UNIVERSITY OF SOUTHAMPTON

Guarded Atomic Actions and Refinement in a System-on-Chip Development Flow: Bridging the Specification Gap with Event-B

by

John Larry Colley

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

November 2010

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by John Larry Colley

Modern System-on-chip (SoC) hardware design puts considerable pressure on existing design and verification flows, languages and tools. The Register Transfer Level (RTL) description, which forms the input for synchronous, logic synthesis-driven design is at too low a level of abstraction for efficient architectural exploration and re-use. The existing methods for taking a high-level paper specification and refining this specification to an implementation that meets its performance criteria is largely manual and error-prone and as RTL descriptions get larger, a systematic design method is necessary to address explicitly the timing issues that arise when applying logic synthesis to such large blocks.

Guarded Atomic Actions have been shown to offer a convenient notation for describing microarchitectures that is amenable to formal reasoning and high-level synthesis. Event-B is a language and method that supports the development of specifications with automatic proof and refinement, based on guarded atomic actions. Latency-insensitive design ensures that a design composed of functionally correct components will be independent of communication latency. A method has been developed which uses Event-B for latency-insensitive SoC component and sub-system design which can be combined with high-level, component synthesis to enable architectural exploration and re-use at the specification level and to close the specification gap in the SoC hardware flow.

Contents

List of Figures

List of Tables

Academic Thesis: Declaration Of Authorship

I, John Larry Colley

declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

Guarded Atomic Actions and Refinement in a System-on-Chip Development Flow: Bridging the Specification Gap with Event-B

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Either none of this work has been published before submission, or parts of this work have been published as:

   On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification

   Colley, J and Butler, M

   Refinement Based Methods for the Construction of Dependable Systems
   Dagstuhl, Germany
   September 2009

Signed: ……

Date: ……

Acknowledgements

I would like to thank my supervisors Michael Butler and João Marques-Silva for their guidance and Edd Turner, Lis Ball, Renato Silva and Ross Horne for their help during my research.

Chapter 1

Introduction

The complexity of modern System-on-Chip (SoC) hardware stretches the existing design and verification flows, languages and tools to the limit of their capabilities [Asanovic et al., 2006], [Sylvester and Keutzer, 2001]. Verification takes a larger and larger proportion of the overall effort and it is often very late in the design process that timing issues, resulting from the very small feature sizes of modern silicon processes, are encountered and can only be corrected by substantial re-design [Carloni and Sangiovanni-Vincentelli, 2002]. The commensurate reduction in the predictability of the verification closure process means that there is a clear need to enhance existing design flows to be better able to manage this increased complexity, without losing the well-established benefits that have driven successful synchronous design.

First, the Register Transfer Level (RTL) description, which forms the input for synchronous, logic synthesis-driven design [Devadas et al., 1994] is at too low a level of abstraction for efficient architectural exploration and re-use [Hoe, 2000] and second, the existing methods for taking a high-level paper specification and refining this specification to an implementation that meets its performance criteria is largely manual and error-prone [Arvind et al., 2004]. Third, as RTL descriptions get larger, a systematic design method is necessary to address explicitly the timing issues that arise when applying logic synthesis to such large blocks [Asanovic, 2007], [Rose et al., 2005].

Term Rewriting Systems (TRS) [Hoe and Arvind, 1999] have been shown to offer a convenient notation, as a set of *guarded atomic actions*, for describing microarchitectures that is amenable to formal reasoning and is used by the industrial tool Bluespec [Nikhil, 2004] for high-level synthesis. Event-B is a language and method [Abrial et al., 2006] that supports the development of specifications with automatic proof and refinement, also based on guarded atomic actions. Latency-insensitive design [Carloni and Sangiovanni-Vincentelli, 2002] is a method for assembling and managing the communication between components in complex synchronous hardware systems which ensures that a design composed of functionally correct components will be independent of communication latency.

In this work a method has been developed which uses Event-B for latency-insensitive SoC component and sub-system design which can be combined with high-level component synthesis to enable architectural exploration and re-use at the specification level and to close the specification gap in the SoC hardware flow. It will be shown that an abstract specification can be refined systematically to a level where it is possible to represent and reason about the key elements that the hardware designer employs in developing an architecture to meet the required constraints on performance and power. These key elements include pipelining strategies, the size of components, and the communication mechanisms employed between components. The goals of this work are threefold.

First, to show that alternative pipeline architectures within components can be developed from a high-level specification, compared and verified systematically. The challenge here is that all possible combinations of pipeline activity must be identified and explored exhaustively.

Second, that a high-level model for latency-insensitive communication can be developed that can be used at the specification level to verify formally that communicating components obey the latency-insensitive protocol and will not cause deadlock.

Third, that the specification can be refined to a level of abstraction that matches that required for input to high-level or RTL synthesis.

Chapter 2 provides background information, covering the technical and financial motivation for the adoption of verification tools and the need for greater confidence in the quality of a design before sign-off and manufacture. It then looks at current verification technologies and solutions, their strengths and weaknesses, and the recent emergence of new technologies to address these weaknesses. It also covers the new and evolving standard for Transaction-Level Modelling (TLM) in SystemC and how it augments the existing hardware design flow. It goes on to provide a background to methods for modelling concurrency, including the use of guarded atomic actions for modelling and high-level synthesis, and an introduction to the Event-B language and method.

The contribution of Chapter 3 is to show how the existing SoC synchronous hardware design flow can be enhanced by the seamless introduction of a formal specification and verification method that addresses the gap that exists in this flow between the specification and the RTL description used for logic synthesis. It looks at the reasons that behavioural synthesis has not in general helped to close this gap, and how guarded atomic actions can be used to provide efficient high-level synthesis and architectural exploration. It then provides an overview of how the Event-B formal method can be combined with RTL and high-level guarded atomic action synthesis to close the specification gap. This method is then elaborated in detail in the subsequent chapters.

The contribution described in Chapter 4 is an Event-B based method that can be used for developing SoC component specifications that takes into account the restrictions that hardware process technology places on component size. It then shows how the method can be applied to pipelined architectures and explores the implications that such architectures have for how simultaneous pipeline events are managed.

The contribution of Chapter 5 is to show how a SoC microprocessor pipelined implementation can be derived formally from its abstract specification using the Event-B based method described in Chapter 4.

Chapter 6 addresses the issues of latency introduced by memory sub-systems. The contribution of this chapter is to show how latency and out-of-order completion can be modelled formally in an IP Lookup circular pipeline.

Chapter 7 covers the emerging latency-insensitive protocols which can de-couple subsystem design from the complex timing interaction that can occur between components. The contribution is an extension of the method described in Chapert 4 to support latency-insensitive design with formal proof.

Chapter 2

Background

2.1 Board-level Design and Verification

Before the widespread adoption of complementary metal oxide semiconductor (CMOS) technology [Horowitz and Hill, 1989], the complexity of application-specific hardware designs was limited by transistor size. Transistor Transistor Logic (TTL) [Horowitz and Hill, 1989] packaged 4 NAND gates in a 2cm x 1cm chip so that approximately 500 gates could be placed on a board. Using multiple boards it was possible to build designs with a few thousand gates.

The advantage, however, of using discrete components was that it was very easy to build a working prototype of the design which could be run at full speed on a tester, and in the event of errors in the design it was possible to insert tester probes anywhere on the board to help locate the errors [Williams, 1986]. Errors could be fixed by replacing components or inserting wire patches. Test patterns were developed by hand, and once the prototype passed the tests, track layout could be performed and the boards manufactured [Russell, 1985]. Even if errors were discovered after manufacture, the boards could be patched on-site.

The low cost of failure meant that there was low demand for commercial tools to aid in design verification, and the majority of tools used, for instance to optimise layout, were developed in-house by the electronics companies themselves.

2.2 Chip-level Design and Verification

As CMOS technology became more accessible, companies such as Texas Instruments and LSI Logic offered methodology and manufacturing services for application-specific

(ASIC) CMOS design. Coupled with the shrinking transistor sizes enabled by the increasingly sophisticated manufacturing processes developed by the semi-conductor companies, it was now possible for fab-less design companies to develop designs of considerable complexity with very low unit cost. By 2006 it was possible to produce designs with more than 5 million gates on a single chip.

It is, however, far more difficult to verify a design on a single chip than on a board. It is no longer possible to produce a working prototype and, prior to manufacture, a photographic mask of the chip [Ragan et al., 2002] , which can cost as much as $2 million, must be produced. The cost of failure has become extremely high. It is these economic considerations that have driven the development of an Electronic Design Automation (EDA) industry that was, in 2006, valued at $5.3 billion [Huygen, 2007].

2.3 EDA Design and Verification Languages

2.3.1 Modelling Languages

Early modelling languages only supported logic gate descriptions and were seldom entered by hand. Schematic capture tools [Russell, 1985] allowed users to enter the design graphically, and the modelling language was used for data interchange between tools [Stanford and Mancuso, 1990], but this technology severely restricted the size of design that could be handled.

In the mid 1980's the modelling languages Verilog [Thomas and Moorby, 2002] and VHDL [Perry, 1994] raised the level of modelling abstraction to the Register Transfer Level (RTL) and enabled a switch from schematic capture to language-driven design.

An RTL description represents the design as a set of communicating processes. Each process can represent a block of combinational logic, a finite state machine or a combination of both. Although the data types are still low level, the significant improvement in control logic abstraction means that much larger designs can be contemplated. Coupled with the availability of Logic Synthesis tools [Devadas et al., 1994], which generate automatically a gate level representation from the RTL, designers had a mechanism for exploiting the increased complexity made available by the new CMOS technology.

At the same time, VHDL and Verilog provided language constructs to support behavioural modelling. VHDL took these constructs, such as procedure calls, from ADA, and Verilog from C. It was hoped that behavioural modelling would supersede RTL modelling, but this has not happened, for reasons that will be discussed later.

2.3.2 Test Languages

Because of the very high masking costs, considerable effort has been placed in the EDA industry into designing languages that can be used to ensure that the model of the design has been thoroughly tested before manufacture and, importantly, to provide a measure of how well the tests cover the design. Specman [Kuhn et al., 2001], VERA [Haque and Michelson, 2001] and SystemVerilog [Sutherland et al., 2004] provide object-oriented languages that allow the environment of the design to be modelled at a very high level. Constrained-Random generation of tests means that the number of tests that can be applied to the design can be significantly higher than could be achieved using hand-written directed tests, and there is more chance that unanticipated corner-cases are covered.

The test languages provide a measure of functional coverage which, coupled with traditional code coverage measures, can be used to increase confidence in the signed-off design.

2.3.3 Property Languages

While test languages provide a functional view of the properties of the design and its environment, property languages provide a formal view. Based on Linear-time Temporal Logic (LTL) [Huth and Ryan, 2004] the property languages PSL (Property Specification Language) [Foster et al., 2005] and SVA (SystemVerilog Assertions) [Sutherland et al., 2004] augment the test languages by providing a means to specify the temporal behaviour of the design (properties/assertions) and its environment (assumptions). The other major advantage of properties is that they may be embedded in the design itself, so that errors can be flagged locally.

2.4 EDA Tools

2.4.1 Simulation-based Verification

RTL simulation remains the dominant technology for the verification of application-specific hardware designs. Figure 2.1 shows how the model of the design, represented by the function `f` of its inputs, outputs and internal state, is placed in a test harness that represents the specification of the design. The test harness may consist of directed-tests, pseudo-random tests and assertions. The design is simulated using the input values generated by each test, and the outputs are checked to ensure correct functionality. At the same time, the code and functional coverage of each test is measured, and these measurements are accumulated to provide an overall value of design coverage. When the accepted coverage threshold is reached, the design is signed-off.

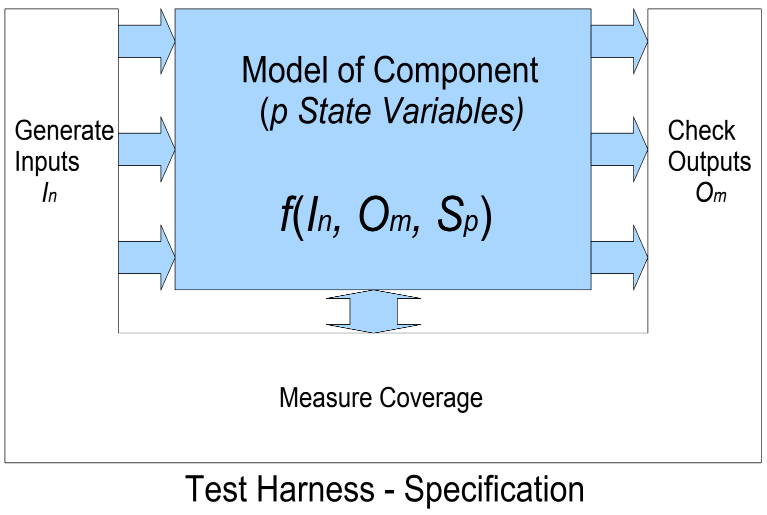


FIGURE 2.1: Simulation Based Verification

2.4.2 Disadvantages of Simulation

Although the use of test languages and functional coverage has provided increased confidence in design sign-off, this has come at a price. The number of tests that can accumulate with a design has increased from hundreds to tens of thousands. Pseudo-random tests can never be as well-targeted as directed tests, and this results in much test duplication. For a 2 million gate design with 20 thousand tests, this can result in simulation times of up to 2 weeks, even if the tests are distributed on simulation farms with hundreds of processors (Infineon, Bristol: Case Studies[1]). The effort required to verify design changes increases greatly as the design progresses.

The real disadvantage of simulation, however, is that the executable specification of the design, the set of tests, is not in any way formal, and therefore any measure of coverage is approximate. For designs that have safety considerations, particularly those in the expanding automotive sector, simulation on its own is not sufficient to give the required confidence for design sign-off.

2.4.3 Formal Verification

Although there was interest in some theorem proving solutions in the 1990's (Larch/VHDL [Baraona and Alexander, 1994], [Alexander and Baraona, 1997] and Lambda [Hadjinicolaou et al., 1994]), their limited capacity and need for expert manual intervention meant that the EDA industry focused on the promise of automatic formal verification

[1]Private Communication

that model checking technology [Clarke and Emerson, 1981], [J.R. Burch et al., 1990] provided.

EDA tool developers and users have put considerable effort into defining the IEEE standard Property Specification Language (PSL), which is based on Linear-time Temporal Logic [Vardi and Wolper, 1984] can be used for both formal and simulation-based verification, and has well-defined formal semantics [Gordon et al., 2003]. The essence of PSL has also been incorporated into SystemVerilog and, as a result, users have access to a mechanism for formal hardware specification in all the major industry tool-sets.

In parallel with the deployment of PSL, considerable advances have been made in model checking technology, particularly with the adoption of SAT-based techniques [Clarke et al., 2001], which means that very large design components can be verified automatically. Model checkers take as input the same RTL model description as used in simulation, and Figure 2.2 shows how the test-bench used for simulation is replaced with a formal description of the design's environment (the assumptions), and a set of properties (the assertions) that specify the behaviour of the design.

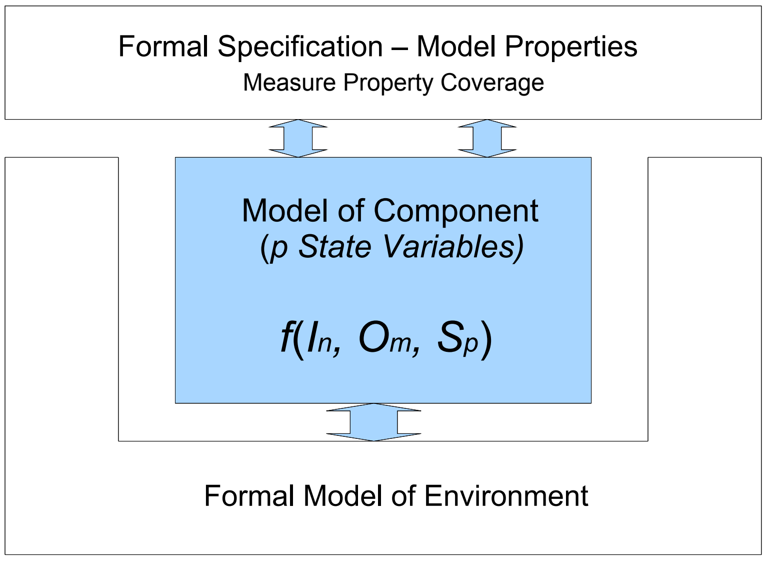

Formal Specification – Model Properties
Measure Property Coverage
Model of Component
(p State Variables)
f(In, Om, Sp)
Formal Model of Environment

FIGURE 2.2: Model Checking

2.4.4 Disadvantages of Current Model Checking Solutions

Apart from the fact that model checking cannot be used on very large designs, the effort required to write the PSL properties for a design component is large. PSL was designed to be used with the low-level data types used in RTL descriptions, and the properties required to describe RTL behaviour can be long and complex. Although suitable for parallel interfaces such as ARM AMBA where the state of the bus is defined explicitly by a set of low-level signals [Cohen et al., 2004], for serial interfaces such as PCI Express (TransEDA PCI Express PSL Property Library[2]) where the bus state has to be decoded from the bit stream, the properties become long and unwieldy. In practice, HDL monitors need to be written to present a higher transaction level stream to the PSL checker, but it would be easier if it were possible to raise the level of abstraction and define transaction-level properties in the property language itself.

Property coverage metrics [Chockler et al., 2001] are also immature, so there is no way of measuring that a set of properties adequately covers the functionality of the models. PSL properties are often written as implications and there is no way of telling that an implication has only been vacuously satisfied and that the left hand side of the implication is always false. This situation can occur in simulation-based verification when the test bench is inadequate and the signals comprising the left hand side have not been fully exercised. It can also occur in model checking when the environment is over-constrained.

2.5 Higher-level Modelling Abstractions

2.5.1 Behavioural Modelling

As described in Section 2.3 on Page 5, modern hardware description languages have inherited the capabilities of the procedural programming languages, ADA and C. Writing procedural descriptions of hardware, excluding all timing and concurrency issues, is termed behavioural modelling. The main advantage of behavioural models is that they can be simulated at speeds orders-of-magnitude faster than their RTL counterparts. The disadvantage is that efficient hardware is not procedural, but highly concurrent, and the translation of a behavioural description to an RTL description that will implement a given hardware architecture is complex and application specific. Much research and development has been undertaken over the last 20 years to automate this translation (behavioural synthesis), but no viable general solution has been found, and it remains largely a manual operation. Without this automatic step in place, any attempts to use behavioural descriptions for formal verification would be of little value.

Currently, behavioural models are usually written in C to provide a behavioural reference specification for the RTL model that can be used in system simulation to improve simulation speed. The C test-bench is re-used to verify the RTL, or the C and RTL models are co-simulated in the design.

[2]Private Communication

2.5.2 Transaction-Level Modelling(TLM)

Another way of increasing simulation speed is to reduce the number of discrete data communications between processes, and the frequency of those data communications [Donlin, 2004]. Where the interface comprises many bit and bit vector signals, these are bundled into a single packet of data for transfer. In addition, if each communication consists of a sequence of data transfers over time, the data packets can be collected and passed as a single transaction. This abstraction of both simulation data values and events in inter-process communication leads to simpler and faster simulation models.

Although primarily developed to speed up simulation, the event and data abstraction principles on which TLM is founded could provide a major contribution to raising the abstraction level for formal verification.

2.6 SystemC Transaction Level Modelling

The Open SystemC Initiative (OSCI), which promotes the SystemC hardware-oriented language based on C++ class libraries, has recognised the importance of TLM [Ghenassia, 2006] and produced a draft standard [OSCI-TLMSubgroup, 2007] for TLM modelling in SystemC. The standard focuses on the communication between hardware processes using standardised interfaces [Rose et al., 2005], [Swan, 2006], [Bombieri et al., 2006a].

2.6.1 TLM level 3

The highest level, TLM level 3, is a behavioural modelling style but with a well-defined communication mechanism. It is designed for high-performance simulation while supporting a design methodology which establishes an inter-process communication mechanism that can be refined at lower levels. TLM level 3 is `untimed`: there is no notion of time or a clock to control execution of the design and the ordering of events is determined by the order in which the communications (transactions) occur between processes. After an initial evaluation, each process waits until there is a change on one or more of its inputs. The process code is then executed and then suspends again, waiting for further input changes. Each execution of the process code is assumed to take zero time. A Master process is written using sequential C++ code and the Master and Slave communicate via `read` and `write` procedure calls implemented by the slave. High-level types are used to represent the data. The master issues a command to the slave using the write procedure call, the operation is performed by the slave and the return result made available in zero time. The Master may therefore call the read procedure call immediately to get the result. When two independent processes communicate in SystemC, the communication and synchronisation is handled by the scheduler of the simulator kernel [Mueller et al., 2001]. The request is placed in a priority queue [Brown, 1988], which incurs a context switch in the SystemC simulator. Since Master/Slave communications at TLM Level 3 do not need to use simulator kernel event scheduling, level 3 models simulate quickly, but retain the disadvantages of behavioural modelling described in 2.5.1 on page 9 above.

2.6.2 TLM level 2

2.6.2.1 Modelling TLM level 2 Processes

At TLM level 2, each process is represented by an Extended Finite State Machine (EFSM) [Gajski et al., 1997], [Bombieri et al., 2006b]. An EFSM provides a more compact representation of the states and transitions of a process than a traditional Finite State Machine (FSM).

