA
000000	0x00	0001**	0x01	*I_shift*	shift instr.
000000	0x00	010000	0x10	*I_movs2i*	move sp. reg. to GPR
000000	0x00	010001	0x11	*I_movi2s*	move GPR to sp. reg.
000000	0x00	10****	0x02	*I_ALU*	ALU instructions

Table B.7 The monomials of the predicates used to decode the instruction word

Appendix

C

Performance of the Pipelined DLX

Short Pipeline

The first implementation uses a standard five stage pipeline as described above. All simulations were made using a Pentium-like memory system, i.e., a 16kb split, two-way level one write-back cache with 32 bytes line size and 4-1-1-1 bus bursts [Int95a, Int95b]. The cache uses LRU replacement and read/write allocation.

As a workload, we used several benchmarks of the SPEC92 benchmark suite [SPE91]. Table C.2 shows the benchmarks and performance result.

Instruction	Latency	Pipelined
addition, subtraction	5	full
conversion	3	full
multiplication	5	full
single precision division	17	five stages
double precision division	21	five stages

Table C.1 Latency of floating point instructions in cycles. Most floating point instructions can be executed fully pipelined; divisions and square root iterate except for five stages.

Benchmark	Type	FP In.	Dhaz before	Dhaz after	CPI before	CPI after	Sp.
008 espresso	int92	0.0%	2.71%	1.84%	1.3567	1.3516	0.4%
013 spice2g6	fp92	1.6%	2.92%	2.07%	1.8678	1.8496	1.0%
015 doduc	int92	8.7%	2.78%	2.11%	2.1027	2.0700	1.6%
022 li	int92	0.0%	3.11%	2.64%	1.9056	1.8841	1.1%
023 eqntott	int92	0.0%	4.09%	3.49%	1.6930	1.6769	1.0%
026 compress	int92	0.0%	2.77%	2.20%	1.6212	1.6052	1.0%
034 mdljdp2	fp92	13.0%	1.83%	1.55%	1.6782	1.6654	0.8%
039 wave5	fp92	18.5%	3.21%	2.93%	1.8741	1.8414	1.8%
047 tomcatv	fp92	31.1%	0.19%	0.16%	2.1999	2.1924	0.4%
048 ora	fp92	27.6%	4.59%	3.45%	2.8024	2.7836	0.7%
052 alvinn	int92	0.7%	3.58%	1.83%	2.3961	2.3520	1.9%
056 ear	fp92	17.3%	2.02%	1.61%	2.1931	2.1804	0.6%
072 sc	int92	0.0%	2.05%	1.55%	1.4530	1.4437	0.6%
077 mdljsp2	fp92	15.7%	1.26%	1.05%	1.6801	1.6709	0.6%
078 swm256	fp92	44.6%	1.61%	1.63%	2.1808	2.1445	1.7%
085 gcc	int92	0.0%	4.74%	3.21%	2.3306	2.3042	1.1%
089 su2cor	fp92	18.9%	2.20%	1.80%	2.9223	2.8949	0.9%
090 hydro2d	fp92	14.6%	4.18%	4.00%	2.4236	2.3931	1.3%
093 nasa7	fp92	28.5%	0.23%	0.04%	2.0788	2.0786	0.0%
094 fpppp	fp92	25.5%	2.58%	2.07%	3.1935	3.1269	2.1%
AVERAGE	-	-	-	-	2.0976	2.0755	1.6%

Table C.2 Experimental results gained using simulations on the short pipeline. In the first column, the benchmark is given, the second column gives the type of the program, the third column shows the percentage of floating point instructions. The columns four and five show the percentage of cycles with data hazard stalls before and after applying bubble removal. The last columns show the CPI values and the total speedup.

In general, the performance depends on the number of multi-cycle instructions, in particular floating point instructions. The table therefore gives the percentage of floating point instructions in the execution stream simulated. On floating point loads, one can observe a speedup up to two percent. Most of the speedup results from reduced data hazards. The table therefore gives the number of cycles spent idle in the decode stage because of data hazards before and after adding bubble removal.