In its simplest form, an EFSM has a single state and no variables, and is represented by purely combinational logic; the output values are a function of the input values.

An EFSM with one or more variables may have a single state with one or more transitions representing the allowable value changes of those variables. For instance, a counter has a transition to set or reset the count variable to zero and a transition to increment the count. The input is `reset` and the output is `count`. Each transition has a condition or guard that enables the transition (Figure 2.3).

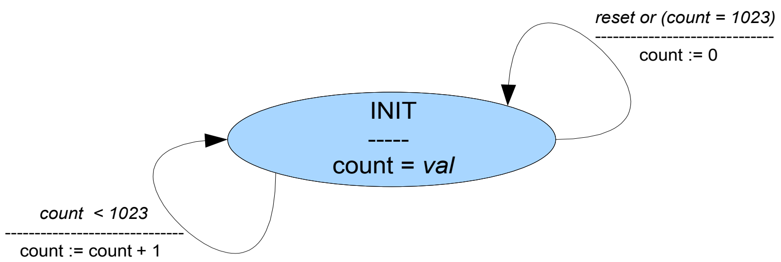


FIGURE 2.3: Counter

In general, an EFSM has a single state variable, zero or more other variables, a set of states that the state variable may take, and a set of transitions between those states. EFSMs are deterministic; only one transition may be enabled from any given state.

2.6.2.2 Modelling TLM level 2 Communications

Each process communicates with other processes using uni-directional channels. These channels are based on the standard, bounded SystemC queue and replace the `read` and `write` calls of the level 3 model. Thus, the Master and Slave at level 2 are modelled as two distinct processes with an output channel to represent the `write` and an input channel to represent the `read` as shown in Figure 2.4. The TLM standard defines `put` and `get` operations for the channels.

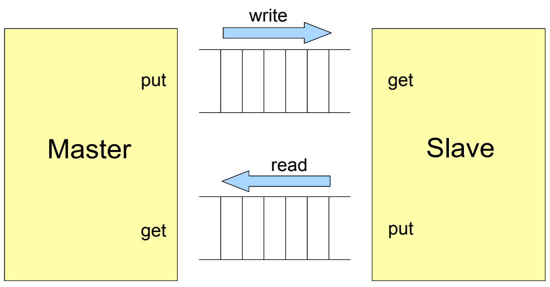


FIGURE 2.4: TLM Level 2 Master/Slave Communication

The level 2 processes are also sensitive to input changes: when a process resumes it evaluates the actions associated with its current state and the values of its inputs, and sets the value of the next state. This set of actions is atomic and represents a process `event` which must be scheduled using the simulation kernel. Simulation is therefore slower than at TLM level 3.

TLM level 2 simulation is `event accurate`: the order in which the process events occur is deterministic and an event can represent a transaction that in the hardware implementation will take place over several clock cycles. Simulation is therefore faster than at RTL where every cycle of the transaction must be scheduled through the simulation kernel.

2.6.3 TLM level 1

TLM level 1 retains the high-level data types of level 2 but now introduces a clock which synchronises the process behaviour: each process is sensitive to the rising (or falling) edge of the clock and the process executes even if its state not going to change. TLM level 1 is termed `cycle accurate`. Because of these extra process evaluations and the fact that each cycle of a transaction must be simulated, simulation is slower than at TLM level 2.

This is the final level of abstraction before bit and bit-vector data types are introduced in the RTL.

2.6.4 TLM Architectural Exploration

TLM level 3 is not suitable for architectural exploration because of its untimed, behavioural nature. It is at levels 1 and 2 that designers can begin to explore different possible architectures to meet performance and power consumption constraints. To facilitate such exploration it is desirable to have an automated route to RTL and early feedback of performance and power consumption estimates from tools downstream in the flow.

Forte Design Systems have introduced the TLM synthesis tool Cynthesizer [Cline, 2007a] which promises such an automated route from TLM to RTL and [Cline, 2007b] discusses the early adoption of TLM synthesis by design teams. As TLM synthesis technology matures, designers will be able to focus most of their attention at the electronic system-level (ESL) [Henkel, 2003] rather than RTL and to utilise the performance and power estimation tools currently available at RTL.

It has been a limitation of the TLM standard at level 1 and 2 that standard bus interfaces had to be modelled using standard channels. The June 2007 release of the standard addresses this issue by providing an abstract, black-box bus interface that takes into account the capabilities of industry-standard buses such as AMBA [ARM, 1999], AXI [ARM, 2003] and OCP (www.ocp.org).

This new bus interface facilitates architectural exploration greatly, but is it possible to raise the level at which architectural exploration can be done to the specification level?

2.7 Microprocessor Pipeline Verification

Early work in the formal verification of microprocessors was focused on simple, non-pipelined processors described at the Register Transfer Level (RTL). In [Joyce et al., 1986] the RTL is represented in the ML programming language and the HOL proof assistant system [Gordon and Melham, 1993] used to discharge the proofs.

In [Burch and Dill, 1994] and [Jones et al., 1995] the representation of the processor is raised to the Instruction Set Architecture (ISA) level and the techniques described focus on the formal verification of the control logic of first a 3-stage pipelined ALU and then the full 5-stage DLX processor. ALU operations are represented as uninterpreted functions. In order to show that the pipelined processor will behave in the same way as a notional non-pipelined version, the concept of pipeline *flushing* is introduced. *Stall* instructions are introduced at the pipeline input to ensure that each instruction is completed before the next is initiated. The notion of *refinement maps* are introduced in [Manolios, 2000] and [Manolios and Srinivasan, 2005a] to extend the flushing concepts of Burch and Dill to more complex 3 and 10-stage pipelines, using the ACL2 functional programming language and theorem prover [Kaufmann and Moore, 2004]. [Manolios and Srinivasan, 2005b] takes a pipelined processor model, translates it into a combination of two different formal languages and then shows that this translation meets its specification. None of these methods is incorporated into the top-down design flow, and the formal verification is performed when the RTL has been completed.

[Tahar and Kumar, 1994] focuses its attention on the formalization of the pipeline hazards that can occur when multiple instructions are executed at once in the DLX pipeline. Structural, data and control hazards are represented and checked using the HOL verification system [Gordon and Melham, 1993]. Incremental design techniques with refinement are described in [Borger and Mazzanti, 1997] to show that a notional DLX pipeline with, however, no overapping instruction execution, can be refined to a pipeline that executes 5 instructions at each clock cycle and manages structural hazards does not encounter a sequence of instructions that would incur data or control hazards. This pipeline is then further refined to model the data and control hazards. Abstract State Machines (ASMs) are used to represent the DLX instructions. In [Kroening and Paul, 2001], a tool that takes a sequential model of the DLX pipeline, which is assumed to be correct, and adds the forwarding logic is described. The tool also provides a proof of correctness for the generated hardware. [Plosila and Sere, 1997] uses Action Systems in a refinement-based approach to processor verification, but does not have the benefit of tool support.

2.8 Modelling Concurrency

2.8.1 Modelling Concurrency with Partial Orders

A process in a concurrent system may be represented as a set of partially ordered multisets (pomsets) [Pratt, 1986]. Each element of a partial order represents an event, and for two events `e` and `f`, `e < f` is interpreted as "event `e` precedes event `f`". Events are not necessarily atomic and can represent `intervals` as well as `instants`. In this case, `e < f` means "the whole of `e` must precede the whole of `f`" . `e` must complete before `f` can begin. Actions label events, and an event is an instance (occurrence) of its action.

The motivation for studying partial orders is to investigate an Event-B modelling style where the specification is as loose as possible in the early stages of refinement, decisions about actual event ordering are deferred for as long as possible and potential concurrency is identified as early as possible. Specifying the behaviour of a component using partial orders means that the specification can be re-used when the component is subsequently re-targeted to a different hardware architecture for power or performance reasons.

Modelling a Communications Channel as a Partial Order

A channel is a process in which the events `Transmit` (T) data and `Receive` (R) data occur in ordered pairs:-

`(data, T) < (data, R)`

For the stream of data `011`, the behaviour of the channel is specified in Figure 2.5.

$$\begin{tikzcd}
(0, T) \arrow[r] \arrow[d] & (1, T) \arrow[r] \arrow[d] & (1, T) \arrow[d] \\
(0, R) \arrow[r] & (1, R) \arrow[r] & (1, R)
\end{tikzcd}$$

FIGURE 2.5: channel behaviour

The actual order in which `transmits` and `receives` occur in the implementation will depend on the performance characteristics of the processes on each end of the channel.

Communication Refinement using Partial Orders

[Lieverse et al., 2001] describes a methodology with which multi-task applications can be designed at the abstract level using high-level inter-task communication. The communication can then be refined by transforming the high-level view of the communication primitives into a partially-ordered representation using lower-level primitives. Then, depending on the hardware architecture, the partial order can be refined into the appropriate total order. Factors that govern the most appropriate total order are whether communication is enabled with channels (message passing) or shared memory and how much local memory a processor has to store local results before passing them to the next process.

2.8.2 Modelling Concurrency with Guarded Atomic Actions

Any hardware component can be represented as a set of variables that comprise the state of the component, and a set of actions on those variables that updates the state. The actions are atomic, all the variable updates comprising an action occur simultaneously, and are protected by guards, variable predicates, that determine which actions are enabled or disabled for a given state of the component. If more than one action is enabled in a given state, a single action is chosen, non-deterministically, for evaluation. The behaviour of the component is represented by the set of legal sequences of atomic actions.

2.8.3 Bluespec

Bluespec is a set of commercial tools, developed from research carried out at MIT [Rosenband and Arvind, 2004], which allows hardware components to be specified as a set of guarded atomic actions, called rules, simulated and then synthesised automatically to RTL. The toolset enables architectural exploration to be done at the specification level, and then RTL to be generated to meet differing power, size and performance constraints. Links are provided to fit in a SystemC TLM flow and, in particular, a library is provided that converts TLM put and get calls into a sequence of bus-specific operations. AMBA, AXI and OCP busses are supported.

Bluespec synthesis is based on a term-rewriting system [Hoe and Arvind, 1999], but no formal verification capability is provided for Bluespec models; the tools set is purely simulation based. The semantics of guarded atomic actions are not strictly adhered to, and it is possible for multiple enabled actions to be executed simultaneously. It is also possible for the user to apply priorities to actions using tool pragmas. Bluespec rules are translated to a set of RTL concurrent assigments, which are processes sensitive to changes to variables on the right hand side of the assignment. Simultaneously-firing rules are therefore mapped to concurrent assignments that are evaluated simultaneously in the target HDL; the left hand sides of the assignments are not updated until all the variables on the right hand sides have had their values updated, preventing race conditions.

2.8.4 CAL

CAL [Bhattacharyya et al., 2008] is a language, also based on guarded atomic actions, for hardware/software co-design. It provides routes for both hardware and software synthesis, but no formal verification capability has been developed.

2.9 Event-B

2.9.1 Introduction

Event-B [Abrial and Mussat, 1998], [Hallerstede, 2007] is a proof-based modelling language and method that enables the development of specifications using refinement. The Rodin platform [Abrial et al., 2006] is the Eclipse-based IDE that provides automated support for Event-B modelling, refinement and mathematical proof.

In Event-B, an abstract model comprises a *machine* that specifies the high-level behaviour and a *context*, made up of sets, constants and their properties, that represents the type environment for the high-level machine. The machine is represented as a set of *state variables*, v and a set of events, *guarded atomic actions*, which modify the state. If more than one action is enabled, then one is chosen non-deterministically for *execution*, an observable transition on the state variables which must preserve an *invariant* on the variables, *I(v)*. A more concrete representation of the machine may then be created which refines the abstract machine, and the abstract context may be extended to support the types required by the refinement. *Gluing invariants* are used to verify that the concrete machine is a correct *refinement*: any behaviour of the concrete machine must satisfy the abstract behaviour. Gluing invariants give rise to proof obligations for pairs of abstract and corresponding concrete events. Events may also have parameters which take, non-deterministically, the values that will make the guards in which they are referenced true.

An event can be represented by the *generalized substitution*,

$$\boxed{\textbf{any } x \textbf{ where } P(x,v) \textbf{ then } v := F(x,v) \textbf{ end}}$$

where x represents the event parameters and v represents the machine state variables. Informally, this event can be fired provided that the guard $P(x, v)$ can be satisfied for some value x. The details are explained in [Abrial, 2005].

2.9.2 Refinement

Event-B refinement allows a model to be built gradually [Abrial and Hallerstede, 2006], starting with an abstract model and then introducing successive, more concrete refinements. Adding variables achieves *spatial extension* and adding events *temporal extension*. Events in the abstract model may be refined by one or more events in the concrete model. New events, which refine *skip* may also be introduced in the refinement. *Data-refinement* [Abrial, 2005] can also be used to modify the state so that an abstract variable can be replaced with a concrete variable that can be implemented in the target hardware or software.

2.9.3 Decomposition

Event-B supports two mechanisms for formal composition and decomposition; *shared event* [Butler, 2009] and *shared variable* [Abrial and Hallerstede, 2006]. Shared event decomposition has tool support in Rodin [Silva et al., 2010].

Shared Event Decomposition

[Butler, 2009] describes a parallel composition operator for machines, where the composition of machines M and N is written $M \mid\mid N$. Machines M and N synchronise over *shared events* which have common names. If eM and eN are the shared events in M and N respectively and m and n are the (disjoint) variables of M and N respectively, then $M \mid\mid N$ is defined as follows.

if

$eM =$ **any** x **where** $P(x,m)$ **then** $v1 := F(x,m)$ **end**

$eN =$ **any** y **where** $Q(y,n)$ **then** $v2 := G(y,n)$ **end**

then

$eM \mid\mid eN =$ **any** $x,\ y$ **where** $P(x,m) \wedge Q(y,n)$ **then** $v1 := F(x,m) \mid\mid v2 := G(y,n)$ **end**

Shared Variable Decomposition

An alternative to shared event decomposition is presented in [Abrial, 2009]. Instead of synchronising over shared events, the machines M and N communicate using shared common variables, which must be replicated in M and N. When $M \mid\mid N$ is decomposed, however, it is necessary to impose the restriction that the shared common variables must not be *data-refined* in any subsequent refinements of M or N. Additional events must also be introduced into M and N, called *external events*, which *simulate* the way the shared events are managed in the composition $M \mid\mid N$.

2.9.4 Variants

A *convergent* event is an event which refines *skip* and has some liveness property associated with it through the definition of a *variant*. A variant is a natural number expression that must be decreased by the convergent event, or a finite set expression that must be made strictly smaller by the convergent event. An *anticipated* event is an event that refines *skip* that is not convergent but will become convergent in a later refinement. [Abrial, 2007]

2.9.5 Records

Structuring of data can be achieved in Event-B by using the conventions established in [Evans and Butler, 2006]. For a set representing the data type T, *projection functions* can be defined which map a field of T to the field's value. For instance, if T has two fields *F1* and *F2* which are natural numbers and one field *F3* which is an integer, then the record can be described thus.

$\texttt{axm1}: \mathit{F1} \in T \rightarrow \mathbb{N}$

$\texttt{axm2}: \mathit{F2} \in T \rightarrow \mathbb{N}$

$\texttt{axm3}: \mathit{F3} \in T \rightarrow \mathbb{Z}$

2.9.6 Witnesses

When an abstract event with parameter p is refined then the parameter p in the concrete event corresponds directly with that in the abstract event. If, however, the parameter p does not appear in the concrete event, then p must receive a concrete value, called a *witness* in the refinement. [Abrial, 2007]

Chapter 3

Enhancing the SoC Hardware Design Flow with Event-B

This chapter covers the modern synchronous design flow and the gap that exists in this flow between the specification and the RTL description used for logic synthesis. It looks at the reasons that behavioural synthesis has not in general helped to close this gap, and how the use of guarded atomic actions can provide efficient high-level synthesis and architectural exploration. Although semi-conductor companies express a clear need to raise the design process to the Electronic System Level (ESL) [Asanovic et al., 2006], the approach that the EDA industry has taken to address this need has been fragmentary, no clear standards have emerged, and the tried and proven RTL design methodology still forms the significant bedrock of any modern SoC design flow (Figure 3.1). The hardware description languages are mature and simulators can be second-sourced. It is, however, the increasing maturity of Logic Synthesis [Devadas et al., 1994] that has made the most significant contribution.

The contribution of this chapter is to show how the Event-B formal method can be incorporated into this flow and combined seamlessly with high-level and RTL synthesis to close the specification gap.

3.1 Background to the Existing Flow

3.1.1 Logic Synthesis

Logic Synthesis, as shown in Figure 3.2, contributes the first step in bridging the gap between the textual specification and the gate level description from which hardware can be generated automatically. Although RTL descriptions raise the level of abstraction

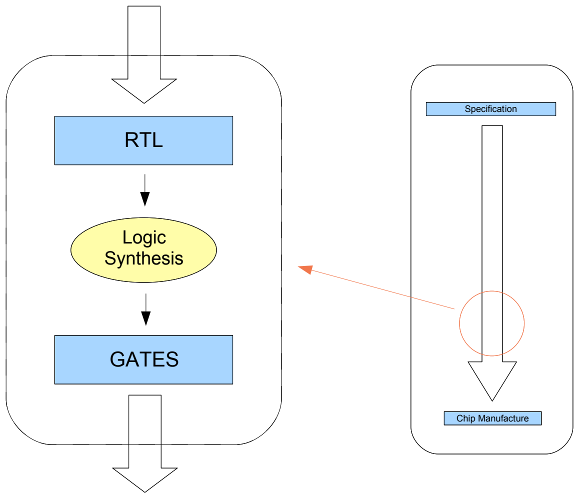


FIGURE 3.1: SoC Flow

significantly, enable language-driven design and reduce simulation effort, if the translation route to gates were a manual process the RTL would simply represent some useful documentation and would make little contribution to closing the specification gap.

Early users of logic synthesis expended significant simulation effort on verifying that the output from synthesis was correct. Today, very little gate level simulation is performed, not only because logic synthesis is mature but also, more importantly, because formal tools have been introduced to augment the design flow.

3.1.2 Formal RTL Verification

Although modern logic synthesis tools are mature and reliable, they are also very large and complex and present the user with an enormous range of control options. A logic synthesis tool could never represent a *trusted component* [Meyer et al., 2003] in the SoC flow in the sense of representing a provable transformation on the RTL. It would be infeasible to show that the tool, given an RTL description as input produced an *equivalent* gate level description as output. It is possible, however, given the logic synthesis input and output description, to reason about the correctness of the translation using the formal methods *property checking* [J.R. Burch et al., 1990] and *equivalence checking* [Drechsler and Horeth, 2002], as shown in Figure 3.3

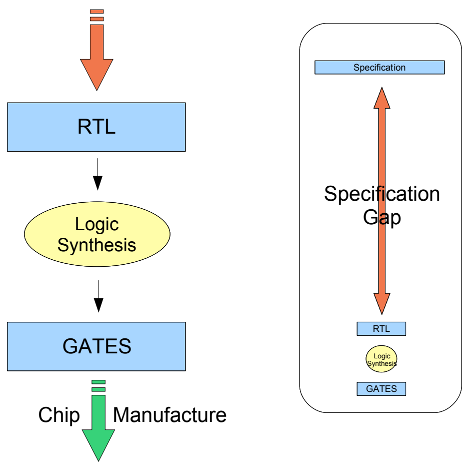


FIGURE 3.2: The Role of RTL Synthesis

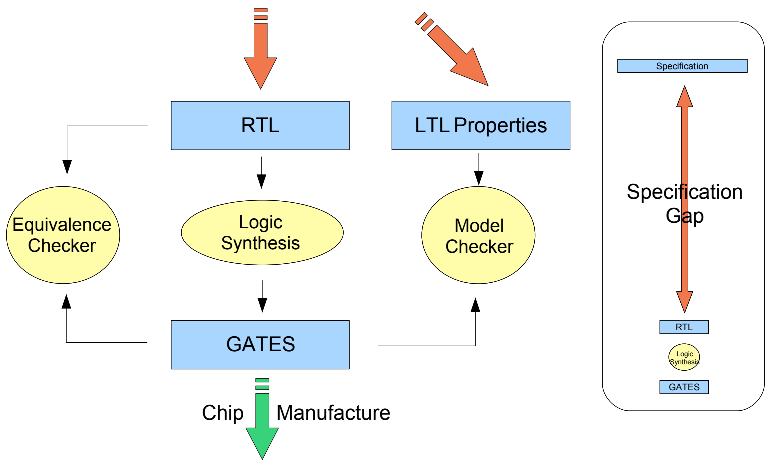


FIGURE 3.3: The Role of Formal RTL Checking

Property Checking

A property checking tool takes a set of temporal properties [Cohen et al., 2004], [Sutherland et al., 2004], derived from the specification and checks these properties against the synthesised gate-level description.

Property checking depends on having a comprehensive set of properties to represent the desired behaviour. It is very difficult to establish whether sufficient properties have been written or not, and *property coverage* [Chockler et al., 2001], [Hoskote et al., 1999] is a topic of ongoing research.

Without a measurable outcome, property checking will not become an indispensable component in the SoC flow, but it does provide an independent check on the validity of the design, and is a valuable auxilliary method for flushing out design faults.

Equivalence Checking

An equivalence checking tool takes the RTL and the synthesised netlist as inputs, converts both into an internal gate-level format and then verifies that they are functionally equivalent. Equivalence checking is a mature technology, requires little user interaction and is now widely used to verify synthesis output.

The combination of logic synthesis with equivalence checking, which provides practical and efficient support for RTL design and a formal link between RTL and gate-level views, represents a significant breakthrough in raising the abstraction level in hardware design. In particular, it raises the level at which architectural exploration and component re-use can be considered.

3.1.3 RTL Architectural Exploration

Before the adoption of RTL synthesis, when design was done using gate-level schematic capture, arriving at an architecture that satisfied performance constraints was laborious and time consuming. Components designed for re-use were only available as gate-level descriptions, which could require significant re-work to be of value in different designs. Once an appropriate architecture had been settled on to target a given hardware technology, represented by a library of cell primitives, even targeting a similar library for a different hardware technology could require significant and time-consuming changes to the gate-level description, including changes to the supposedly re-usable components.

The first clear benefit of RTL synthesis is that it facilitates technology-independent design. The second benefit is that it is possible to explore micro-level architectural alternatives without changing the RTL description. Using the rich set of control options in the synthesis process it is possible to generate differing gate-level implementations to meet differing performance requirements. Re-usable components are now provided as RTL descriptions which can therefore also be manipulated by the synthesis process.

If, however, it is not possible to reach the required targets through synthesis alone, alternative RTL architectures must be developed, and even though RTL/RTL equivalence checking can be used to ensure functional equivalence between different RTL representations, RTL architectural exploration, shown in Figure 3.4, is still very time consuming.

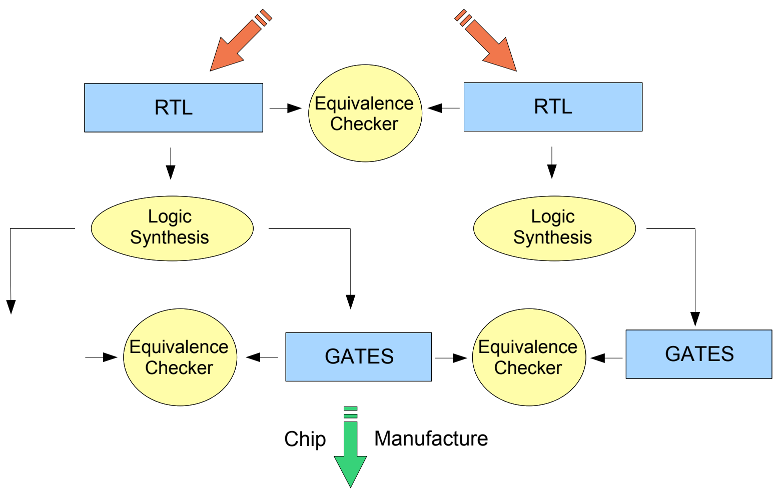


FIGURE 3.4: RTL Architectural Exploration

3.1.4 Closing the Gap: Behavioural Synthesis

As SoC's have become more complex, managing system design, architectural exploration and component re-use at the Register Transfer Level, has become increasingly difficult. Since high-level, behavioural models and system-level specifications are already developed using programming languages such as C and C++, behavioural synthesis from a programming language source is an attractive concept [Edwards, 2005]. If efficient hardware could be generated automatically from a collection of behavioural descriptions, and re-usable components could also be represented in this way, then the gap between specification and implementation would close dramatically. Behavioural Synthesis takes a behavioural description and generates an RTL description that can then be used as an input to logic synthesis. The behavioural synthesis flow is shown in Figure 3.5.

Behavioural Synthesis Issues

Although there is a clear relationship between RTL processes and their gate level equivalents, and the transformations between the two levels are well defined, no such relationships and transformations can be easily defined between a behavioural description and an RTL description. Fundamentally, RTL descriptions by their very nature contain architectural information and behavioural descriptions do not. What is needed is a high

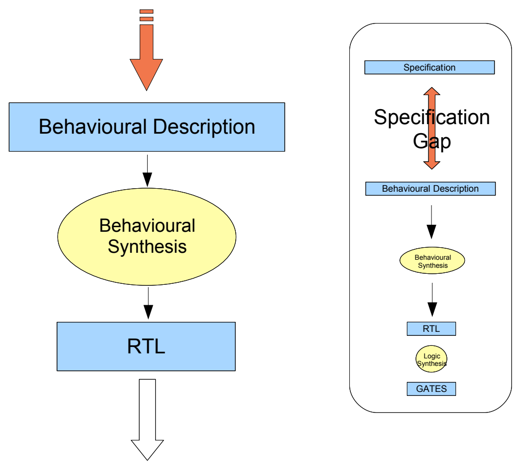


FIGURE 3.5: Behavioural Synthesis Flow

level representation that can represent the architecture of the target implementation in an abstract way and is amenable to formal reasoning.

3.1.5 Closing the Gap: High-level Synthesis with Term Rewriting Systems

Term Rewriting Systems (TRS) [Baader and Nipkow, 1998] offer a natural way to describe hardware architectures which facilitates architectural exploration at the specification level and allows the designer to explore design trade-offs much earlier in the process [Arvind and Shen, 1999]. TRS can be used to describe both deterministic and non-deterministic behaviour; as an abstract specification is successively refined the non-determinism, which aids greatly in the representation of concise abstract specifications, can systematically be made more concrete. In addition to being amenable to formal analysis, TRS descriptions, in the form of *guarded atomic actions*, have also been demonstrated [Hoe and Arvind, 1999], [Arvind and Hoe, 1999], [Arvind et al., 2004], [Hoe, 2004] to be amenable to high-level synthesis [R. Kumar et al., 1996]. Once a detailed TRS description of a hardware component is available, there is an automatic route to an RTL description [Rosenband and Arvind, 2005] and therefore to hardware using current logic synthesis flows [Devadas et al., 1994]. The commercial, high-level synthesis tool, Bluespec [Nikhil, 2007], is already being used successfully in industrial flows (Figure 3.7).

Term Rewriting Systems

A Term Rewriting System is a set of *rules* which defines a set of transitions on *state*, represented by TRS *terms*. Each rule is guarded by a predicate on the current state and has an associated *atomic action* which updates the state [Arvind and Shen, 1999]. More formally, a TRS is defined as a $tuple(S, R, S_0)$ where S is a set of terms, R is a set of re-writing rules and S_0 is a set of initial terms, $S_0 \subseteq S$. *States* are represented by TRS *terms* and *transitions* are represented by TRS *rules*. A *rule* is of the form

$$\begin{array}{rl} & s1 \quad if\ p(s1) \\ \rightarrow & s2 \end{array}$$

where $s1$ and $s2$ are terms and p is a predicate. The term $s1$ is rewritten to the term $s2$.

Consider a microprocessor $Proc$ with an instruction address (program counter) ia, a register file rf and instruction memory im. Any given state of the processor can be represented as $Proc(ia, rf, im)$. The predicate representing the next instruction to be processed in the instruction memory (at the instruction address) can be written as say $im[ia]$ = "$rd := Op(rs_1, rs_2)$". If, for a given state of the processor, this instruction is encountered in the instruction memory, then the rule will fire and the state of the processor will updated be appropriately: the instruction address will be incremented to point to the next instruction location and the value of the target register rd in the register file will be updated to reflect the result of the operation on the two source registers rs_1 and rs_2. The state of the instruction memory im is unchanged. This rule is shown in Figure 3.6.

$$\begin{array}{ll} \mathrm{Proc}(ia, rf, im) & \text{if } im[ia] = rd := \mathrm{Op}(rs_1, rs_2) \\ \mathrm{Proc}(ia + 4, rf[rd := \mathrm{v}], im) & \text{where } v := \mathrm{Op}(rf[rs_1], \mathrm{rf}[rs_2]) \end{array}$$

FIGURE 3.6: TRS Processor Instruction Rule

If, for a given state, the predicates of more than one of the rules evaluate to *true*, then one of the enabled rules is chosen non-deterministically for evaluation. [Dave, 2005] describes the use of the commercial high-level synthesis tool Bluespec [Nikhil, 2004], which generates RTL from a TRS description, to design a processor. In practice the TRS synthesis tool allows the use of pragmas to impose event ordering, and if simultaneously enabled events do not conflict (do not attempt to write different values to the same variable) they are scheduled to be evaluated simultaneously. The synthesis tool generates an RTL scheduler to implement these semantics. At present, no formal checker exists to verify that the derived RTL is a correct refinement of the TRS description, but the nature of *guarded atomic actions* means that they are amenable to formal analysis, and the transformations from TRS *rules* to RTL *processes* could be formalised. How TRS-based high-level synthesis fits in the design flow is shown in Figure 3.7.

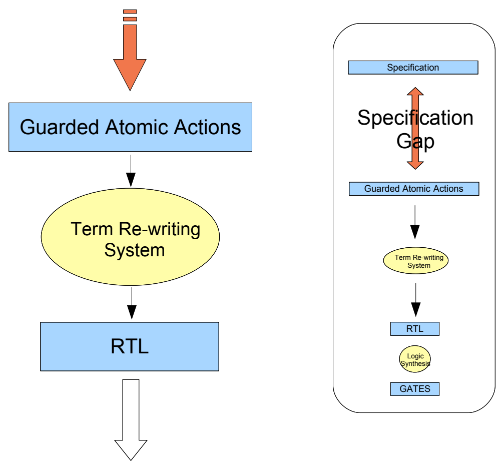


FIGURE 3.7: TRS Synthesis Flow

TRS Architectural Exploration

TRS synthesis enables the abstraction level at which architectural exploration can be conducted to be raised considerably [Arvind et al., 2004]. At a very early stage of the design it is possible to explore different architectures and use the downstream TRS and RTL synthesis flow to generate a gate level description for performance and power estimation. Figure 3.8 shows that the synthesis process can generate different RTL descriptions from the same TRS input under user control, or different TRS descriptions can be synthesised and compared downstream.

Together with higher-level architectural exploration comes the opportunity for higher-level component re-use [Ng et al., 2007].

TRS Synthesis Issues

Incorporating TRS synthesis into an SoC design flow can decrease the design effort considerably, but currently does not decrease the verification effort, since most of the verification must be conducted downstream in the flow. Bluespec descriptions can be simulated before synthesis, but simulation facilities are limited and can only be used effectively to eliminate gross errors. The major part of the verification effort must

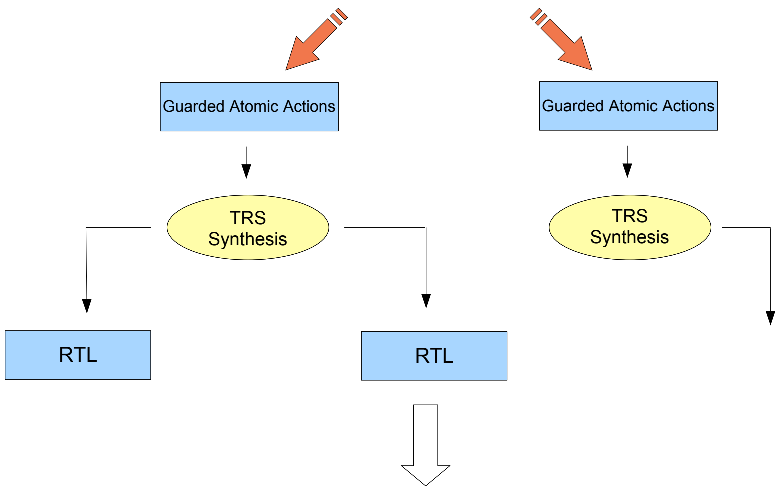


FIGURE 3.8: TRS Architectural Exploration

still be expended on the generated RTL. This effort includes writing test benches for simulation and properties for model checking. To get the full benefit from TRS synthesis it will be necessary to raise at least some of the verification effort to the TRS level.

Bluespec works well for SoC component design, but for the development of SoC sub-systems, complex inter-component interactions, described using Bluespec method invocation, can lead to inefficiency in the synthesised design. Composition is *functional* in Bluespec. A guarded action can make a call to a method which itself can be guarded and can in turn make further method calls. Bluespec differentiates between the top-level *explicit* guard and the *implicit* guards of the called methods [Nikhil, 2007]. Explicit and implicit guards must be combined to determine whether an action is enabled or not and this leads to complexity and strong coupling between the components. [Asanovic, 2007] identifies this problem and proposes an approach where composition is *connection-based*. Bluespec is used to model the components and then an explicit inter-component communication mechanism is used to model the sub-systems, decoupling the components with message queues. What is clearly needed is an environment in which both components and sub-systems can be modeled and reasoned about in a systematic way.

The specification gap still remains. Textual or behavioural descriptions of the specification must be transformed by hand to TRS descriptions, and since these descriptions are fundamentally different in nature, this transformation process can be a major potential source of errors.

3.2 Closing the Gap: The Event-B method with TRS

Since Event-B is also based on guarded atomic actions, the semantics for updating the values of variables is the same as for Bluespec, CAL and indeed RTL.

Support for non-determinism in Event-B means that early abstract models can be underspecified in a natural way. In terms of hardware modelling, this non-determinism matches well the desirable characteristic to be able to represent the abstract model as a partial order on a set of events.

Events can also have parameters which take, non-deterministically, the values that will make the guards in which they are referenced true. This provides a flexible and proof-driven mechanism for describing the environment for which a hardware model's behaviour is defined, analogous to the pseudo-random, constraint-based techniques of the simulation test languages such as SpecMan [Hollander et al., 2001] and VERA [Haque and Michelson, 2001].

An event takes the following form:-

```
**Event**  $E \mathrel{\widehat{=}}$
    **any**
        $p_1 .. p_n$
    **where**
        $G_1 \wedge .. \wedge G_n$
    **then**
        $A_1 .. A_n$
    **end**
```

where $p_1..p_n$ are parameters, $G_1..G_m$ are predicates representing the event guards and $A_1..A_p$ atomic assignments representing the event actions.

As explained in Section 2.9 on Page 17, Event-B refinement allows a model to be built gradually [Abrial and Hallerstede, 2006], starting with an abstract model and then introducing successive, more concrete refinements. Adding variables achieves *spatial extension* and adding events *temporal extension*. Events in the abstract model may be refined by one or more events in the concrete model. New events, which refine *skip* may also be introduced in the refinement.

3.2.1 Event-B Specification Refinement in the SoC Flow

A method is proposed where Event-B is used to represent the abstract hardware specification with the TRS-style descriptions familiar to hardware designers who use of Bluespec or CAL. The abstract specification is then refined systematically to reflect the architectural decisions of the designer. It will be shown in the following chapters how the designer will be able to deal with one architectural consideration at a time, refining a particular aspect of the design to a concrete representation while leaving the rest of the representation abstract. At each refinement step it will be shown how the Rodin tool helps the designer to discover the gluing invariants that must be proved to demonstrate that the concrete representation is a correct refinement of the abstract. These invariants are fundamental properties of the design that can be translated directly into PSL or SVA descriptions and used downstream in the flow for RTL formal and simulation-based verification. The refinement process continues until a concrete representation of the specification has been derived that is suitable for either TRS or RTL synthesis. The verification effort has been raised to the specification level because the concrete representation has been proved formally to implement the abstract specification. Verification is made manageable because it is performed incrementally within the design flow. The hardware and its associated properties are described in an event-based language which is a natural vehicle for synchronous hardware description. All the associated proof obligations are generated and proved by the Rodin tool environment. Figure 3.9 shows how the TRS flow is augmented by this method.

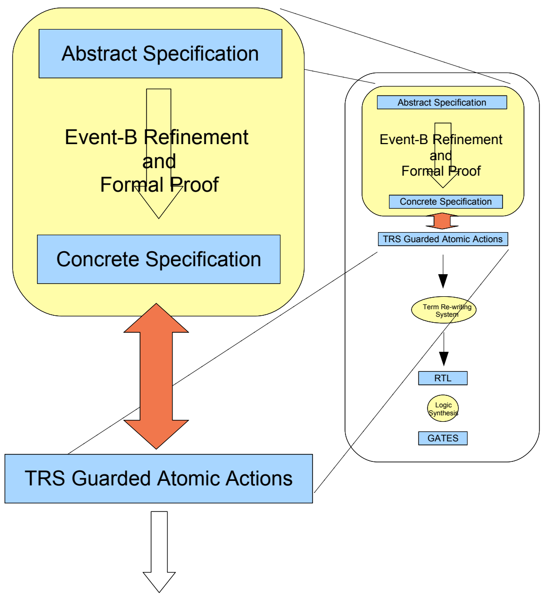


Figure 3.9: Event-B and TRS flow

3.2.2 Architectural Exploration at the Specification Level

The designer can with this method choose to explore different architectures at a very early stage of the development process. As shown in Figure 3.10, alternative concrete representations can be derived from the specification, translated automatically using a TRS mapper to a Bluespec or CAL representation and then passed to tools downstream in the flow that predict the performance and power consumption of the target hardware. This feedback from downstream tools then allows the user to select the appropriate concrete representation or explore further alternatives. Where an existing design is to be targeted to a new hardware platform, architectural decisions can be revisited at the specification level.

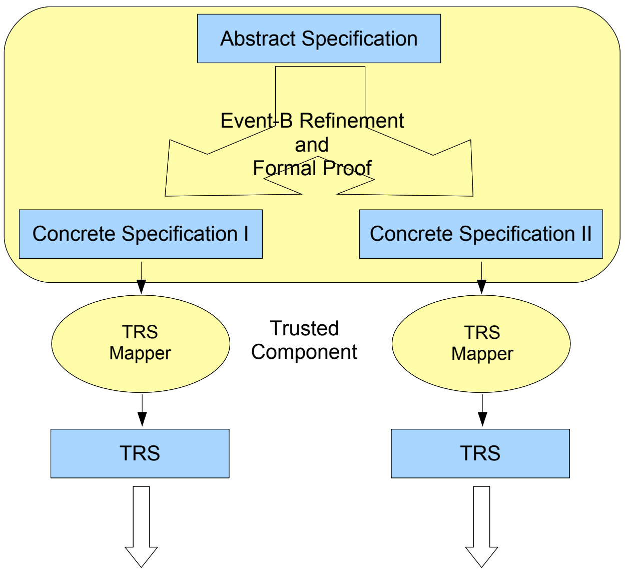


FIGURE 3.10: Event-B Architectural Exploration

In general, increasing concurrency can increase performance, but will increase power consumption. Reducing the clock speed, however, reduces power consumption at a greater rate than it is increased by exploiting concurrency [Kumar et al., 2003]. Therefore architectural solutions which maximise concurrency are often desirable, and is the reason that modern SoCs incorporate increasing numbers of processor cores. It is also the reason that *pipelining* [Hennessy and Patterson, 2006] is so widely exploited. Pipelining uses shared registers to communicate between simple, specialised pipeline stages. The simplicity of each stage contributes to keeping power consumption low because the low gate count minimises the amount of transistor switching. Similary, shared registers can provide an efficient mechanism for communication between state machines.

For synchronous design, however, shared register communication can only be used for localised, *intra-component* communication, because the logic synthesis tools cannot manage the global track delays between components located in non-adjacent areas of the chip. This issue will be dealt with in detail in the next chapter. Transaction Level Modelling (TLM) [Ghenassia, 2006] and Network on Chip (NoC) technology [Gebhardt and Stevens, 2008] have evolved to meet the requirements for *inter-component* communication on SoCs and, in particular, to support component re-use.

To raise architectural exploration and re-use to the specification level it is essential that the issues of *inter*- and *intra-component* communication can be addressed at this level, before synthesis. It is therefore important that the specification method supports both *component* and *sub-system* design and it is an key goal of the following chapters to show that specification refinement with formal proof can be used, starting with an abstract specification, to develop, systematically, alternative sub-system architectures to meet differing performance and power consumption goals.

Chapter 4

Developing SoC Components

This chapter looks at the use of the Event-B method for developing SoC component specifications, and how this relates to the restrictions that hardware process technology places on component size. It first looks at the modelling and refinement of finite state machines in general and then looks at pipelined architectures and the implications that such architectures have for how simultaneous pipeline events are managed. It concludes with a detailed investigation of how latency can be managed in an IP Lookup circular pipeline.

The contribution described in this chapter is the development of a method where an abstract specification of an SoC component can be refined formally to derive a concrete implementation of the component that has a direct correspondence to its HDL description. Whether the concrete implementation is represented by a single process or by multiple processes communicating with shared variables, the method enables the implementation to be verified formally against its abstract specification.

- A single process may be represented in hardware design by a Finite State Machine.
- Multiple processes may be represented by a set of communicating Finite State Machines which communicate with message-passing FIFOs or with shared registers.
- A hardware pipeline is a special case of a set of multiple communicating processes.

An approach is presented where a concrete representation of a single FSM, amenable to hardware synthesis, is derived systematically from an abstract hardware specification using Event-B refinement. The approach is then extended to show how an abstract hardware specification can be modeled, refined and then decomposed to form a concrete, synthesisable pipelined representation, comprising several communicating processes, that has been proved to implement its abstract specification.

This chapter presents the building blocks of the method which will be elaborated in later chapters.

4.1 Restrictions on SoC Component Size

In a modern, low-power System-on-Chip (Soc) development flow it is already possible to incorporate several microprocessors (*multi-core*) in a design [Geer, 2005] and in the near future it is feasible that this number could increase to several hundred (*many-core*) [Asanovic et al., 2006]. The trend towards multi-processor design has come about because the constraints of sub-90 nanometre design mean that it is no longer possible to simply increase processor clock speeds to achieve higher performance [Geer, 2005]. These design constraints, which restrict the size of all hardware processing components, not just microprocessors, result from several factors, which are explored in [Sylvester and Keutzer, 2001].