This performance gain might seem neglectable. However, consider that the hardware effort for this gain is just a couple of gates.

Benchmark	CPI before	CPI after	Speedup
008 espresso	2.0853	2.0499	1.7%
013 spice2g6	2.3885	2.2735	5.1%
015 doduc	2.7613	2.5809	7.0%
022 li	2.6438	2.5121	5.2%
023 eqntott	2.2981	2.2024	4.3%
026 compress	2.0625	2.0061	2.8%
034 mdljdp2	2.0084	1.9108	5.1%
039 wave5	2.1896	1.9956	9.7%
047 tomcatv	2.6826	2.4668	8.7%
048 ora	3.2551	3.0940	5.2%
052 alvinn	2.9042	2.7288	6.4%
056 ear	2.8320	2.7181	4.2%
072 sc	2.2510	2.1875	2.9%
077 mdljsp2	2.0336	1.9429	4.7%
078 swm256	2.5566	2.4074	6.2%
085 gcc	2.8373	2.6987	5.1%
089 su2cor	3.4189	3.1873	7.3%
090 hydro2d	2.5153	2.3582	6.7%
093 nasa7	2.1982	2.1567	1.9%
094 fpppp	3.4060	3.0965	10.0%
AVERAGE	2.0976	2.0755	5.5%

Table C.3 Experimental results gained using simulations on the long pipeline

Long Pipeline

In order to achieve high clock frequencies, modern microprocessors feature very long pipelines with up to twenty stages. However, longer pipelines are also more sensitive to data hazards because of load instructions. We therefore simulated a RISC pipeline with nine stages total in order to evaluate the effect of pipeline bubble removal. We expected the benefit of pipeline bubble removal increase with pipeline complexity.

All other parameters and the workload remain the same. Table C.3 shows the results. Not surprisingly, the CPI rates raise. As expected, the percental speedup gained by the stall engine also increases. With individual benchmarks we see speedups up to ten percent and an average of five percent. However, the programs used for the nine stage pipeline are the same as for the five stage pipeline, i.e., they are compiled (and therefore optimized) for the five stage pipeline. Thus, we expect less CPI and less speedup with code compiled with optimizations for the nine stage pipeline.

Appendix

D

Liveness Verification using SMV

D.1 Introduction

The idea of Model-checking [CE81b, CES86] is to check the complete set of reachable states of a state transition system for a desired property, e.g., an invariant. However, this approach suffers from the state explosion problem since the state space grows exponentially with the number of variables.

However, one easily encapsulates the stall engine as a module with well-defined interface. In this module, the full bits are the only registers. All other registers of the machine are not required for the stall engine. The number of full bits is exactly the number of stages. Thus, there are five one-bit registers in the stall engine for the DLX design discussed in the previous chapters. Thus, there are 2^5 possible states. It therefore seems feasible to verify properties of the stall engine for fixed size pipelines.

In the following, we will apply a well-known symbolic model-checking system called *SMV* by Kenneth McMillan [McM93]. Symbolic model-checking systems represent the state space as boolean formula. All operations (property checking) are done on this formula, which usually is much faster than just enumerating the reachable states.

An introduction of the hardware specification language used by SMV is beyond the scope of this thesis. Thus, the specification of the stall engine hardware in SMV language and a small introduction can be found in appendix D. The specification of the liveness criterion in SMV is done using temporal operators that are similar to those used in CTL (Computation Tree Logic) [CE81b][1].

Let p be a time predicate. For the specification of the liveness property, the following two temporal operators are used:

- The operator $F\,p$ holds iff there is a cycle T in the future such that the predicate p holds. The definition of the operator $\exists^{\geq T}$ in chapter 3 is identical to this definition.
- The operator Gp holds iff the predicate p holds for all future cycles.

Applying a CTL operator on a time predicate results in a time predicate. Thus, the operators can be combined: For example, in SMV, $GF\,p$ denotes a predicate p that is finite true according to definition 3.4.