First, increases in clock speed increase power consumption and cause heat dissipation problems. Second, the small feature sizes mean that the benefits of faster device switching can be negated by the delays incurred when connecting these devices. Synchronous design tool-chains rely on the combinational logic settling between clock edges and these global track delays therefore restrict the speed at which a component can be clocked. As components get larger and more complex, a corresponding increase in global track delay length is also incurred, limiting the size of components that can be incorporated into a SoC. Third, the capacity limitations of current verification tools, whether formal or simulation-based, coupled with the large cost of verifying complex hardware logic also limits the size and complexity that can be handled. There is therefore a compelling argument for SoC hardware components to be kept as simple as possible.

The detailed investigation of these physical factors in [Sylvester and Keutzer, 2001] concludes that SoC component size needs to be restricted to between 100,000 and 200,000 gates, and that a Network on Chip (NoC) protocol [Carloni and Sangiovanni-Vincentelli, 2002] is then used to manage the communication between components.

This chapter focuses on the methods required to refine the different types of hardware component encountered on an SoC, from a high-level specification to a representation that is suitable for high-level or RTL synthesis. Chapter 7 will address the development of sub-systems and inter-component communication.

4.2 State Machines

For an abstract model, represented as a Term Rewriting System or in an HDL at the Transaction Level, each constituent process may be represented by an Extended Finite

State Machine (EFSM) [Gajski et al., 1997], [Bombieri et al., 2006b]. An EFSM provides a more compact representation of the states and transitions of a process than a traditional Finite State Machine (FSM).

In its simplest form, an EFSM has a single state and no variables, and is represented by purely combinational logic; the output values are a function of the input values.

An EFSM with one or more variables may have a single state with one or more transitions representing the allowable value changes of those variables. For instance, a counter has a transition to set or reset the count variable to zero and a transition to increment the count. The input is `reset` and the output is `count`. Each transition has a condition or guard that enables the transition (Figure 4.1).

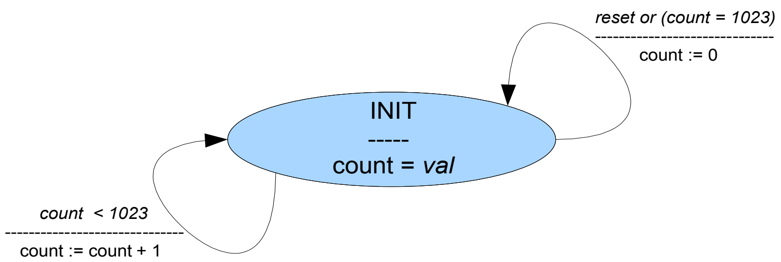


FIGURE 4.1: Counter

In general, an EFSM has a single, explicit state variable, zero or more other variables, a finite set of states that the state variable may take, and a set of transitions between those states. EFSMs are deterministic; only one transition may be enabled from any given state.

4.3 Case Study: Developing an EFSM for Huffman Encoding/Decoding

Huffman Encoding [Huffman, 1952] is an algorithm used for lossless data compression. In this development, an EFSM implementation of the Huffman Encoding/Decoding Algorithm is derived formally from its abstract specification using Event-B refinement and proof. The purpose of this development is to encode a stream of vowels into a stream of bit values, decode the stream and prove that the decoded encoded stream is identical to the original stream. An encoding tree for the letters A, E, I, O, and U is constructed using the probabilities of encountering a given letter in the stream. The more commonly encountered the character the shorter the bit encoding. The tree is shown in Figure 4.2.

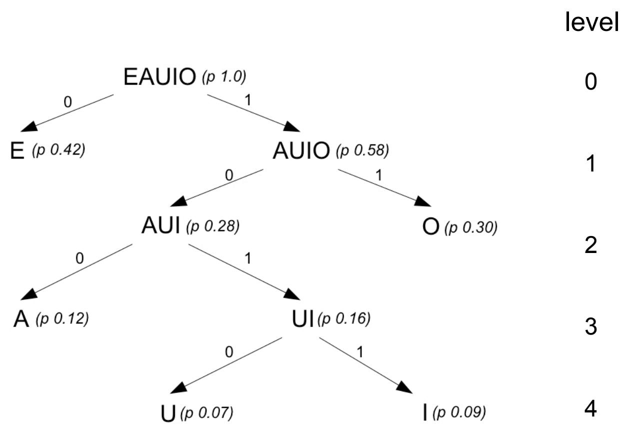


FIGURE 4.2: HUFFMAN Encoding Tree

The development begins with an abstract model which defines the control flow for the encoding/decoding process. In the first refinement, the character to be encoded is read from the input stream and stored. Then, in the second refinement the bit encoding for each character is introduced. Finally, in the third refinement the bit encoding of the character is decoded and gluing invariants introduced to prove that the decoded encoded stream is identical to the original stream.

4.3.1 The Abstract Model

We present an abstract model of Huffman Encoding/Decoding using Event-B. Two state variables `char_to_encode` and `string_to_decode` and three events `get_char`, `encode` and `decode` are identified. The partial order on events is shown in Figure 4.3. After the first `encode` event, `get_char` and `decode` are enabled simultaneously and can occur in any order.

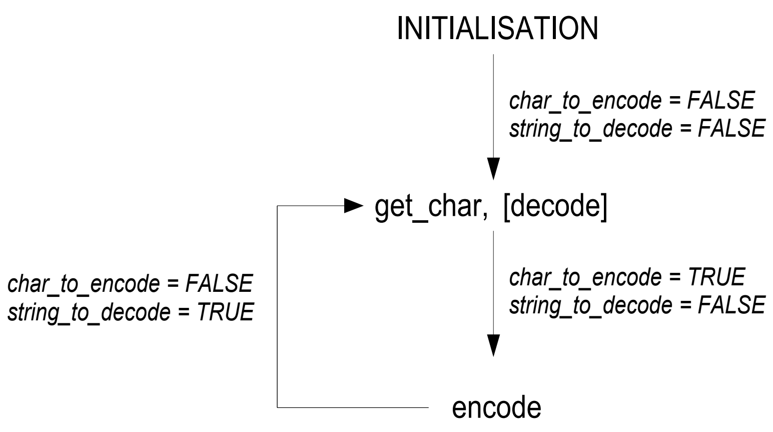


FIGURE 4.3: HUFFMAN Partial Order

The alphabet is represented as an enumerated set and `get_char` does a non-deterministic assignment to the variable `current_char` to represent the input character stream.

CONTEXT HUFFC

SETS

`Alphabet`

CONSTANTS

`A`, `E`, `I`, `O`, `U`

AXIOMS

axm1 : $Alphabet = \{A, E, I, O, U\}$

END

At this level of abstraction there is no explicit encoding or decoding; the model simply describes the control logic.

MACHINE HUFFM

SEES HUFFC

VARIABLES

`current_char`, `char_to_encode`, `string_to_decode`

INVARIANTS

inv1 : $current\_char \in Alphabet$

inv2 : $char\_to\_encode \in BOOL$

inv3 : $string\_to\_decode \in BOOL$

EVENTS

Initialisation

```
begin
    act1 : $current\_char := A$
    act2 : $char\_to\_encode := FALSE$
    act3 : $string\_to\_decode := FALSE$
end
```

Event $get \mathrel{\widehat{=}}$

```
any
    $char$
where
    grd1 : $char \in Alphabet$
    grd2 : $char\_to\_encode = FALSE$
then
    act1 : $char\_to\_encode := TRUE$
    act2 : $current\_char := char$
end
```

Event $encode \mathrel{\widehat{=}}$

```
when
    grd1 : $char\_to\_encode = TRUE$
    grd2 : $string\_to\_decode = FALSE$
then
    act1 : $char\_to\_encode := FALSE$
    act2 : $string\_to\_decode := TRUE$
end
```

Event $decode \mathrel{\widehat{=}}$

```
when
    grd1 : $string\_to\_decode = TRUE$
    then
        act1 : $string\_to\_decode := FALSE$
    end
END
```

4.3.2 The First Refinement

The `encode` event is refined to introduce 5 new events; one for each character. A new variable `encoded_char` is introduced to store the value of the character from the input stream. This variable is required because `current_char` can be overridden before encoding and decoding operations on the character are complete. The model has potentially a one stage pipeline and, in general, one element of storage is required for each pipeline stage. In practice, this means that there is always a trade-off between performance and power consumption. At this stage the decision as to whether to exploit the pipelining or not is left open.

For example, the event representing the encoding of the letter `E` is as follows.

```
Event  $encode\_E \widehat{=}$
refines  $encode$
    when
        grd1 : $char\_to\_encode = TRUE$
        grd2 : $string\_to\_decode = FALSE$
        grd3 : $current\_char = E$
    then
        act1 : $char\_to\_encode := FALSE$
        act2 : $string\_to\_decode := TRUE$
        act3 : $encoded\_char := E$
    end
```

The following invariants are introduced which define `encoded_char` and establish its relationship with the abstract variable `current_char`.

```
INVARIANTS
    inv1 : $encoded\_char \in Alphabet$
    inv2 : $char\_to\_encode = FALSE \land string\_to\_decode = TRUE \Rightarrow current\_char = encoded\_char$
```

4.3.3 The Second Refinement

Now the bit encoding is introduced using four boolean variables, and each of the `encode` events extended to incorporate the encoding. For example:-

Event $encode\_A \mathrel{\widehat{=}}$

refines $encode\_A$

when

- `grd1` : $char\_to\_encode = TRUE$
- `grd2` : $string\_to\_decode = FALSE$
- `grd3` : $current\_char = A$

then

- `act1` : $char\_to\_encode := FALSE$
- `act2` : $string\_to\_decode := TRUE$
- `act3` : $encoded\_char := A$
- `act4` : $bit0 := TRUE$
- `act5` : $bit1 := FALSE$
- `act6` : $bit2 := FALSE$

end

New invariants relate the encoded char to the bit variables.

INVARIANTS

- `inv1` : $bit0 \in BOOL$
- `inv2` : $bit1 \in BOOL$
- `inv3` : $bit2 \in BOOL$
- `inv4` : $bit3 \in BOOL$
- `inv5` : $string\_to\_decode = TRUE \land encoded\_char = E \Rightarrow bit0 = FALSE$
- `inv6` : $string\_to\_decode = TRUE \land encoded\_char = O \Rightarrow bit0 = TRUE \land bit1 = TRUE$
- `inv7` : $string\_to\_decode = TRUE \land encoded\_char = A \Rightarrow bit0 = TRUE \land bit1 = FALSE \land bit2 = FALSE$
- `inv8` : $string\_to\_decode = TRUE \land encoded\_char = U \Rightarrow bit0 = TRUE \land bit1 = FALSE \land bit2 = TRUE \land bit3 = FALSE$
- `inv9` : $string\_to\_decode = TRUE \land encoded\_char = I \Rightarrow bit0 = TRUE \land bit1 = FALSE \land bit2 = TRUE \land bit3 = TRUE$

4.3.4 The Third Refinement

Up to now the `decode` event has done nothing but update the state. In this refinement events and variables are introduced to model the encoding tree. The variable `decoded_char` is introduced to store the decoded encoded value, and the variable `level` which is set to zero to represent the root and is incremented during the tree walk. For the events that refine `skip` it is necessary to introduce a variant which must be decreased by these events.

VARIANT

$3 - \mathtt{level}$

The proof obligations generated for the variant ensure that the tree walk terminates.

The decode events in this refinement are named `decode_L_B` where L represents the tree level and B represents the branch taken in the search. The encoding tree is represented using two types of event: those which represent the leaves of the tree and refine `decode`. For instance,

```
Event  decode_2_0 ≙
refines  decode
    when
        grd1 : $string\_to\_decode = TRUE$
        grd2 : $bit2 = FALSE$
        grd3 : $level = 2$
    then
        act1 : $string\_to\_decode := FALSE$
        act2 : $decoded\_char := A$
    end
```

and those which represent intermediary nodes and refine `skip`. For instance,

```
Event  decode_0_1 ≙
Status  convergent
    when
        grd1 : $string\_to\_decode = TRUE$
```

grd2 : $bit0 = TRUE$

grd3 : $level = 0$

then

act1 : $level := 1$

end

The tree is therefore represented by a total of 8 decode events, `decode_0_0, decode_0_1, decode_1_0, decode_1_1, decode_2_0, decode_2_1, decode_3_0 and decode_3_1`.

The following invariants link the original `encoded_char` to the final `decoded_char`:-

INVARIANTS

inv1 : $decoded\_char \in Alphabet$

inv2 : $level \in 0 \mathinner{..} 3$

inv3 : $string\_to\_decode = FALSE \land encoded\_char = E \land level = 0 \Rightarrow decoded\_char = E$

inv4 : $string\_to\_decode = FALSE \land encoded\_char = O \land level = 1 \Rightarrow decoded\_char = O$

inv5 : $string\_to\_decode = FALSE \land encoded\_char = A \land level = 2 \Rightarrow decoded\_char = A$

inv6 : $string\_to\_decode = FALSE \land encoded\_char = U \land level = 3 \Rightarrow decoded\_char = U$

inv7 : $string\_to\_decode = FALSE \land encoded\_char = I \land level = 3 \Rightarrow decoded\_char = I$

inv8 : $3 - level \geq 0$

All proof obligations associated with the abstract model and its refinements are discharged automatically, as shown in Table 4.1.

	Total no. of proof obligations	Discharged Automatically	Discharged Manually	Not Discharged
Abstract Model	0	0	0	0
First Refinement	8	8	0	0
Second Refinement	35	35	0	0
Third Refinement	94	94	0	0

TABLE 4.1: Huffman Proofs

Through the use of the gluing invariants, it has been proved that the decoded encoded stream produced by the concrete implementation is identical to the original input stream. The EFSM representing the concrete model, shown in Figure 4.4, can now be implemented as a single RTL process. There is, however, potential for pipelining in the Huffman algorithm, as described above. The following section will describe how an abstract hardware specification may be refined into a pipelined implementation comprising two or more RTL processes communicating with shared memory.

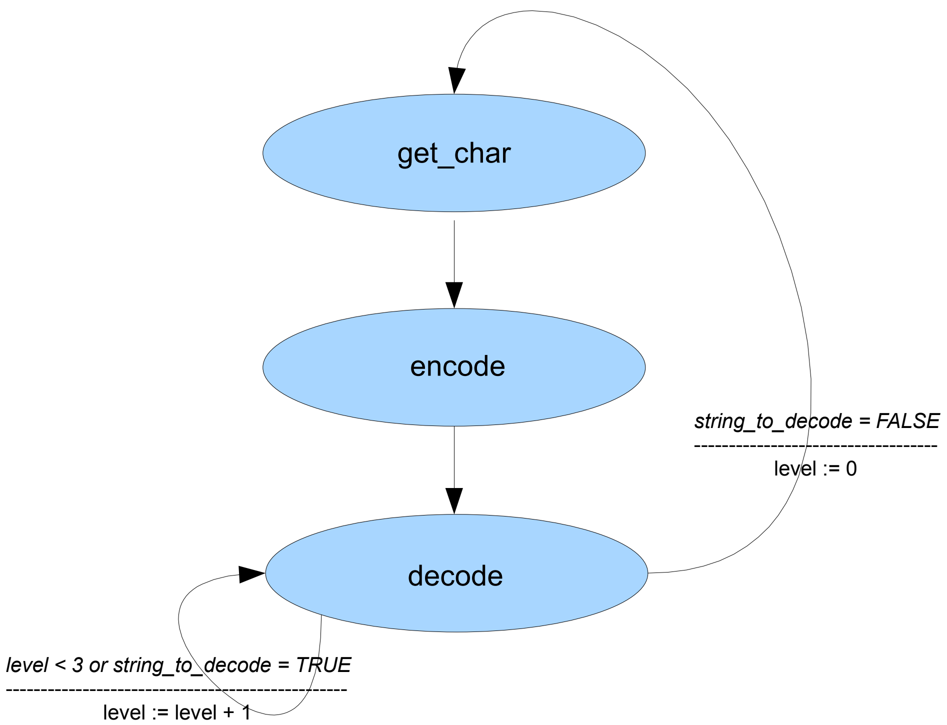


FIGURE 4.4: Huffman EFSM

4.4 Pipelines

Although it is good practice, as exemplified by the SystemC TLM methodology, to employ FIFOs for communication between SoC components, it is often desirable, for performance reasons, to implement synchronous communication within a component with shared variables. If a component is unable to meet its performance targets, then it may be split into sub-components which not only share the workload, but can operate concurrently. This technique is called *pipelining*. A consequence of using a shared variable for communication is that while one sub-component is reading the value of the shared variable, the other sub-component will simultaneously write to the same variable, and this must be taken into account when modelling the pipeline.

4.4.1 Modern SoC Microprocessor Pipelines

In order to achieve high instruction throughput, modern microprocessors can execute several instructions at once in a multi-stage pipeline [Sutherland, 1989]. Early RISC processors typically implemented 5 pipeline stages as exemplified by the DLX microprocessor [Hennessy and Patterson, 2006], but during the 1990's, to keep up with increasing performance needs, more and more stages were introduced. Intel's Pentium III microprocessor has 10 stages and was followed by the Pentium IV with 20. During the design of the Pentium IV however, it was found that in order to keep the global track delays within the required bounds to meet performance targets, it was necessary to manage the latency in the 20-stage pipeline explicitly. Track delay was emerging as the major factor governing pipeline depth [Taylor et al., 2002]. The further shrinking of device sizes and lower power requirements has required a re-evaluation of what constitutes the optimal number of stages [Hartstein and Puzak, 2004] and many modern SoC Microprocessors: MIPS, Virtex PowerPC, ARM9, OR1K and Tensilica Extensa all implement a 5-stage pipeline.

4.4.2 Designing and Verifying an SoC Microprocessor Pipeline

Despite the elegance and simplicity of the core DLX pipeline architecture, any implementation based on this architecture is still subject to design faults which can be extremely difficult to detect using traditional, simulation-based verification techniques. In particular, resource conflicts encountered when several pipelined instructions execute at once can easily be missed. These resource conflicts result from *structural*, *data* and *control* hazards [Hennessy and Patterson, 2006]. What is needed is a suitable representation and a systematic and rigorous methodology for the design of an SoC microprocessor which supports architectural exploration of the trade-offs incurred by the different ways of dealing with pipeline hazards. Ideally such a methodology would fit well with existing synchronous design methodologies and provide a seamless link with the lower level representations required for chip manufacture.

4.4.3 A Pipeline Example: Counting Playing Cards

In this example, an abstract Event-B model is developed to represent the specification of a card counting game. Then, to exploit the potential concurrency inherent in the specification, the abstract model is refined to produce a concrete model which implements a two-stage pipeline. In the development of the refinement it will be shown that the simultaneous access of shared variables must be managed correctly. Appropriate gluing invariants are introduced to prove that the pipelined implementation meets its abstract specification.

Informal Specification

Each card in the pack is assigned a positive integer value. As cards are taken successively from the pack, their values are accumulated and a count kept of the number of cards taken. If the count reaches a predefined limit, or if the accumulated total meets or exceeds a predefined maximum value, the session terminates and a fresh session begins.

The Abstract Model

The Context defines the maximum accumulated total, the maximum card count and the maximum card value.

CONTEXT PIPE21C

CONSTANTS

`MaxTotal`, `MaxCardValue`, `MaxCards`

AXIOMS

`axm1` : $MaxTotal \in \mathbb{N}_1$

`axm2` : $MaxCardValue \in \mathbb{N}_1$

`axm3` : $MaxCardValue < MaxTotal$

`axm4` : $MaxCards \in \mathbb{N}_1$

END

Two variables *count* and *total* model the number of cards taken and the accumulated card values respectively.

INVARIANTS

inv5 : $count \leq MaxCards$

inv6 : $total < MaxTotal + MaxCardValue$

The event *Accumulate* takes as a parameter the card value and updates *count* and *total* appropriately. (The guard *grd3* is independent of *cardval* and different cards may have the same value.)

```
Event  Accumulate ≙
    any
        cardval
    where
        grd1 : $cardval \in 1 \, .. \, MaxCardValue$
        grd2 : $count < MaxCards$
        grd3 : $total < MaxTotal$
    then
        act1 : $total := total + cardval$
        act2 : $count := count + 1$
    end
```

The event *Reset* zeroes *count* and *total* when the terminating conditions are satisfied.

```
Event  Reset ≙
    when
        grd1 : $count = MaxCards \lor total \geq MaxTotal$
    then
        act1 : $count := 0$
        act2 : $total := 0$
    end
```

Although not in general a requirement of Event-B modelling, it is a requirement of this particular system that the model is free from deadlock. At least one of the events *Accumulate* and *Reset* must always be enabled. This is achieved by introducing a theorem to represent the disjunction of the guards of the 2 events.

thm1 : $(count < MaxCards \wedge total < MaxTotal) \vee count = MaxCards \vee total \geq MaxTotal$

This gives rise to a proof obligation: the theorem must follow from the invariants and axioms.

In addition, although not a general Event-B modelling requirement, in this system the enabling of *Accumulate* and *Reset* should be mutually exclusive. The conjunction of the guards should never be true.

thm2 : $\neg((count < MaxCards \wedge total < MaxTotal) \wedge (count = MaxCards \vee total \geq MaxTotal))$

The behaviour of the abstract model may be represented by the Finite State Machine (FSM), shown in Table 4.2, where *count* and *total* represent the state variables and the abstract events *Accumulate* and *Reset* represent the transitions.

Current State		**Actions**	**Next State**	
count	total		count	total
<MaxCards	<MaxTotal	*Accumulate*	count+1	total+value
=MaxCards	$*$	*Reset*	0	0
$*$	≥MaxTotal	*Reset*	0	0

Table 4.2: Abstract Model Events

An implementation could now be derived from this abstract model in which a card value is generated and added to the total in a single clock cycle. Each evaluation of *Accumulate* or *Reset* therefore represents a clock cycle in the implementation. Although this implementation would meet its functional specification, it may be found not to meet its performance targets in terms of speed or power consumption. The complexity of the implementation could mean that either an operation could not be completed within the required clock period, or that doing so would generate too much heat. Event-B refinement is therefore used to explore the viability of a pipelined solution.

The Refined Model

The overall task is split in two. The first sub-task generates the card value from a pack of cards and the second sub-task adds this value to the total and checks the termination conditions. A pipeline register is introduced comprising two variables. The pipeline is shown in Figure 4.5.

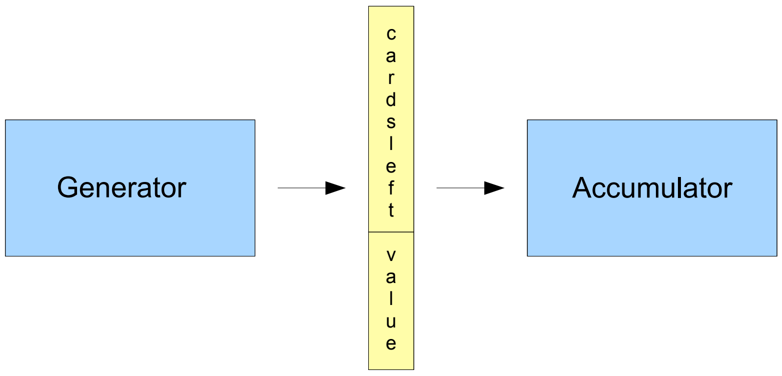


FIGURE 4.5: The Pipeline

The variable *value* represents the card value generated and the variable *cardsleft* represents the number of cards left in the current pack. The new event *Generate* is introduced which writes to *value* and the guards of the abstract event *Accumulate* are strengthened so that it reads from *value*. A pack of cards is available in the initial state, but when this pack runs out, a new pack must be fetched. The event *GetNewPack* is introduced to do this.

```
Event  Generate ≙
    any
        cardval
    where
        grd1 : $cardval \in 1 \,..\, MaxCardValue$
        grd2 : $cardsleft > 0$
    then
        act1 : $value := cardval$
        act2 : $cardsleft := cardsleft - 1$
    end
Event  GetNewPack ≙
    where
        grd1 : $cardsleft = 0$
    then
        act1 : $cardsleft := PackSize$
    end
```

Event *Accumulate* $\widehat{=}$

refines *Accumulate*

```
    any
        cardval
    where
        grd1 : $cardval = value$
        grd2 : $count < MaxCards$
        grd3 : $total < MaxTotal$
    then
        act1 : $total := total + cardval$
        act2 : $count := count + 1$
    end
```

The pipeline, however, will not operate as desired. Initially, only the *Generate* event is enabled, but an evaluation of this event results in both *Generate* and *Accumulate* being enabled simultaneously. If *Generate* is chosen, non-deterministically, for evaluation, the initial value generated will be incorrectly overwritten. If, however, *Accumulate* is chosen for evaluation and then chosen a second time, the original generated value will be incorrectly used twice. Using a FIFO instead of a variable for communication would ensure the desired operation, but the overhead of incorporating the FIFO and its control logic in small components is prohibitive. There are therefore two alternative architectures that can be considered. A single state variable could be used to manage the enabling of the *Generate* and *Accumulate* events and ensure the desired ordering. An implementation of this architecture would require two clock cycles for each operation and for very low power applications this may be acceptable. In general, however, it desirable to exploit the simultaneity offered by a pipelined architecture to maximise throughput. An example of the desired behaviour is shown in Figure 4.6.

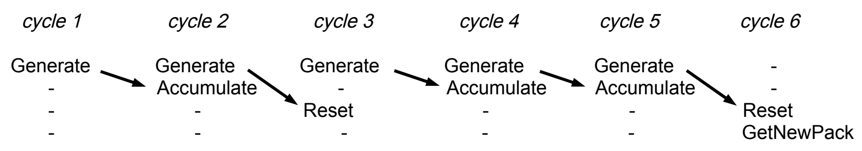


FIGURE 4.6: Pipeline Execution

To model this behaviour, it is necessary to introduce further events into this refinement which represent a merge of the events in the two pipeline stages. How the events in the first stage must be merged with those in the second is dependent on the value of the pipeline control variable *cardsleft*. *Generate* is enabled when *cardsleft* is greater than zero and *GetNewPack* is enabled when *cardsleft* is equal to zero. Either *Accumulate* or *Reset* is enabled simultaneously depending on the values of *count* and *total*. The exception is when a new pack is presented and *cardsleft* equals *PackSize*. In this case only *Generate* is enabled. The merged events and the conditions under which they are enabled, as represented by the values of *cardsleft*, *count* and *total*, is shown in Table 4.3. The table also includes the states of these variables that result from the evaluation of the merged events. An entry "$*$" indicates *"don't care"* and "$-$" indicates *"unchanged"*.

Consider an example of merging events. The merged event *GenerateAccumulate*, shown in the second row of Table 4.3, both reads and writes the variable *value*. Event-B semantics ensures that value of the variable read is always the value prior to update, and this matches precisely the semantics of a hardware implementation.

```
Event  GenerateAccumulate ≙
refines  Accumulate
    any
        gcardval, cardval
    where
        grd1 : gcardval ∈ 1 .. MaxCardValue
        grd2 : cardval = value
        grd3 : total < MaxTotal
        grd4 : count < MaxCards
        grd5 : cardsleft > 0
        grd6 : cardsleft < PackSize
    then
        act1 : value := gcardval
        act2 : total := total + cardval
        act3 : count := count + 1
        act4 : cardsleft := cardsleft − 1
    end
```

Current State			**Actions**	**Next State**		
cardsleft	count	total		cardsleft	count	total
PackSize	$*$	$*$	*Generate*	PackSize-1	–	–
1..PackSize-1	<MaxCards	<MaxTotal	*Generate* *Accumulate*	cardsleft-1	count+1	total+value
1..PackSize-1	=MaxCards	$*$	*Generate* *Reset*	cardsleft-1	0	0
1..PackSize-1	$*$	≥MaxTotal	*Generate* *Reset*	cardsleft-1	0	0
0	<MaxCards	<MaxTotal	*GetNewPack* *Accumulate*	PackSize	count+1	total+value
0	=MaxCards	$*$	*GetNewPack* *Reset*	PackSize	0	0
0	$*$	≥MaxTotal	*GetNewPack* *Reset*	PackSize	0	0

TABLE 4.3: Refined Model Merged Events

	Total no. of proof obligations	Discharged Automatically	Discharged Manually	Not Discharged
Abstract Model	11	11	0	0
First Refinement	13	13	0	0

TABLE 4.4: Pipeline Proofs

All proof obligations associated with the abstract model and its refinement, which includes an event for each case identified in Table 4.3, are discharged automatically by the Rodin tool, as shown in Table 4.4.

We modelled simultaneous execution of pipeline stages as a single event in Event-B. We then identified the different combinations of cases for the stages (Table 4.3). In the next section we outline a general scheme for managing event simultaneity in pipelines.

4.4.4 Event Simultaneity in Pipelines

Consider a 2-stage pipeline with the first stage represented by the event E$_1$ and the second by the event E$_2$, and each event comprising a guard g$_i$ and an action A$_i$ (E$_i$ is g$_i$$A_i$). If g_1 and g_2 are both true, A_1 and A_2 are executed simultaneously. If either of g_1 and g_2 is true (exclusively) then one of A_1 and A_2 is executed. If both guards are false then neither action is executed. The required cases of behaviour are shown in Table 4.5.

g_1	$\wedge$	g_2	$\longrightarrow$	$A_1 \parallel A_2$
g_1	$\wedge$	$\neg g_2$	$\longrightarrow$	A_1
$\neg g_1$	$\wedge$	g_2	$\longrightarrow$	A_2
$\neg g_1$	$\wedge$	$\neg g_2$	$\longrightarrow$	–

TABLE 4.5: 2-stage 2-event pipeline

If a particular conjunction of two guards always evaluates to false, then the associated action will never occur. For instance, if $g_1 \wedge \neg g_2$ can never be true, then action A_1 can never be executed on its own.

More generally, a pipeline stage may be represented by a set of events. For instance, in a 2-stage pipeline where each stage, i, is represented by two events E_{i1} $(g_{i1} A_{i1})$and E_{i2} $(g_{i2} A_{i2})$ then, where N_{gi} is $\neg(g_{i1} \vee g_{i2})$, the required behaviour is shown inTable 4.6. Again, not all combinations may be possible.

g_{11}	$\wedge$	g_{21}	$\longrightarrow$	$A_{11} \parallel A_{21}$
g_{11}	$\wedge$	g_{22}	$\longrightarrow$	$A_{11} \parallel A_{22}$
g_{11}	$\wedge$	N_{g2}	$\longrightarrow$	A_{11}
g_{12}	$\wedge$	g_{21}	$\longrightarrow$	$A_{12} \parallel A_{21}$
g_{12}	$\wedge$	g_{22}	$\longrightarrow$	$A_{12} \parallel A_{22}$
g_{12}	$\wedge$	N_{g2}	$\longrightarrow$	A_{12}
N_{g1}	$\wedge$	g_{21}	$\longrightarrow$	A_{21}
N_{g1}	$\wedge$	g_{22}	$\longrightarrow$	A_{22}
N_{g1}	$\wedge$	N_{g2}	$\longrightarrow$	–

Table 4.6: 2-stage 4-event pipeline

For a 2-stage pipeline, where the stages are represented by m and n events respectively, the combinations are shown in Table 4.7

g_{11}	$\wedge$	g_{21}	$\longrightarrow$	$A_{11} \parallel A_{21}$
.	.	.	.	.
g_{11}	$\wedge$	g_{2n}	$\longrightarrow$	$A_{11} \parallel A_{2n}$
g_{11}	$\wedge$	N_{g2}	$\longrightarrow$	A_{11}
.	.	.	.	.
.	.	.	.	.
g_{1m}	$\wedge$	g_{21}	$\longrightarrow$	$A_{1m} \parallel A_{21}$
.	.	.	.	.
g_{1m}	$\wedge$	g_{2n}	$\longrightarrow$	$A_{1m} \parallel A_{2n}$
g_{1m}	$\wedge$	N_{g2}	$\longrightarrow$	A_{1m}
N_{g1}	$\wedge$	g_{21}	$\longrightarrow$	A_{21}
.	.	.	.	.
N_{g1}	$\wedge$	g_{2n}	$\longrightarrow$	A_{2n}
N_{g1}	$\wedge$	N_{g2}	$\longrightarrow$	–

Table 4.7: 2-stage m+n-event pipeline

Each row in the table, apart from the last, corresponds to a combined event that will need to be represented in the machine for the pipeline to be modelled correctly. Combined events whose guards always evaluate to *false*, however, can be excluded. Since the guards of the events representing a pipeline stage need not be mutually exclusive, more than one combined event may be enabled for a given set of variable values.

4.4.5 Measuring Pipeline Complexity at the Specification Level

The number of simultaneous events that need to be considered is exponential with the number of pipeline stages. For a pipeline stage S_i represented by n_i events, there is a maximum of $n_i + 1$ ways that these events can be combined with the events of other stages, since the absence of an event must also be considered. Thus, for a pipeline with p stages the maximum number of combined events is

$$(n_1 + 1) \times (n_2 + 1) \times \cdots \times (n_p + 1)$$

In practice, not all combinations are usually possible, but nevertheless the number of combined events increases rapidly with the number of stages in a component pipeline. For instance, a 7-stage pipeline with 2 events per stage could have up to 2187 combined events. The number of combined events, however, is a direct measure of the complexity of the component and represents the number of cases that must be considered for full component verification. Valuable feedback to the designer on the precise number of feasible simultaneous pipeline event combinations can therefore be provided during micro-architectural exploration at the specification level.

4.4.6 Pipeline Feedback

In this section we consider the issue of pipeline feedback. Consider the example from [Shankar, 1998], a pipelined version of which is shown in Figure 4.7. The component P guarantees that its output is *even* if both its inputs are *odd*. The component Q guarantees that its outputs are both *odd* if its input is even. It is required that three invariants are preserved by the composition of P and Q as a pipeline.

$$\boxed{\text{Invariant 1: } x \text{ is always } \textit{odd}}$$

$$\boxed{\text{Invariant 2: } y \text{ is always } \textit{odd}}$$

$$\boxed{\text{Invariant 3: } z \text{ is always } \textit{even}}$$

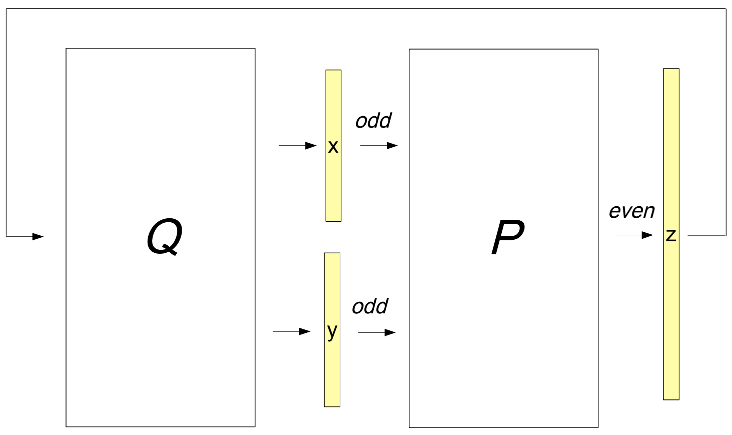


FIGURE 4.7: Even Number Generator

The difficulty with this example arises from the fact that in composing P and Q the output of P is *fed back* to the input of Q.

For a pipeline section with no feedback, as shown in Figure 4.8, where *E1* and *E2* are events and *V1* and *V2* are shared registers, then *E2* followed by *E1* *(E2;E1)* is equivalent to *E1* and *E2* occuring simultaneously $(E1||E2)$. Even if *V2* depends on *V1*, by evaluating *E2* before *E1*, *E2* sees the previous value of *V1* which matches precisely the hardware semantics for simultaneous register read and write.

For a pipeline section with feedback, however, as shown in Figure 4.9, there is *no interleaving* that represents $(E1||E2)$. It is therefore necessary to merge the events *E1* and *E2* into a single, combined event *E1E2* to model correctly the simultaneity of the pipeline section. For pipeline sections with feedback, the invariants must be proved for the complete, merged section. Once these invariants have been proved, however, the merged pipeline section may be *decomposed*, using the techniques described in Section 2.9 on Page 17 and illustrated later in this section, into the events that represent each of its constituent stages. By the use of Event-B decomposition, the exponential explosion of combined pipeline events with pipeline length can be controlled.

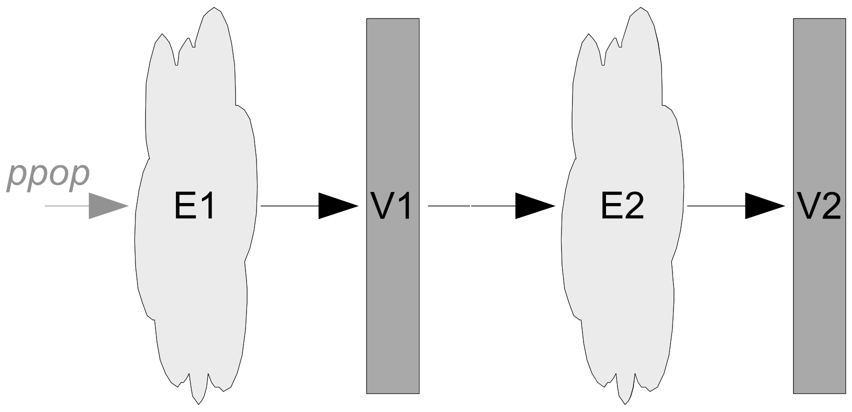


FIGURE 4.8: Pipeline without Feedback

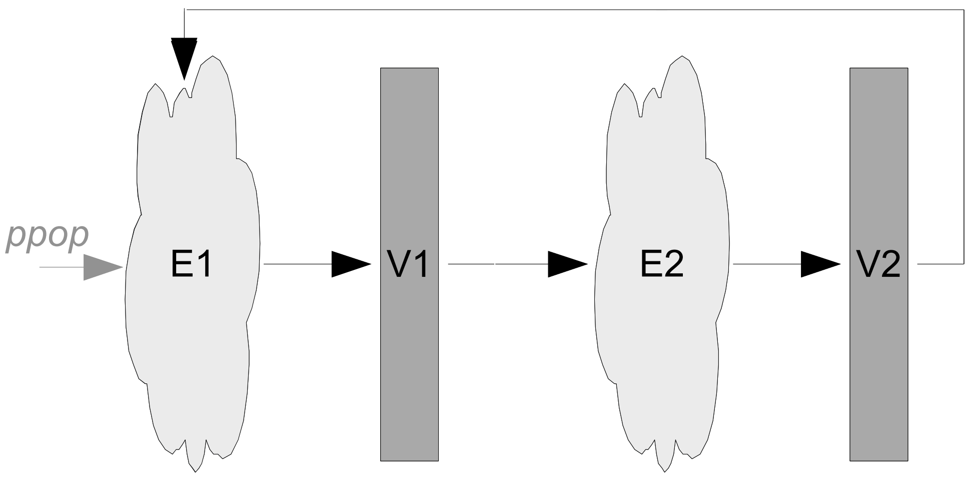


FIGURE 4.9: Pipeline with Feedback

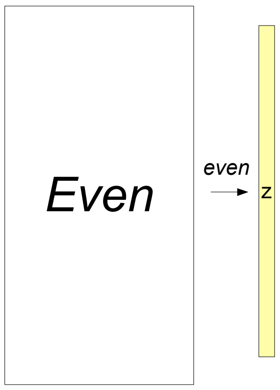


FIGURE 4.10: Abstract Model

The Abstract Model

At the abstract level, the Even Number Generator may be considered as a black box that writes an even number to a register z, as shown in Figure 4.10, preserves the invariant

inv2 : $z\ mod\ 2 = 0$

and can be described with the single event

```
Event  $Even \mathrel{\widehat{=}}$
    any
        $v$
    where
        grd1 : $v \in \mathbb{N}$
        grd2 : $v\ mod\ 2 = 0$
    then
        act1 : $z := v$
    end
```

This abstraction captures the important property of this system: register z may be updated, but always with an even number.

The First Refinement

The simultaneous behaviour of the complete pipeline with feedback can now be modelled. The micro-architecture of this refinement is shown in Figure 4.11.

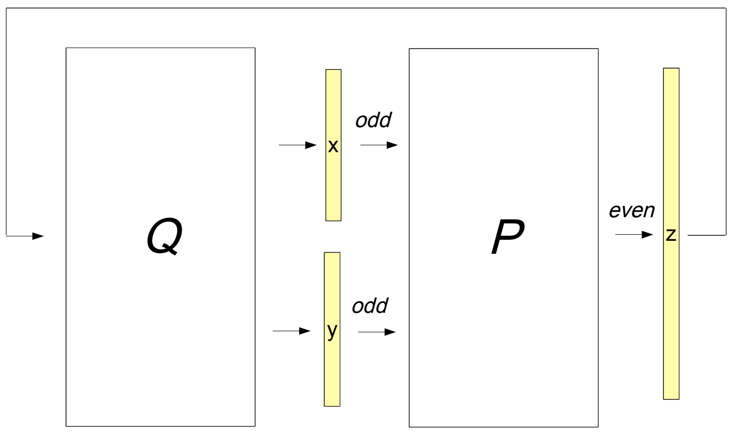


FIGURE 4.11: First Refinement

Two registers x and y are introduced together with the following invariants

inv1 : $x \in \mathbb{N}$

inv2 : $x \ mod \ 2 \neq 0$

inv3 : $y \in \mathbb{N}$

inv4 : $y \ mod \ 2 \neq 0$

inv5 : $(x + y) \ mod \ 2 = 0$

and the parameter v is bound to the registers x and y with the *witness*

v : $v = x + y$

which *instantiates* the parameter of the abstract model with a concrete value, as described in Section 2.9 on Page 19.

The register z now takes the value $x + y$ while *simultaneously* x and y both take the value $z + 1$ as shown in the actions

```
act1 : $z := x + y$

act2 : $x := z + 1$

act3 : $y := z + 1$
```

The event *Evencmp* represents the composed behaviour $P||Q$.

```
**Event** $Evencmp \mathrel{\widehat{=}}$
**refines** $Even$
    **begin**
        **with**
            v : $v = x + y$
        act1 : $z := x + y$
        act2 : $x := z + 1$
        act3 : $y := z + 1$
    **end**
**END**
```

All proof obligations are discharged automatically by the Rodin tool, showing that the composed concrete implementation is a correct refinement of the abstract model.