Thus, the assumption that the external stall signals are finite true is denoted as follows in SMV language:

$$G\,F\,\neg ext_k \tag{D.1}$$

We furthermore assume that if all stages below stage k are empty (i.e., not full), the data hazard hazard signal of stage k is off ($below\_full_k$ holds iff at least one stage below stage k is full):

$$G\,(\neg below\_full_k \Longrightarrow \neg dhaz_k) \tag{D.2}$$

Using these assumptions, SMV verifies the following property of the stall engine logic for a fixed number of stages:

$$G\,F\,ue_k \tag{D.3}$$

However, the verification time grows exponentially in the number of stages. Table D.1 shows experimental results on an AMD machine with 350 MHZ. The liveness of the stall engine of the DLX pipeline with five stages is verified within a second; however, the run time becomes critical with 10 stages and beyond. Commercial designs feature up to 30 stages, which would result in an estimated run time of about 10,000 years. However, the author's machine ran out of memory while model-checking the stall engine with eleven stages or beyond.

[1] CTL is a subset of a more general temporal logic described in [CE81a], using the syntax of [BMP81].

Stages	BDD nodes	Time [s]
1	14	0.04
2	850	0.07
3	4387	0.10
4	10027	0.34
5	25764	1.15
6	101598	6.12
7	265952	14.68
8	643058	46.97
9	1294767	242.43
10	5008999	833.76

Table D.1 Experimental results for verifying the liveness criterion of the stall engine using SMV on an AMD machine with 350 MHZ

D.2 Using Induction

In this section, we try to speed up the verification manually. The first step is to split the proof goal into *n* subgoals, one subgoal for the correctness criterion for each stage. This makes the verification both faster and reduces the memory consumption.

SMV supports a simple form of induction. This can be applied in analogy to the proof presented in the previous section. By assuming the liveness property for stage $k+1$, one simplifies the proof of the liveness property for stage k. This makes the verification both faster and reduces the memory consumption dramatically. The price for this is dropping the full automatization of the proof.

Table D.2 and figure D.1 show the run time for verifying the liveness criterion for a stall engine with up to 18 stages using no induction and using one stage induction.

Model-checking was initiated by Clarke and Emerson in [CE81b] and [CES86]. Various authors improved the idea in order to handle larger state spaces. In order to handle the state explosion problem, BDDs (binary decision diagrams) were applied for model-checking [McM93]. A big contribution to model-checking is from Bryant by his research on BDD techniques [Bry86]. Recently, McMillan applied classical theorem proving techniques for model-checking, e.g., in [McM98, McM99b].

Stages	no induction		1 stage induction	
	BDD nodes	Time [s]	BDD nodes	Time [s]
1	14	0.04	14	0.04
2	237	0.07	237	0.08
3	701	0.16	701	0.16
4	1147	0.25	1147	0.24
5	2154	0.65	2154	0.38
6	10035	1.62	3842	0.63
7	14274	4.91	6747	1.32
8	27142	12.98	10046	2.78
9	36096	26.83	12413	4.25
10	51856	46.26	29375	7.36
11	70591	133.67	49935	15.93
12	98800	212.42	98857	22.70
13	139882	581.36	139945	102.35
14	213847	3096.31	213916	184.58
15	308379	10163.90	308454	386.58
16	444037	37322.40	444118	763.36
17	-	-	662026	1729.37
18	-	-	1044414	5416.22

Table D.2 Experimental results for verifying the liveness criterion of the stall engine using SMV. The columns two and three contain the BDD node count and the runtime in seconds for verifying the liveness criterion for all stages separately using no induction. The columns four and five contain the BDD node count and time for verifying using induction.

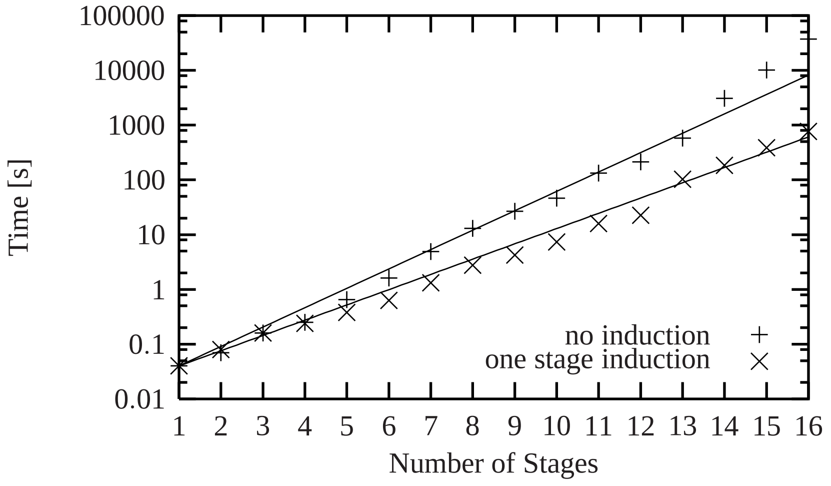


Figure D.1 Visualization of the experimental results for verifying the stall engine using SMV in table D.2

Appendix D

Liveness Verification using SMV

Bibliography

[AS99] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro Special Issue on Modeling and Validation of Microprocessors*, 19(3):36–46, May/June 1999.

[BBCZ98] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *Lecture Notes in Computer Science*, pages 369–386. Springer-Verlag, 1998.

[BCRZ99] A. Biere, E. Clarke, E. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag, 1999.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessors control. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1994.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of ACM/IEEE Design Automation Conference (DAC'98)*, pages 522–527. ACM Press, 1998.

[Ber01] Christoph Berg. Verification of an IEEE floating point adder (draft). Master’s thesis, Universität des Saarlandes, FB. Informatik, 2001.

[Bey01] Sven Beyer. *Verification of a microprocessor’s memory interface (Draft)*. PhD thesis, University of Saarland, Computer Science Department, 2001.

[BJK01] Christoph Berg, Christian Jacobi, and Daniel Kroening. Formal verification of a basic circuits library. In *Proc. 19th IASTED International Conference on Applied Informatics, Innsbruck (AI’2001)*, pages 252–255. ACTA Press, 2001.

[BM96] E. Boerger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J.P Bowen, M.B. Hinchey, and D. Till, editors, *ZUM’97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 151–187. Springer-Verlag, 1996.

[BMP81] M. Ben-Ari, Z. Manna, and A. Pneuli. The temporal logic of branching time. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages (POPL ’81)*, pages 164–176. ACM Press, Jan 1981.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BS89] M. Bickford and M. Srivas. Verification of a pipelined microprocessor using CLIO. In *Proceedings of Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[Bur91] Jerry R. Burch. Using BDDs to verify multipliers. In *Proceedings of the 28th ACM/IEEE Design Automation Conference (DAC’91)*, pages 408–412, New York, 1991. ACM Press.

[CE81a] Edmund Clarke and Allen Emerson. Characterizing properties of parallel programs as fixpoints. In *7th International Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[CE81b] Edmund Clarke and Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *In Logic of Programs: Workshop, Yorktown Heights*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[CES86] Edmund Clarke, Allen Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGM86] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 41–66. North-Holland, 1986.

[CHYP94] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale N. Patt. Branch classification: a new mechanism for improving branch predictor performance". In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 22–31, 1994.

[Coe95] Tim Coe. Inside the Pentium FDIV bug. *Dr. Dobb's Journal of Software Tools*, 20(4), Apr 1995.

[Coh87] Avra J. Cohn. A proof of correctness of the VIPER microprocessor: The first level. In Graham Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1987.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In *2nd International Conference on Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1994.

[CS95] Robert P. Colwell and Randy L. Steck. A 0.6um bicmos processor employing dynamic execution. International Solid State Circuits Conference (ISSCC), 1995.

[Cyr93] David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, 1993.

[Del98] Peter Dell. Die Auswirkung von Mechanismen zur out-of-order Ausführung auf den Cyclecount von RISC-Architekturen. Master's thesis, Universität des Saarlandes, FB. Informatik, 1998.

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In H.F. Li and D.K. Probst, editors, *Advances in Hardware Design and Verification: IFIP WG 10.5 Internatinal Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 23–47. Chapmann & Hall., 1997.

[FFK88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 2–5. IEEE Computer Society Press, 1988.