4.4.7 An Alternative Compositional Approach to Pipeline Refinement

Recall the abstract machine architecture of Figure 4.12

and the event of the abstract model

```
**Event** $Even \mathrel{\widehat{=}}$
    **any**
        $v$
    **where**
        grd1 : $v \in \mathbb{N}$
        grd2 : $v \; mod \; 2 = 0$
    **then**
        act1 : $z := v$
    **end**
```

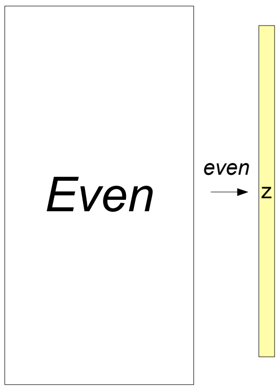


Figure 4.12: Abstract Model

The First Refinement

Now consider an Event-B machine with a single event *Evenp* which refines the abstract machine and models the component P of Figure 4.11 together with an abstract representation of the environment in which P must be placed if it is to deliver its specified behaviour.

Event *Evenp* $\widehat{=}$

refines *Even*

any

$pin1, pin2$

where

grd1 : $pin1 \in \mathbb{N}$

grd2 : $pin1 \ mod \ 2 \neq 0$

grd3 : $pin2 \in \mathbb{N}$

grd4 : $pin2 \ mod \ 2 \neq 0$

grd5 : $(pin1 + pin2) \ mod \ 2 = 0$

with

v : $\mathtt{v} = \mathtt{pin1} + \mathtt{pin2}$

then

act1 : $z := pin1 + pin2$

end

The micro-architecture is shown in Figure 4.13.

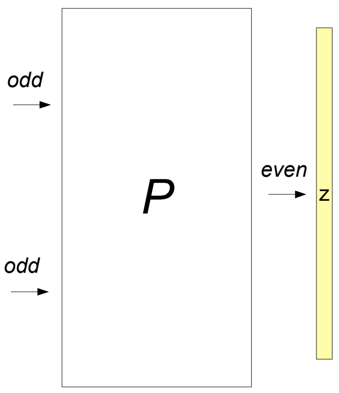


FIGURE 4.13: First Refinement

There is a requirement on the *environment* in which component P is used to ensure that both inputs are *odd*, represented by the guards

grd1 : $pin1 \in \mathbb{N}$

grd2 : $pin1 \ mod \ 2 \neq 0$

grd3 : $pin2 \in \mathbb{N}$

grd4 : $pin2 \ mod \ 2 \neq 0$

If these conditions are met, then the model takes its two inputs *pin1* and *pin2*, adds them together and writes the result to the output register z. The parameter v is replaced by the parameters *pin1* and *pin2* using the witness

v : $\mathtt{v} = \mathtt{pin1} + \mathtt{pin2}$

Thus, if P's environment fulfills its requirement to ensure that both inputs are *odd* then P will produce an even number on its output.

All proof obligations are discharged, showing that component P, together with its parameterised environment, is a correct refinement of the abstract model.

The Second Refinement

Now consider a parameterised representation of the component Q of Figure 4.11 and its environment that takes a single input, increments the value on that input and writes the result to two output registers x and y. There is a requirement on the *environment* in which component Q is used to ensure that the input is *even*. If the environment fulfills this requirement then Q will produce an odd number on each of its outputs.

The micro-architecture of the component is shown in Figure 4.14.

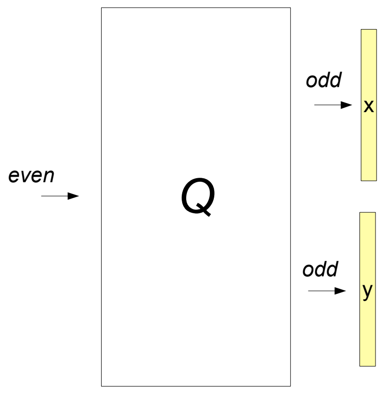


Figure 4.14: Component Q and its Environment

and its single event is as follows.

```
Event   Oddp ≙
    any
        pin
    where
        grd1 :  $pin \in \mathbb{N}$
        grd2 :  $pin\ mod\ 2 = 0$
    then
        act1 :  $x := pin + 1$
        act2 :  $y := pin + 1$
    end
```

In this refinement, the machine representing the component P and its environment is composed, using the Rodin composition plug-in, with the machine from the first refinement (representing Q and its environment) to form a machine with a single event PQ which represents the composition of the events *Evenp* and *Oddp* and refines *Evenp*.

```
Event  PQ ≙
refines  Evenp
    any
        pin, pin1, pin2
    where
        grd1 : $pin \in \mathbb{N}$
        grd2 : $pin\ mod\ 2 = 0$
        grd3 : $pin1 \in \mathbb{N}$
        grd4 : $pin1\ mod\ 2 \neq 0$
        grd5 : $pin2 \in \mathbb{N}$
        grd6 : $pin2\ mod\ 2 \neq 0$
        grd7 : $(pin1 + pin2)\ mod\ 2 = 0$
    then
        act1 : $z := pin1 + pin2$
        act2 : $x := pin + 1$
        act3 : $y := pin + 1$
    end
```

All proof obligations are discharged automatically by the Rodin tool, showing that the composed machine is a correct refinement of the abstract machine. At this stage, however, no communication has been established between the components because the parameters have not yet been bound to the registers, as shown in Figure 4.15.

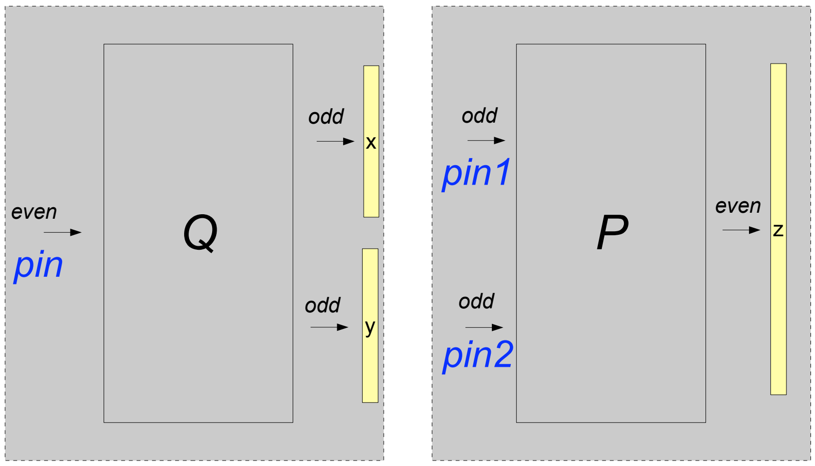


FIGURE 4.15: Initial Component Composition

The Third Refinement

Now, the *parameterised* inputs and outputs of the composed machine can take the values from the appropriate registers using witnesses. The parameter *pin* is bound to the register z and *pin1* and *pin2* are bound to the registers x and y respectively. The concrete, combined event is as follows.

```
**Event** $PQ \widehat{=}$
**refines** $PQ$
    **begin**
        **with**
            pin : pin = z
            pin1 : pin1 = x
            pin2 : pin2 = y
        act1 : $z := x + y$
        act2 : $x := z + 1$
        act3 : $y := z + 1$
    **end**
**END**
```

Shared register communication has been established, all proof obligations are discharged automatically, showing that this is a correct refinement of the abstract model and the final, concrete micro-architecture of Figure 4.16 is achieved.

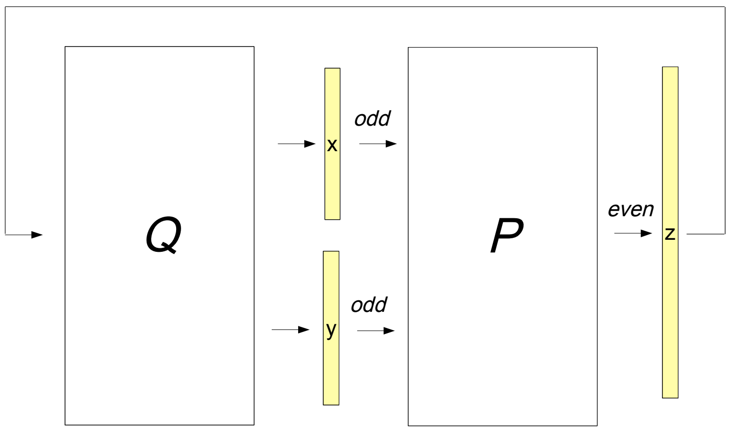


Figure 4.16: Pipelined Implementation

The event refinement steps used can be summarised as follows.

Even *(Abstract event)*

$\sqsubseteq$

Evenp *(Component P with its Environment)*

$\sqsubseteq$

Oddp || Evenp *(P with its Environment || Q with its Environment)*

$\sqsubseteq$

PQ

4.4.8 Pipeline Decomposition

The last refinement is a concrete representation of the simultaneous behaviour of the 2-stage pipeline, with each evaluation of the composed event *Evencmp* representing a clock cycle in the hardware pipeline. The model can now be decomposed formally using Event-B *shared event decomposition*, as described in Section 2.9 on Page 18, into two models, to represent the two hardware *processes* that constitute the two stages of the pipeline, communicating with the shared registers x, y and z. The two models each comprise a single event, P and Q respectively,

```
Event  P ≙
    begin
        act1 : $z := x + y$
    end
Event  Q ≙
    begin
        act1 : $x := z + 1$
        act2 : $y := z + 1$
    end
```

and have a direct correspondence with the two Verilog RTL processes shown in Table 4.8

always @ (posedge clk)
begin
$z <= x + y$
end
	always @ (posedge clk)
	begin
	$x <= z + 1$
	$y <= z + 1$
	end

TABLE 4.8: Verilog RTL Processes

At each clock cycle, z takes the value $x + y$ while, simultaneously, x and y takes the value $z + 1$. The update semantics of the Event-B and the Verilog are identical since in each case the registers are updated with the *previous* values of the registers referred to on the right-hand side of the assignments; the *non-blocking* nature of the assignments used in the Verilog ensures that all updates happen in parallel and match the atomic nature of the Event-B actions.

The three key invariants from the Event-B model,

inv2 : $z \mathit{~mod~} 2 = 0$

inv3 : $x \mathit{~mod~} 2 \neq 0$

inv4 : $y \mathit{~mod~} 2 \neq 0$

can also be translated directly into assertions in the hardware property specification language PSL to accompany the RTL.

inv2 : $assert\ always((z\ \%\ 2) == 0)$

inv3 : $assert\ always((x\ \%\ 2) \,!= 0)$

inv4 : $assert\ always((y\ \%\ 2) \,!= 0)$

4.4.9 Generalising the Approach to Pipeline Verification with Event-B

This example suggests a general approach to pipeline modelling and verification with Event-B, which will be explored in detail in subsequent chapters and is a key contribution of our work.

The method begins with an abstract Event-B model representing the specification of the required behaviour. The abstract specification represents a high-level view of the hardware which executes in a single cycle. A refinement of the abstract model is then introduced which represents the behaviour as a two-stage pipeline; the second stage represents a concrete stage of the final implementation while the first stage is an abstraction of the rest of the pipeline. The choice of which stage to make concrete depends on the nature of the abstract specification. For instance, the specification may be defined in terms of the effect that an operation has on the pipeline registers or the program counter and this will determine how the refinement proceeds, as explained in the next chapter.

The two stages communicate via shared registers and gluing invariants will need to be introduced to prove that the two-stage pipeline implements the abstract specification. This model executes in two cycles but with overlapping execution of the two stages which, coupled with any feedback in the pipeline, can lead to *data* or *control* hazards. It will be shown in the next chapter how the proof obligations generated by the Rodin tool can be used to discover the appropriate invariants to deal with the hazards.

This two-stage model is then further refined into a three-stage pipeline with now two of the stages representing the final implementation while the third is an abstraction of the rest of the pipeline. Again, gluing invariants are introduced to prove that the three-stage pipeline of the refinement implements the more abstract two-stage model and to deal with any further pipeline hazards. This process continues until all pipeline stages within a pipeline feedback loop have a concrete implementation.

Once the invariants have been proved, which shows that the overlapping execution within the feedback loop has been implemented correctly, shared event decomposition is used to decompose the pipeline model into a set of models, one for every stage of the pipeline. Models representing the stages outside the feedback loop can then be introduced. Each pipeline stage model represents a hardware process of the final pipeline implementation, and the invariants represent the properties of that pipeline.

The Event-B models representing each pipeline stage can then be mapped directly to a Bluespec, CAL or RTL process representation and the invariants mapped to a property specification language such as PSL.

In the next chapter we will apply this general approach to a typical SoC Microprocessor and also show how, by decomposing the problem, each instruction of the ISA can be modeled and refined individually before arriving at a concrete implementation of the pipeline.

Chapter 5

Developing an SoC Pipelined Microprocessor Model

The contribution described in this chapter is the development of the method outlined at the end of Chapter 4 so that an abstract specification of an SoC microprocessor can be refined formally to derive a concrete implementation of the microprocessor that has a direct correspondence to its HDL description. The method supports the exploration and development of the pipelined microarchitecture, where the abstract machine represents directly an instruction from the ISA that specifes the effect that the instruction has on the microprocessor register file and/or the program counter. Refinement is then used systematically to derive a concrete, pipelined execution of that instruction that is proved to implement correctly its abstract ISA specification. When all the instructions have been modelled, formal re-composition of the instruction contributions to each stage is then used to derive the full, pipelined ISA implementation. Microarchitectural considerations are raised to the specification level and design choices can be verified much earlier in the flow.

The motivation for this work is to demonstrate an approach that uses Event-B and refinement to develop and verify a concrete implementation of a typical SoC microprocessor from its ISA specification. A TRS-style description of the DLX instruction set is presented, which is a natural mapping from the DLX specification taken from [Hennessy and Patterson, 2006] and shown in Table 5.1, to show that an abstract specification of the DLX pipeline can be developed in Event-B, that this specification can be refined systematically to a point where different architectural trade-offs can be explored to deal with pipeline hazards and that finally a concrete specification suitable for synthesis can be produced that has been verified formally to meet its abstract specification.

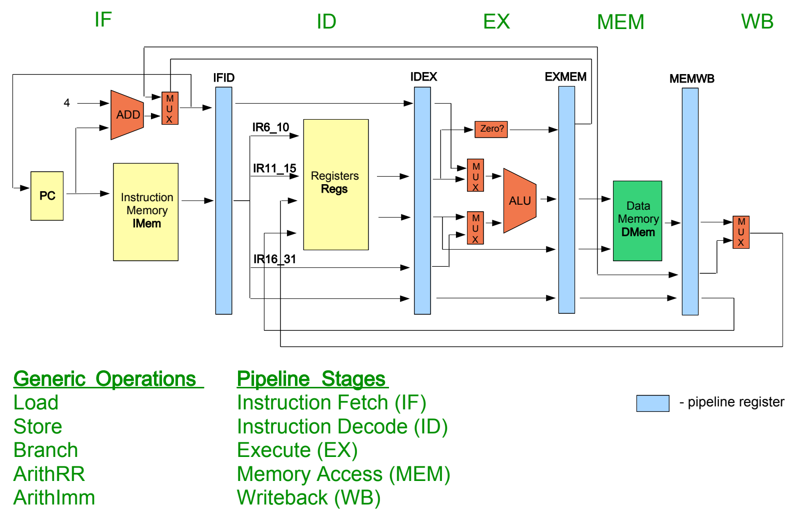


FIGURE 5.1: DLX 5-stage Pipeline

5.1 Modelling DLX with Event-B

DLX has 5 generic instructions: Load, Store, Branch, ALU (register/register) and ALU (register/immediate) which are executed over 5 pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execute(EX), Memory Access (MEM) and Writeback (WB) as shown in Figure 5.1.

5.1.1 Instruction Fetch (IF)

The IF stage gets the instruction address from the program counter (PC) register and fetches the instruction from the *Instruction Memory* (IMem). The instruction is written to the pipeline register IFID. If a branch instruction has been encountered in the pipeline, then PC is updated to reflect the new instruction address. Otherwise PC is incremented to point to the next instruction in IMem. The value of PC is also written to the IFID pipeline register.

5.1.2 Instruction Decode (ID)

DLX has 3 32-bit instruction types, where the instruction (IR) fields and offsets differ according to type.

I-type:

0..5	6..10	11..15	16..31
Opcode	rs1	rd	Immediate

where *rs1* ($IR_{6..10}$) is the *source register*, *rd* ($IR_{11..15}$) is the *destination register* and *Immediate* ($IR_{16..31}$) is a 16-bit source data value stored in the instruction (used for *register/immediate* arithmetic operations and for conditional branches),

R-type:

0..5	6..10	11..15	16..20	21..31
Opcode	rs1	rs2	rd	func

where *rs1* ($IR_{6..10}$) and *rs2* ($IR_{11..15}$) are *source registers*, *rd* ($IR_{16..20}$) is the *destination register* and *func* ($IR_{21..31}$) is the ALU operation to be performed, and

J-type:

0..5	6..31
Opcode	Branch Offset

used for unconditional jumps.

The values at offsets $IR_{6..10}$ and $IR_{11..15}$,read from the *IFID* pipeline register, are used as indices into the register file *Regs* and the resulting values are stored in the registers *A* and *B* respectively, which form part of the *IDEX* pipeline register. The value at offset $IR_{16..31}$ is written directly to the register*Imm* in the *IDEX* pipeline register, together with the instruction itself and PC value from the previous stage.

5.1.3 Execute (EX)

This stage performs 3 different operations, depending on instruction type.

For ALU operations, the values of *A* and (*B* or *Imm*) are passed to the ALU and the result written to the $ALUoutput$ register.

For *Load* and *Store* operations, the values of *A* and *Imm* are added together by the ALU to generate the memory address, which is also stored in $ALUoutput$. The value of *B* is written directly to the *EXMEM* pipeline register.

For conditional *Branch* instructions, the PC and *Imm* values are added by the ALU to generate the branch offset, also stored in $ALUoutput$. In addition, the *cond* register is set to "1" if the boolean operation (branch if equal/not equal to "0") on the value of register *A* determines that the branch will occur. (In all other cases, *cond* is set to "0".)

5.1.4 Memory Access (MEM)

The address line of the data memory *DMem* takes the value of the register *ALUoutput* from the previous stage. In addition, the values of *ALUoutput* and *cond* are fed back to the *Instruction Fetch* stage so that the next value of PC can be determined if there is a branch.

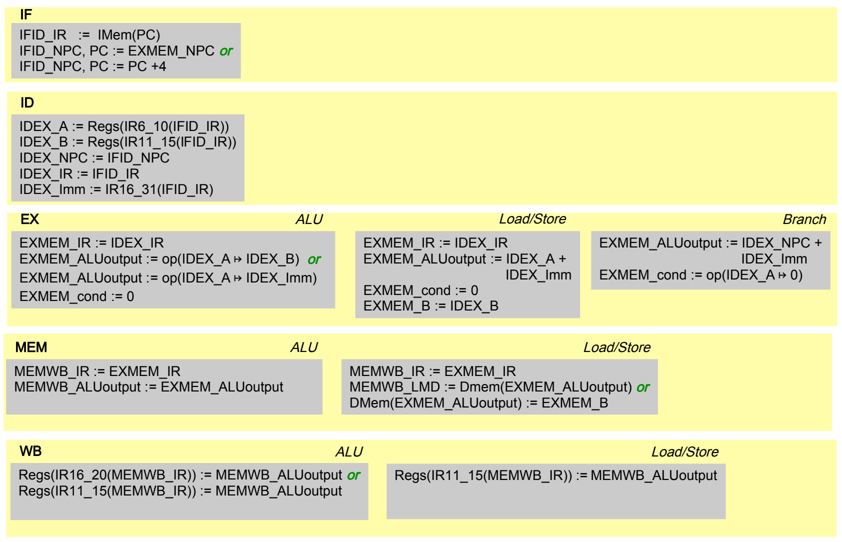
For *Store* operations, the value of *B* from the previous stage is written to the data line of *Dmem*.

For *Load* operations the value on the output line of *DMem* is written to the register *LMD*.

For *ALU* operations, the value of *ALUoutput* is transferred directly to the *MEMWB* pipeline register.

5.1.5 Writeback (WB)

For *ALU* and *Load/Store* operations the value of *ALUoutput* from the previous stage is written to the register file at the location determined by the target register in the instruction.



```
IF
IFID_IR   :=  IMem(PC)
IFID_NPC, PC := EXMEM_NPC or
IFID_NPC, PC := PC +4

ID
IDEX_A := Regs(IR6_10(IFID_IR))
IDEX_B := Regs(IR11_15(IFID_IR))
IDEX_NPC := IFID_NPC
IDEX_IR := IFID_IR
IDEX_Imm := IR16_31(IFID_IR)

EX
ALU
EXMEM_IR := IDEX_IR
EXMEM_ALUoutput := op(IDEX_A ↦ IDEX_B)  or
EXMEM_ALUoutput := op(IDEX_A ↦ IDEX_Imm)
EXMEM_cond := 0

Load/Store
EXMEM_IR := IDEX_IR
EXMEM_ALUoutput := IDEX_A +
                          IDEX_Imm
EXMEM_cond := 0
EXMEM_B := IDEX_B

Branch
EXMEM_ALUoutput := IDEX_NPC +
                        IDEX_Imm
EXMEM_cond := op(IDEX_A ↦ 0)

MEM
ALU
MEMWB_IR := EXMEM_IR
MEMWB_ALUoutput := EXMEM_ALUoutput

Load/Store
MEMWB_IR := EXMEM_IR
MEMWB_LMD := Dmem(EXMEM_ALUoutput) or
DMem(EXMEM_ALUoutput) := EXMEM_B

WB
ALU
Regs(IR16_20(MEMWB_IR)) := MEMWB_ALUoutput or
Regs(IR11_15(MEMWB_IR)) := MEMWB_ALUoutput

Load/Store
Regs(IR11_15(MEMWB_IR)) := MEMWB_ALUoutput
```

FIGURE 5.2: Event-B DLX Instructions

Figure 5.2 shows how the DLX Pipeline Structure as summarised in [Prabhu, 2000] and shown in Table 5.1 can be mapped directly to an Event-B description whilst retaining the style and structure of the original representation.