[Fly95] Michael Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones & Bartlett, 1995.

[Gau95] Thilo Gaul. An abstract state machine specification of the DEC-Alpha processor family. Technical Report [Verifix/UKA/4], University of Karlsruhe, 1995.

[Ger98] N. Gerteis. The performance impact of precise interrupt handling on a RISC processor (German). Master's thesis, University of Saarland, Computer Science Department, Germany, 1998.

[Hew94] Hewlett Packard. *PA-RISC 1.1 Architecture Reference Manual*, 1994.

[HGS99] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods: IFIP WG 10.5 Advanced Research Working Conference, CHARME '99*, pages 8–316. Springer-Verlag, 1999.

[HGS00] Ravi Hosabettu, Ganesh Gopalakrishnan, and Mandayam Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In Allen Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Hil00] Mark Hillebrand. Design and evaluation of a superscalar RISC processor. Master's thesis, Universität des Saarlandes, FB. Informatik, Saarbrücken, 2000.

[Hos00] Ravi Hosabettu. *Systematic Verification of Pipelined Microprocessors*. PhD thesis, University of Utah, Department of Computer Science, 2000.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[HQR98] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proceedings of the 10th International Conference on Computer-aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer-Verlag, 1998.

[Hun94] Warren A. Hunt. *FM8501, a verified microprocessor*, volume 795 of *Lecture Notes in Artificial Intelligence and Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[IEE85] Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[Int95a] Intel Corporation. *2430FX PCIset Datasheet 82437FX System Controller (TSC) and 82438FX Data Path Unit (TDP)*, 1995.

[Int95b] Intel Corporation. *Pentium Processor Family Developer's Manual, Vol. 1-3*, 1995.

[Jac01] Christian Jacobi. *Formal Verification of a fully IEEE compliant Floating Point Unit (Draft)*. PhD thesis, University of Saarland, Computer Science Department, 2001.

[JNFSV97] Jawahar Jain, Amit Narayan, M. Fujita, and A. Sangiovanni-Vincentelli. A survey of techniques for formal verification of combinational circuits. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD '97)*, pages 445–454. IEEE Society Press, 1997.

[Joy88a] Jeffrey J. Joyce. Formal specification and verification of microprocessor systems. *Microprocessing & Microprogramming*, 24(1-5):371–8, 1988.

[Joy88b] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–158. Kluwer Academic Publishers, 1988.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KM96] Matt Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *Proc. of the Eleventh Annual Conference on Computer Assurance*, pages 23–34. IEEE Computer Society Press, 1996.

[KMP99] Daniel Kroening, Silvia M. Mueller, and Wolfgang Paul. A rigorous correctness proof of the Tomasulo scheduling algorithm with precise interrupts. In *Proc. of the SCI'99/ISAS'99 International Conference*, 1999.

[KP95] Jörg Keller and Wolfgang J. Paul. *Hardware Design — Formaler Entwurf Digitaler Schaltungen*. TEUBNER, Stuttgart, Leipzig, 1995.

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

[KPM00] Daniel Kroening, Wolfgang J. Paul, and Silvia M. Mueller. Proving the correctness of pipelined micro-architectures. In Klaus Waldschmidt and Christoph Grimm, editors, *Proc. of the ITG/GI/GMM-Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 89–98. VDE Verlag, 2000.

[Krö99] Daniel Kröning. Design and evaluation of a RISC processor with a Tomasulo scheduler. Master's thesis, University of Saarland, Computer Science Department, Germany, 1999.

[Lei99] Holger Leister. *Quantitative Analysis of Precise Interrupt Mechanism for Processors with Out-Of-Order Execution*. PhD thesis, University of Saarland, Computer Science Department, 1999.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[LO96] Jeremy Levitt and Kunle Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *33rd Design Automation Conference (DAC'96)*, pages 558–563, New York, 1996. Association for Computing Machinery.

[LS84] Jonny K. F. Lee and Alan J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, January 1984.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[McM98] Kenneth L. McMillan. Verification of an implementation of Tomasulo's algorithm by composition model checking. In *Proc. 10th International Conference on Computer Aided Verification*, pages 110–121, 1998.