Stage	ALU instruction	Load or store instruction	Branch instruction
IF	$IF/ID.IR \leftarrow Mem[PC]$; $IF/ID.NPC, PC \leftarrow (if\ EX/MEM.cond\ \{EX/MEM.NPC\}\ else\ \{PC+4\})$;		
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR_{6..10}]$; $ID/EX.B \leftarrow Regs[IF/ID.IR_{11..15}]$; $ID/EX.NPC \leftarrow IF/ID.NPC$; $ID/EX.IR \leftarrow IF/ID.IR$; $ID/EX.Imm \leftarrow (IR_{16})^{16}\#\#IR_{16..31}$;		
EX	$EX/MEM.IR \leftarrow ID/EX.IR$; $EX/MEM.ALUoutput \leftarrow$ $ID/EX.A\ op\ ID/EX.B$; *or* $EX/MEM.ALUoutput \leftarrow$ $ID/EX.A\ op\ ID/EX.Imm$; $EX/MEM.cond \leftarrow 0$;	$EX/MEM.IR \leftarrow ID/EX.IR$; $EX/MEM.ALUoutput \leftarrow$ $ID/EX.A + ID/EX.Imm$; $EX/MEM.cond \leftarrow 0$; $EX/MEM.B \leftarrow ID/EX.B$	$EX/MEM.ALUoutput$ $\leftarrow$ $ID/EX.NPC+$ $ID/EX.Imm$; $EX/MEM.cond \leftarrow$ $(ID/EX.A\ op\ 0)$;
MEM	$MEM/WB.IR \leftarrow$ $EX/MEM.IR$; $MEM/WB.ALUoutput \leftarrow$ $EX/MEM.ALUoutput$;	$MEM/WB.IR \leftarrow$ $EX/MEM.IR$; $MEM/WB.LMD \leftarrow$ $Mem[EX/MEM.ALUoutput]$; *or* $Mem[EX/MEM.ALUoutput] \leftarrow$ $EX/MEM.B$;	
WB	$Regs[MEM/WB.IR_{16..20}] \leftarrow$ $MEM/WB.ALUoutput$; *or* $Regs[MEM/WB.IR_{11..15}] \leftarrow$ $MEM/WB.ALUoutput$;	$Regs[MEM/WB.IR_{11..15}] \leftarrow$ $MEM/WB.LMD$;	

TABLE 5.1: DLX Pipeline Structure

5.2 A General Overview of the Method

Using Event-B, we develop a method where an abstract machine represents directly an instruction from the ISA that specifes the effect that the instruction has on the microprocessor register file or the program counter (PC). Refinement is then used systematically to derive a concrete, pipelined execution of that instruction. At each refinement step the importance is shown of addressing the inherent simultaneity that characterises the pipelined behaviour, the explosion of complexity that can result from this simultaneity and the effect that feedback, which may introduce data or control hazards, has on pipeline construction. The architect can focus on developing and refining the pipeline description and defining the verification conditions which link successive levels of refinement; the necessary proof obligations are generated and discharged in the Event-B development environment, Rodin. Once the instruction operation within a feedback loop has been correctly modelled and proved, formal decomposition is used to derive the contribution of the instruction to the *process* representing each pipeline stage within the feedback loop. The instruction model can then be extended formally to represent the remaining pipeline stages. When all the instructions have been modelled, formal re-composition of the instruction contributions to each stage is then used to derive the full, pipelined ISA implementation.

To illustrate the method, the *register/register arithmetic* and *branch* instructions of a typical SoC microprocessor are specified and refined. The technique, termed *forwarding*, where intermediate values are fed back to a stage that needs them, is employed in modern microprocessors to provide a very efficient means of managing data hazards [Hennessy and Patterson, 2006]. Finding errors in the forwarding logic has, however, been found to be difficult and expensive [Kroening and Paul, 2001]. With the introduction of appropriate invariants in our approach, it is shown that the concrete, pipelined refinement will not preserve these invariants unless the data hazards are detected and managed appropriately. The concrete Event-B model implements forwarding in a way that corresponds directly to the techniques used in microprocessor design and is proved, automatically, in the Rodin environment to be a correct refinement of the abstract ISA specification.

A method for managing pipeline control hazards caused by branching is to detect the branch instruction in the *Instruction Decode* pipeline stage and to *stall* the pipeline until it is known whether the branch is taken or not. If the branch is not taken, the pipeline flow can simply resume. If the branch is taken, pending instructions are flushed, the address of the next instruction is determined, and operation resumes at the new value of PC. Again, with the introduction of appropriate invariants, it will be proved that the pipelined implementation executes the instructions in accordance with the ISA specification.

The final, concrete models of the instructions comprise a set of Event-B machines, one for each pipeline stage. The machines representing each particular stage are then composed formally to provide a complete, concrete model of the pipeline. Thus, microarchitectural considerations are raised from the register transfer level to the specification level and design choices can be verified much earlier in the flow. It will be shown that the concrete model also has a direct correspondence to an equivalent hardware description at the register transfer level or in the high-level languages Bluespec and CAL, which like Event-B are based on guarded atomic actions. The method proposed therefore has the potential to be integrated into an existing synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

5.3 Abstracting and Refining the Arithmetic Instruction

In our approach, modelling begins with an abstract representation of the instruction that can be validated against the ISA. The microarchitecture of the abstract machine for the arithmetic instruction specifies its effect on the *register file.* The instruction is issued and completes in a single stage and is therefore free of data hazards. The abstract machine is then refined, introducing a single concrete pipeline stage at each refinement step while the rest of the pipeline is left abstract. In the first refinement, a 2-stage pipeline, the register file is read by the first stage and written by the second simultaneously which can result in RAW hazards. The abstract specification is therefore not met by the refinement and the proof obligations generated by the Rodin tool to verify the correctness of the refinement cannot be proved. The method forces the architect to address the inherently hazardous nature of the pipeline in order to prove the correctness on the refinement. It will be shown how the Event-B proof method helps in identifying the problem and the solution. It will be also be shown how the natural solution to this problem, *forwarding*, can be implemented in the refinement and how, with the use of appropriate gluing invariants, the refinement can be proved correct. The second refinement, a 3-stage pipeline, introduces additional RAW hazards which again can be dealt with by introducing forwarding. RAW hazards are symptomatic of the inherent feedback associated with reading and writing the register files simultaneously, and this simultaneity must be correctly modelled by the architect to ensure that the *Instruction Decode*, *Execute* and *Write Back* stages can be proved to implement the abstract specification, using gluing invariants that represent the whole feedback loop. Once the feedback loop has been modeled and the gluing invariants proved, showing that the overlapping execution within the feedback loop has been implemented correctly, decomposition of the pipeline stages into the individual processes which have a direct mapping to the final hardware implementation can occur.

5.3.1 The Abstract ISA Model

The structure of a *register/register arithmetic* instruction associates the opcode with a destination register *Rr* and two source registers *Ra* and *Rb*. The Event-B *context*, *PIPEC*, for the arithmetic instruction therefore defines a set of operations *Op*, the type *Register*, the subset of operations that are of type *register/register arithmetic*, *ArithRRop*, and the relationship between the fields of the arithmetic instruction and their associated registers. The conventions of Evans and Butler [2006] are followed to model operation fields as described in Section 2.9 on Page 17. The context also defines *No Operation*, *NOP*.

CONTEXT PIPEC

SETS

`Op`

CONSTANTS

`Register`, `Rr`, `Ra`, `Rb`, `NOP`, `ArithRROp`

AXIOMS

`axm1` : $Register \subseteq \mathbb{N}$

`axm2` : $Rr \in Op \rightarrow Register$

`axm3` : $Ra \in Op \rightarrow Register$

`axm4` : $Rb \in Op \rightarrow Register$

`axm5` : $ArithRROp \subseteq Op$

`axm6` : $NOP \in Op$

`axm7` : $NOP \notin ArithRROp$

END

Each *Op* has three register fields, modelled as projection functions, *(Rr, Ra, Rb)*. *ArithRROp* is the set of all register/register arithmetic operations. *NOP* is a null operation. The abstract machine, *PIPEM*, defines the register file *Regs* and a single event *ArithRR* that specifies the effect that execution of the instruction has on the register file. The microarchitecture of the abstract machine is shown in Figure 5.3.

For simplicity of presentation, the addition operation is shown, but this can more generally be represented by an uninterpreted function, Burch and Dill [1994], without affecting the proof approach used. The more general approach is outlined later in the section. The parameter *pop* specifies the environment for the event; given an instruction of type *ArithRROp*, the state of the register file will be updated according to that instruction.

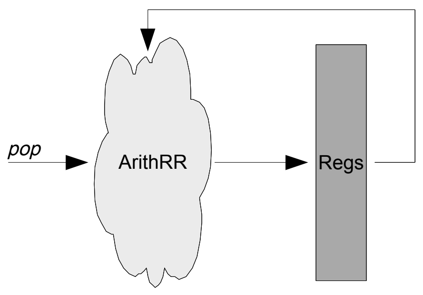


FIGURE 5.3: Abstract Machine: Microarchitecture

MACHINE PIPEM

SEES PIPEC

VARIABLES

Regs

INVARIANTS

inv1 : $Regs \in Register \rightarrow \mathbb{Z}$

EVENTS

Initialisation

```
    begin
        act1 : $Regs := Register \times \{0\}$
    end
```

Event $ArithRR \mathrel{\widehat{=}}$

```
    any
        $pop$
    where
        grd1 : $pop \in ArithRROp$
    then
        act1 : $Regs(Rr(pop)) := Regs(Ra(pop)) + Regs(Rb(pop))$
    end
```

END

5.3.2 The First Refinement: a 2-stage pipeline

A 2-stage pipeline is now introduced which refines the abstract machine. The second pipeline stage is a concrete representation of the *Write Back (WB)* stage while the first stage is still abstract, representing the *Fetch/Decode/Execute* operations of the pipeline. The microarchitecture of the refined machine is shown in Figure 5.4.

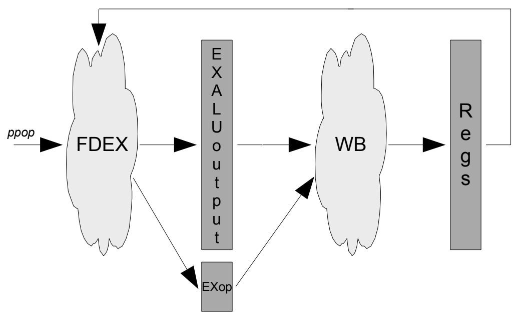


FIGURE 5.4: Refined Machine: Microarchitecture

MACHINE PIPER1

REFINES PIPEM

SEES PIPEC

VARIABLES

Regs, EXALUoutput, EXop

INVARIANTS

inv1 : $EXop \in Op$

inv2 : $EXALUoutput \in \mathbb{Z}$

inv3 : $EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop))$

EVENTS

Initialisation

begin

act1 : $Regs := Register \times \{0\}$

act2 : $EXop := NOP$

act3 : $EXALUoutput := 0$

end

Two new variables, *EXALUoutput* and *EXop* are introduced to represent the *EXWB* pipeline registers. The parameter *pop* of the abstract *ArithRR* event is bound to the concrete register *EXop* using an Event-B *witness* and a new parameter *ppop* represents the environment of the refined event, *FDEXWB*. The *FDEXWB* event models the simultaneous execution of both pipeline stages.

```
Event  $FDEXWB \widehat{=}$
refines  $ArithRR$
    any
        $ppop$
    where
        grd1 : $EXop \in ArithRROp$
        grd2 : $ppop \in ArithRROp$
    with
        pop : pop = EXop
    then
        act1 : $Regs(Rr(EXop)) := EXALUoutput$
        act2 : $EXALUoutput := Regs(Ra(ppop)) + Regs(Rb(ppop))$
        act3 : $EXop := ppop$
    end
END
```

It is now necessary to introduce the *gluing invariant* to establish that this is a correct refinement of the abstract machine. To preserve the meaning of the abstract specification, the new variable *EXALUoutput* must always have the value *Regs(Ra(EXop)) + Regs(Rb(EXop))*, as represented by the invariant *inv3*. The Rodin prover, however, cannot discharge the following proof obligation generated by the tool to show that the invariant is preserved.

$Ra(ppop) = Rr(EXop)$

$\vdash$

$Regs(Ra(ppop)) + Regs(Rb(ppop)) = EXALUoutput$

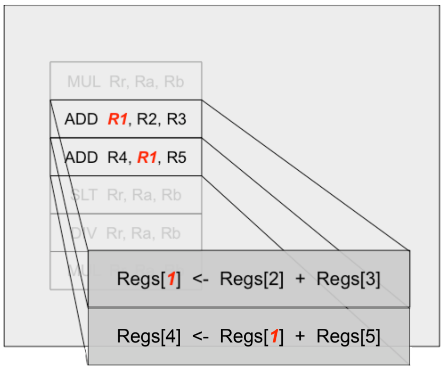


FIGURE 5.5: Successive Instructions can Interfere

5.3.3 Detecting the RAW Hazard

The abstract *FDEX* pipeline stage may only read from the source registers *Ra* and *Rb* if they do not coincide with the target register *Rr* of the previous instruction, represented by *Rr(EXop))*. This interference between successive instructions is shown in Figure 5.5.

Therefore, in the failing proof obligation,

$Ra(ppop) = Rr(EXop)$

$\vdash$

$Regs(Ra(ppop)) + Regs(Rb(ppop)) = EXALUoutput$

the goal may in general be false. One way of addressing this is to strengthen the hypothesis in order to falsify it. A new guard, *grd3* is therefore introduced into the refined event to model the case when there is no hazard on register *Ra*. A further guard, *grd4* is also introduced to check that there is no hazard on register *Rb*.

grd1 : ...

grd2 : ...

grd3 : $Rr(EXop) \neq Ra(ppop)$

grd4 : $Rr(EXop) \neq Rb(ppop)$

The Rodin prover now shows that the invariant

$EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop))$

is preserved by the refined machine.

5.3.4 Dealing Correctly with the RAW Hazard

It is now necessary to deal with the cases where a hazard is encountered on register Ra alone, on register Rb alone and on both registers Ra and Rb. In each case, the required value(s) can be read from the $EXALUoutput$ register. This corresponds directly to the *forwarding* technique used in microprocessor design. Three extra events are introduced to deal with each case. For instance, for the hazard on register Ra, the guards of the event are

$$\texttt{grd3}: \; Rr(EXop) = Ra(ppop)$$

$$\texttt{grd4}: \; Rr(EXop) \neq Rb(ppop)$$

and the associated action now reads the value of Ra from $EXALUoutput$.

$$\texttt{act2}: \; EXALUoutput := EXALUoutput + Regs(Rb(ppop))$$

The Rodin prover shows that, for each case, the invariant is preserved. The microarchitecture of the modified refined machine is shown in Figure 5.6.

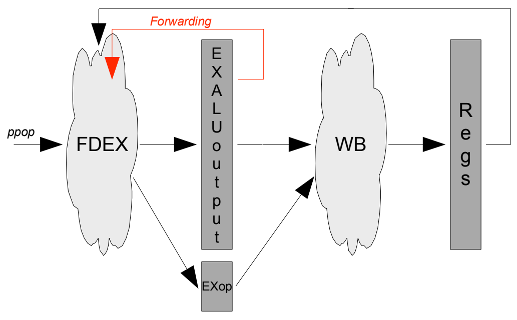


FIGURE 5.6: Refined Machine with Forwarding: Microarchitecture

5.3.5 The Second Refinement: a 3-stage pipeline

In the second refinement, the concrete Execute *(EX)* stage is introduced together with the *IDEX* pipeline registers.

The microarchitecture is shown in Figure 5.7.

The registers A and B store the values of Ra and Rb respectively. The following two new gluing invariants are introduced.

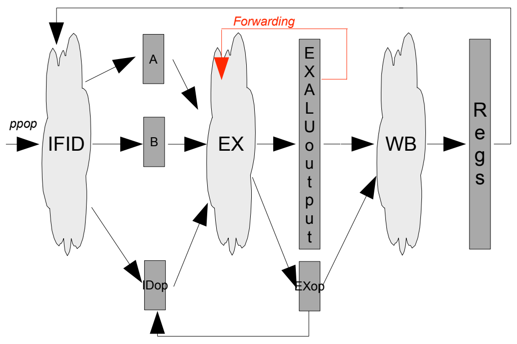


FIGURE 5.7: Second Refinement: Microarchitecture

inv1 : $A = Regs(Ra(IDop))$

inv2 : $B = Regs(Rb(IDop))$

The introduction of the third pipeline stage in this refinement introduces further data hazards to those dealt with in the previous section. In addition to adjacent instructions interfering, in the 3-stage pipeline instructions that are one apart can also interfere, as shown in Figure 5.8. If, for instance, the source register Ra coincides with the target register Rr in the $EXop$ register, then the register A must have its value forwarded from $EXALUoutput$ rather than getting it directly from the register file.

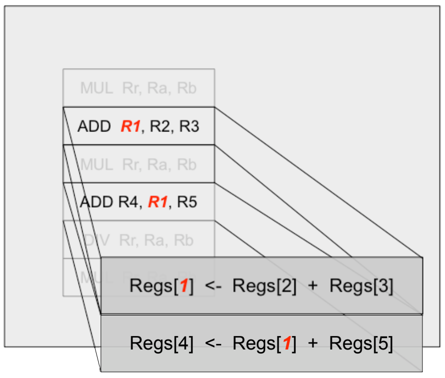


FIGURE 5.8: Instructions one apart can Interfere

All possible RAW hazard combinations must be dealt with. Four potential hazards are incurred by adjacent instructions and four by instructions one apart leading to a total of sixteen combinations, modeled by sixteen events. For example in the following event $IDRAWaEXWBnoRAW$, the guards

$\texttt{grd3}: \ Rr(EXop) \neq Ra(IDop)$

$\texttt{grd4}: \ Rr(EXop) \neq Rb(IDop)$

cater for the case where there is no interference between the adjacent *IDop* and *EXop* instructions, while the guards

$\texttt{grd5}: \ Rr(EXop) = Ra(pppop)$

$\texttt{grd6}: \ Rr(EXop) \neq Rb(pppop)$

cater for the case where the source register *Ra* of the fetched instruction is the same as the target *Rr* of the *EXop* instruction. In this case *EXALUoutput* is updated normally using the values from the registers *A* and *B*,

$\texttt{act2}: \ EXALUoutput := A + B$

register *B* gets its value directly from the register file,

$\texttt{act5}: \ B := Regs(Rb(pppop))$

but register *A* must have its value forwarded from *EXALUoutput*

$\texttt{act4}: \ A := EXALUoutput$

The complete description of event *IDRAWaEXWBnoRAW* follows.

Event *IDRAWaEXWBnoRAW* $\widehat{=}$

refines *EXWBnoRAW*

  any

    pppop

  where

    $\texttt{grd1}: \ EXop \in ArithRROp$

    $\texttt{grd2}: \ IDop \in ArithRROp$

    $\texttt{grd3}: \ Rr(EXop) \neq Ra(IDop)$

    $\texttt{grd4}: \ Rr(EXop) \neq Rb(IDop)$

    $\texttt{grd5}: \ Rr(EXop) = Ra(pppop)$

    $\texttt{grd6}: \ Rr(EXop) \neq Rb(pppop)$

```
    with
        ppop : ppop = IDop
    then
        act1 : $Regs(Rr(EXop)) := EXALUoutput$
        act2 : $EXALUoutput := A + B$
        act3 : $EXop := IDop$
        act4 : $A := EXALUoutput$
        act5 : $B := Regs(Rb(pppop))$
        act6 : $IDop := pppop$
    end
```

The case where there are no hazards is managed by the event *IDnoRAWEXWBnoRAW*

```
Event  $IDnoRAWEXWBnoRAW \widehat{=}$
refines  $EXWBnoRAW$
    any
        $pppop$
    where
        grd1 : $EXop \in ArithRROp$
        grd2 : $IDop \in ArithRROp$
        grd3 : $Rr(EXop) \neq Ra(IDop)$
        grd4 : $Rr(EXop) \neq Rb(IDop)$
        grd5 : $Rr(EXop) \neq Ra(pppop)$
        grd6 : $Rr(EXop) \neq Rb(pppop)$
    with
        ppop : ppop = IDop
    then
        act1 : $Regs(Rr(EXop)) := EXALUoutput$
        act2 : $EXALUoutput := A + B$
        act3 : $EXop := IDop$
        act4 : $A := Regs(Ra(pppop))$
        act5 : $B := Regs(Rb(pppop))$
        act6 : $IDop := pppop$
    end
```

and the case where hazards are encountered on both *Ra* and *Rb* for both adjacent and one-apart instructions is managed by the event *IDRAWabEXWBabRAW*.