[McM99a] Kenneth L. McMillan. Circular compositional reasoning about liveness. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods: IFIP WG 10.5 Advanced Research Working Conference, CHARME '99*, pages 342–345. Springer-Verlag, 1999.

[McM99b] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods: IFIP WG 10.5 Internatinal Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 1703 of *Lecture Notes in Computer Science*, pages 219–233. Springer-Verlag, 1999.

[Min95] Manfred Minimair. Design, analysis and implementation of an adder by Ladner and Fisher. Technical Report 95-15, RISC-Linz, Johannes Kepler University, Linz, Austria, 1995.

[MLD$^+$99] Silvia M. Mueller, Holger Leister, Peter Dell, Nikolaus Gerteis, and Daniel Kroening. The impact of hardware scheduling mechanisms on the performance and cost of processor designs. In *Proc. of the 15th GI/ITG Conference 'Architektur von Rechensystemen' ARCS'99*, pages 65–73. VDE Verlag, 1999.

[Mot97] PowerPC 750 RISC Microprocessor Technical Summary, 1997.

[MP95] Silvia M. Müller and Wolfgang J. Paul. *The Complexity of Simple Computer Architectures*. Lecture Notes in Computer Science 995. Springer-Verlag, 1995.

[MP96] S. Müller and W. Paul. Making the original scoreboard mechanism deadlock free. In *Proc. 4th Israeli Symposium on Theory of Computing and Systems (ISTCS)*, pages 92–99. IEEE Computer Society Press, 1996.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer-Verlag, 2000.

[MPK00] Silvia M. Müller, Wolfgang Paul, and Daniel Kröning. Proving the correctness of processors with delayed branch using delayed PC. In I. Althoefer, N. Cai, G. Dueck, L. Khachatrian, M. Pinsker, A. Sarkozy, I. Wegener, and Zhang Z., editors, *Proc. Symposium on Numbers, Information and Complexity, Bielefeld*, pages 579–588. Kluwer, 2000.

[Mül97] Silvia M. Müller. Complexity and correctness of computer architectures. In *Proc. 4th Workshop on Parallel Systems and Algorithms (PASA'96)*, pages 125–146. World Scientific Publishing, 1997.

[PA96] J. Pihl and E. J. Aas. A multiplier and squarer generator for high performance dsp applications. In *Proceedings of the 39th Midwest Symposium on Circuits and Systems*, Iowa, 1996.

[PH94] David A. Patterson and John L. Hennessy. *The Hardware/Software Interface*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 1994.

[Pra95] Vaughan R. Pratt. Anatomy of the Pentium bug. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer-Verlag, 1995.

[PS94] Dionisios N. Pnevmatikatos and Gurinadar S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proc. of the 21th Annual Symposium on Computer Architecture*, pages 120–129, 1994.

[SGGH94] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. *Formal Methods in System Design*, 4(1):181–210, 1994.

[SH99] Jun Sawada and Warren A. Hunt. Results of the verification of a complex pipelined machine model. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods: IFIP WG 10.5 Advanced Research Working Conference, CHARME '99*, pages 313–316. Springer-Verlag, 1999.

[SK96] Philip M. Sailer and David R. Kaeli. *The DLX Instruction Set Architecture Handbook*. Morgan Kaufmann, San Francisco, 1996.

[Smi81] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–148, 1981.

[SP88] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[SPA92] SPARC International Inc. *The SPARC Architecture Manual*. Prentice Hall, 1992.

[SPE91] SPEC Newsletter, Vol. 3, No. 4, 1991.

[Tom67] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[VB00] Miroslav N. Velev and Randal E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *Proceedings of ACM/IEEE Design Automation Conference (DAC'00)*, pages 112–117. ACM Press, 2000.

[Win95] Phillip J. Windley. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers*, 44(1):54–72, 1995.

[Yeu84] Bik Chung Yeung. *8086/8088 Assembly Language Programming*. Wiley & Sons, 1984.

[YP92] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. of 19th Int. Sym. on Computer Architecture*, pages 124–134, 1992.