Event $IDRAWabEXWBabRAW \mathrel{\widehat{=}}$

refines $EXWBabRAW$

any

$pppop$

where

grd1 : $EXop \in ArithRROp$

grd2 : $IDop \in ArithRROp$

grd3 : $Rr(EXop) = Ra(IDop)$

grd4 : $Rr(EXop) = Rb(IDop)$

grd5 : $Rr(EXop) = Ra(pppop)$

grd6 : $Rr(EXop) = Rb(pppop)$

with

ppop : ppop = IDop

then

act1 : $Regs(Rr(EXop)) := EXALUoutput$

act2 : $EXALUoutput := EXALUoutput + EXALUoutput$

act3 : $EXop := IDop$

act4 : $A := EXALUoutput$

act5 : $B := EXALUoutput$

act6 : $IDop := pppop$

end

In this case the registers *EXALUoutput*, *A* and *B* are all updated with the forwarded value from *EXALUoutput*. The microarchitecture, with forwarding, is shown in Figure 5.9.

All the proof obligations generated are discharged automatically by the Rodin tool, as shown in Table 5.2.

	Total no. of proof obligations	Discharged Automatically
Abstract Model	3	3
1st Refinement	33	33
2nd Refinement	192	192

TABLE 5.2: Pipeline Proofs

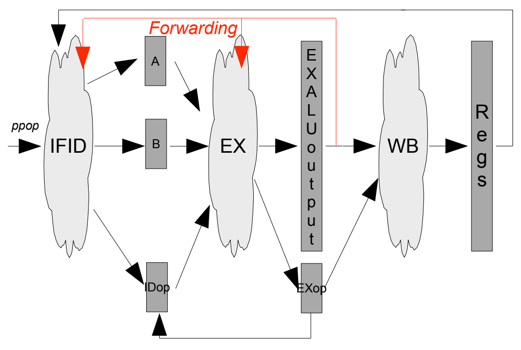


FIGURE 5.9: Second Refinement with Forwarding: Microarchitecture

5.3.6 Shared Event Decomposition of the Feedback Loop

Since the *ArithRR* instruction does not access memory, the *MEM* stage does nothing other than pass on values to the next pipeline stage and therefore, for clarity of presentation, it has been omitted. In the final four-stage pipeline, as shown in Figure 5.10, there is no feedback between the IF stage and the rest of the pipeline. It is therefore possible, now that the global gluing invariants for the feedback loop have been proved, to perform a three-way decomposition of the sixteen events of the second refinement into three distinct machines representing the concrete EX and WB stages and the IFID stage which is still abstract. The Event-B shared event decomposition mechanism is explained in Section 2.9 on Page 17. Each Event-B machine represents a *process* communicating by shared pipeline variables.

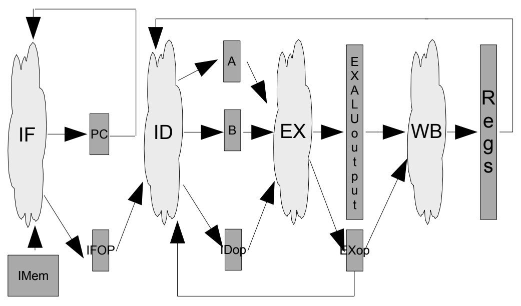


FIGURE 5.10: Third Refinement: Microarchitecture

We now outline the decomposed components. The WB stage is represented by a machine with a single event.

Event $WB \mathrel{\widehat{=}}$

> **when**
>
> grd1 : $EXop \in ArithRROp$
>
> **begin**
>
> act1 : $Regs(Rr(EXop)) := EXALUoutput$
>
> **end**

The WB event reads the $EXALUoutput$ register and writes to the Register File.

The EX stage is represented by a machine comprising four mutually exclusive events. For instance, in the case when a hazard is encountered on register Ra alone,

Event $EXaRAW \mathrel{\widehat{=}}$

> **when**
>
> grd1 : $IDop \in ArithRROp$
>
> grd2 : $EXop \in ArithRROp$
>
> grd3 : $Rr(EXop) = Ra(IDop)$
>
> grd4 : $Rr(EXop) \neq Rb(IDop)$
>
> **then**
>
> act1 : $EXALUoutput := EXALUoutput + B$
>
> act2 : $EXop := IDop$
>
> **end**

The EX events read from the $IDOP$, A and B registers and write to the $EXop$ and $EXALUoutput$ registers.

The abstract IFID stage is also represented by a machine comprising four mutually exclusive events. For instance, in the case when a hazard is encountered on both source registers Ra and Rb,

Event $IFIDRAWab \mathrel{\widehat{=}}$

> **any**
>
> $pppop$
>
> **where**
>
> grd1 : $EXop \in ArithRROp$
>
> grd2 : $IDop \in ArithRROp$
>
> grd3 : $Rr(EXop) = Ra(pppop)$

grd4 : $Rr(EXop) = Rb(pppop)$

then

act1 : $A := EXALUoutput$

act2 : $B := EXALUoutput$

act3 : $IDop := pppop$

end

This gives a total of nine events in three machines for the IFID, EX and WB stages, compared with the 16 combined events before decomposition. Decomposing as soon as the feedback loop has been verified, stems the exponential increase in the number of events.

5.3.7 The Third Refinement: a 4-stage pipeline

The abstract IFID model is now refined, introducing the concrete program counter *PC* and the instruction memory *IMem*

CONSTANTS

IMSIZE

IMem

AXIOMS

axm1 : $IMSIZE \in \mathbb{N}$

axm2 : $IMem \in 0 \mathrel{..} IMSIZE \rightarrow Op$

END

resulting in four concrete events which represent the combined operation of the IF and ID stages. In the case when a hazard is encountered on both source registers *Ra* and *Rb*,

Event *IFIDRAWab* $\widehat{=}$

refines *IFIDRAWab*

where

grd1 : $EXop \in ArithRROp$

grd2 : $IDop \in ArithRROp$

grd3 : $Rr(EXop) = Ra(IFop)$

```
        grd4 : $Rr(EXop) = Rb(IFop)$
        grd5 : $PC < IMSIZE$
    with
        ppop : pppop = IFop
    then
        act1 : $A := EXALUoutput$
        act2 : $B := EXALUoutput$
        act3 : $IDop := IFop$
        act4 : $PC := PC + 1$
        act5 : $IFop := IMem(PC)$
    end
```

The concrete IFID machine is now decomposed into two machines, IF and ID which represent the Instruction Fetch and Instruction Decode stages of the hardware pipeline respectively, as shown in Figure 5.10.

The Instruction Fetch stage is represented by a single event,

```
Event  $IF \widehat{=}$
    when
        grd1 : $PC < IMSIZE$
    then
        act1 : $PC := PC + 1$
        act2 : $IFop := IMem(PC)$
    end
```

which reads and writes PC and writes to the $IFop$ register,

and the Instruction Decode stage by four events of the form

```
Event  $IFIDRAWab \widehat{=}$
    where
        grd1 : $EXop \in ArithRROp$
        grd2 : $IDop \in ArithRROp$
        grd3 : $Rr(EXop) = Ra(IFop)$
        grd4 : $Rr(EXop) = Rb(IFop)$
    then
```

act1 : $A := EXALUoutput$

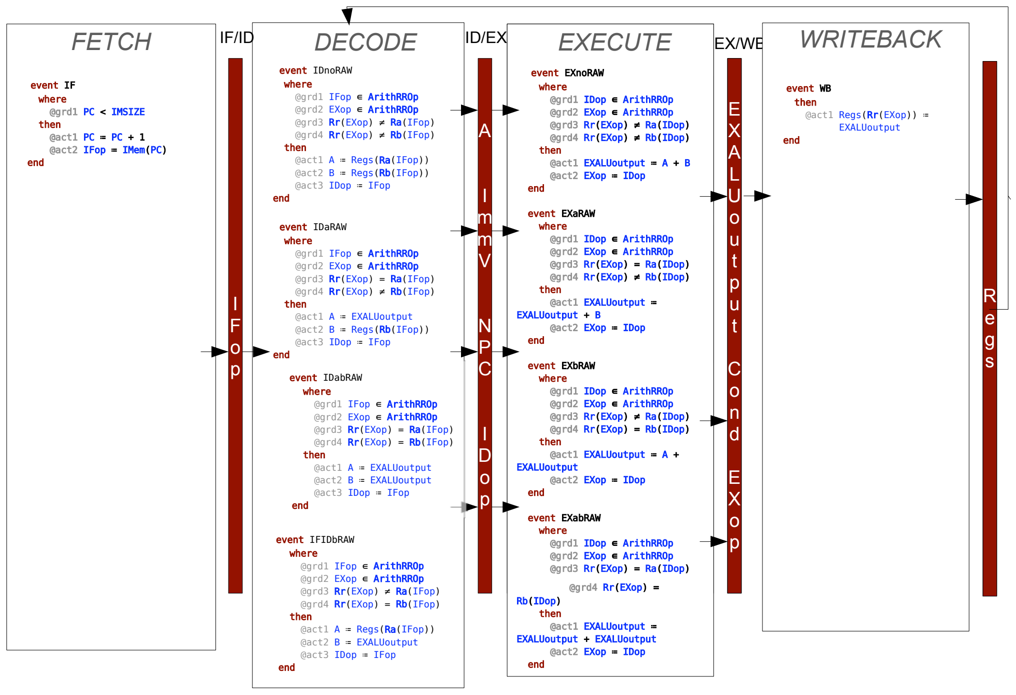
act2 : $B := EXALUoutput$

act3 : $IDop := IFop$

end

which read from the $EXALUoutput$ and $IFOP$ registers and write to the $IDop$, A and B registers, resulting in a total of ten events in four processes.

All parameters have been resolved to concrete pipeline registers, and each process may be mapped directly to an RTL representation. Each of the $DECODE$ and $EXECUTE$ processes comprise four mutually exclusive events which are mapped to four branches in an RTL case statement. The final concrete pipeline is shown in Figure 5.11.



```
event IF
  where
    @grd1 PC < IMSIZE
  then
    @act1 PC ≔ PC + 1
    @act2 IFop ≔ IMem(PC)
end
```

```
event IDnoRAW
  where
    @grd1 IFop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) ≠ Ra(IFop)
    @grd4 Rr(EXop) ≠ Rb(IFop)
  then
    @act1 A ≔ Regs(Ra(IFop))
    @act2 B ≔ Regs(Rb(IFop))
    @act3 IDop ≔ IFop
end

event IDaRAW
  where
    @grd1 IFop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(IFop)
    @grd4 Rr(EXop) ≠ Rb(IFop)
  then
    @act1 A ≔ EXALUoutput
    @act2 B ≔ Regs(Rb(IFop))
    @act3 IDop ≔ IFop
end

  event IDabRAW
    where
      @grd1 IFop ∈ ArithRROp
      @grd2 EXop ∈ ArithRROp
      @grd3 Rr(EXop) = Ra(IFop)
      @grd4 Rr(EXop) = Rb(IFop)
    then
      @act1 A ≔ EXALUoutput
      @act2 B ≔ EXALUoutput
      @act3 IDop ≔ IFop
  end

event IFIDbRAW
  where
    @grd1 IFop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) ≠ Ra(IFop)
    @grd4 Rr(EXop) = Rb(IFop)
  then
    @act1 A ≔ Regs(Ra(IFop))
    @act2 B ≔ EXALUoutput
    @act3 IDop ≔ IFop
end
```

```
event EXnoRAW
  where
    @grd1 IDop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) ≠ Ra(IDop)
    @grd4 Rr(EXop) ≠ Rb(IDop)
  then
    @act1 EXALUoutput ≔ A + B
    @act2 EXop ≔ IDop
end

event EXaRAW
  where
    @grd1 IDop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(IDop)
    @grd4 Rr(EXop) ≠ Rb(IDop)
  then
    @act1 EXALUoutput ≔
EXALUoutput + B
    @act2 EXop ≔ IDop
end

event EXbRAW
  where
    @grd1 IDop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) ≠ Ra(IDop)
    @grd4 Rr(EXop) = Rb(IDop)
  then
    @act1 EXALUoutput ≔ A +
EXALUoutput
    @act2 EXop ≔ IDop
end

event EXabRAW
  where
    @grd1 IDop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(IDop)
        @grd4 Rr(EXop) =
Rb(IDop)
  then
    @act1 EXALUoutput ≔
EXALUoutput + EXALUoutput
    @act2 EXop ≔ IDop
end
```

```
event WB
  then
    @act1 Regs(Rr(EXop)) ≔
          EXALUoutput
end
```

Figure 5.11: ArithRR 4-stage Pipeline

5.3.8 Generalising the ArithRR model

To generalise this approach for uninterpreted arithmetic operations, the action

$\texttt{act1}: \mathit{Regs}(\mathit{Rr}(p)) := \mathit{Regs}(\mathit{Ra}(p)) + \mathit{Regs}(\mathit{Rb}(p))$

can be replaced with

$\texttt{act1}: \mathit{Regs}(\mathit{Rr}(p)) := f(\mathit{Regs}(\mathit{Ra}(p)) \mapsto \mathit{Regs}(\mathit{Rb}(p)))$

where

$\texttt{grd1}: f = \mathit{func}(p)$

and

$\texttt{axm8}: \mathit{func} \in \mathit{Op} \rightarrow (\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N})$

is a field of the arithmetic instruction. The invariants are generalised as well.

5.4 Abstracting and Refining the Branch Instruction

The branch instruction presents a different modelling challenge to the arithmetic instruction. Incorrect branch prediction results in a *Control Hazard*, which must be managed correctly in the pipeline. The implementation must detect the hazard, stall the pipeline and flush out the incorrectly pre-fetched instructions.

Modelling begins with an abstract representation of the instruction that can be validated against the ISA. The microarchitecture of the abstract machine for the branch instruction specifies its effect on an abstract variable that represents the next instruction to be executed. The instruction is issued and completes in a single stage and therefore the next instruction to be executed is always known. The abstract machine is then refined, introducing a single concrete pipeline stage at each refinement step while the rest of the pipeline is left abstract. In the first refinement, the registers associated with the *Execute* stage are introduced. In the second refinement, the *Write Back* stage is developed, which introduces RAW hazards on the *Ra* source register. As with the arithmetic instruction, the hazards are managed using forwarding. In the third refinement, the IDEX pipeline registers are introduced, together with a 2-bit counter *Stall* to manage pipeline stalling. In the fourth refinement, the program counter *PC* is introduced, together with a set of gluing invariants that link *PC* to the abstract variable that represents the next instruction to be executed. Finally, in the fifth refinement, this abstract variable is removed since it is not needed in the implementation. Once this is done, decomposition of the pipeline stages into individual processes can occur.

5.4.1 The Abstract ISA Model

The microarchitecture of the abstract model is shown in Figure 5.12.

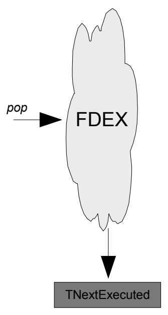


FIGURE 5.12: AbstractMachine: Microarchitecture

The abstract machine, *bPIPEM*, defines two parameterised events, *NoBranchp* and *Branchp*, that specify the effect that execution of a branch instruction has on the program counter PC when a branch fails or succeeds respectively. The current value of PC is represented by the parameter *ppc* and the next value of PC by *ppcprime*. The current instruction in Instruction Memory is represented by *pop* and the Register File by *pr*. The value of *pop's* source register, *Ra*, is compared with 0 using *BoolOp* which represents either a *BEQ0* or a *BNE0* instruction. If the comparison fails, PC is incremented by one. Otherwise, PC is incremented by the *Immediate* value of *pop*. The variable *TNextExecuted* is introduced to represent the address of the next instruction to be executed. The event *EX* represents the effect on *TNextExecuted* by the execution of instructions other than *Branch*.

Event *NoBranchp* $\widehat{=}$

any

$pop, pr, ppc, ppcprime$

where

grd1 : $pop \in BranchOp$

grd2 : $ppc \in 0 \,..\, IMSIZE$

grd3 : $pr \in Register \rightarrow \mathbb{Z}$

grd4 : $BoolOp(pr(Ra(pop)) \mapsto 0) = 0$

grd5 : $ppcprime = ppc + 1$

then

act1 : $TNextExecuted := ppcprime$

end

Event *Branchp* $\widehat{=}$

any

$pop, pr, ppc, ppcprime$

where

grd1 : $pop \in BranchOp$

grd2 : $ppc \in 0 \,..\, IMSIZE$

grd3 : $pr \in Register \rightarrow \mathbb{Z}$

grd4 : $BoolOp(pr(Ra(pop)) \mapsto 0) = 1$

grd5 : $ppcprime = ppc + Imm(pop)$

then

act1 : $TNextExecuted := ppcprime$

end

```
Event  EX ≙
    any
        pop, ppc, ppcprime
    where
        grd1 : $pop \notin BranchOp$
        grd2 : $ppc \in 0 \,..\, IMSIZE$
        grd3 : $ppcprime = ppc + 1$
    then
        act1 : $TNextExecuted := ppcprime$
    end
```

The abstract machine executes instructions in a single stage and determines the next instruction to be executed. In the final pipelined implementation it will be necessary to prove that the program counter is updated in accordance with the abstract specification; gluing invariants will need to be introduced to link the concrete program counter to the address of the next instruction executed, represented by the variable *TNextExecuted*.

5.4.2 The First Refinement

The registers associated with EX stage, *EXALUoutput*, *Cond* and *EXop* are introduced, as shown in Figure 5.13.

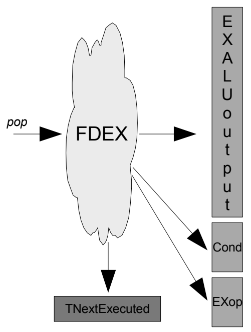


Figure 5.13: Refined Machine: Microarchitecture

The current instruction address is added to the *Immediate* value of the branching instruction and the result written to *EXALUoutput* irrespective of whether the branch will be taken or not. *Cond*, the register which will be used to determine whether the branch is taken or not, is set to 1 by the *EXBranch* event and 0 by the *EXnoBranch* event. In the final pipeline, *EXALUoutput* and *Cond* are read by the *Instruction Fetch* stage to calculate the new value of PC.

```
Event  EXBranch ≙
refines  Branchp
    any
        pop, pr, ppc, ppcprime
    where
        grd1 : $pop \in BranchOp$
        grd2 : $ppc \in 0 \,..\, IMSIZE$
        grd3 : $pr \in Register \rightarrow \mathbb{Z}$
        grd4 : $BoolOp(pr(Ra(pop)) \mapsto 0) = 1$
        grd5 : $ppcprime = ppc + Imm(pop)$
    then
        act1 : $EXALUoutput := ppc + Imm(pop)$
        act2 : $Cond := BoolOp(pr(Ra(pop)) \mapsto 0)$
        act3 : $EXop := pop$
        act4 : $TNextExecuted := ppcprime$
    end

Event  EXNoBranch ≙
refines  NoBranchp
    any
        pop, pr, ppc, ppcprime
    where
        grd1 : $pop \in BranchOp$
        grd2 : $ppc \in 0 \,..\, IMSIZE$
        grd3 : $pr \in Register \rightarrow \mathbb{Z}$
        grd4 : $BoolOp(pr(Ra(pop)) \mapsto 0) = 0$
        grd5 : $ppcprime = ppc + 1$
    then
        act1 : $EXALUoutput := ppc + Imm(pop)$
        act2 : $Cond := BoolOp(pr(Ra(pop)) \mapsto 0)$
        act3 : $EXop := pop$
        act4 : $TNextExecuted := ppcprime$
    end
```

The event, which represents all other non-branching instructions, must set the value of *Cond* to 0 to ensure that correct next instruction is fetched.

```
Event  EX ≙
refines  EX
    any
        $pop, ppc, ppcprime$
    where
        grd1 : $pop \notin BranchOp$
        grd2 : $ppc \in 0 \,..\, IMSIZE$
        grd3 : $ppcprime = ppc + 1$
    then
        act1 : $EXALUoutput :\in \mathbb{Z}$
        act2 : $Cond := 0$
        act3 : $EXop := pop$
        act4 : $TNextExecuted := ppcprime$
    end
```

5.4.3 The Second Refinement: a 2-stage pipeline

In the second refinement, the *Register File* and the *Write Back* stage is introduced as shown in Figure 5.14. Although the *branch* instruction does not itself modify the registers, an earlier *arithmetic* instruction in the pipeline can. A RAW hazard is encountered if a *branch* instruction reads the value of the *Ra* register at the same time that an *arithmetic* instruction is writing back to the same register. The hazard is dealt with using forwarding.

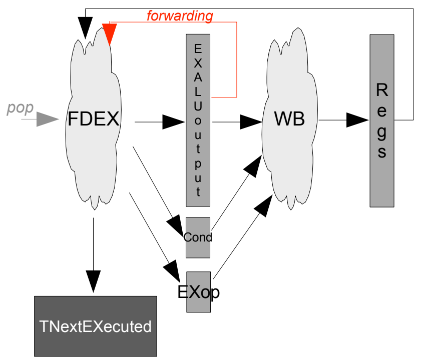


FIGURE 5.14: Second Refinement

Two events are required for the case when no branch is taken and the *EXop* instruction is an *ArithRROp*. The first, when there is a hazard on register *Ra*, $Rr(EXop) = Ra(pop)$ and the evaluation of *BoolOp* must use the forwarded value,

Event $EXRAWaNoBranchWB \mathrel{\widehat{=}}$

refines $EXNoBranch$

```
    any
        $pop, ppc, ppcprime$
    where
        grd1 : $pop \in BranchOp$
        grd2 : $ppc \in 0 \,..\, IMSIZE$
        grd3 : $BoolOp(Regs(Ra(pop)) \mapsto 0) = 0$
        grd4 : $ppcprime = ppc + 1$
        grd5 : $EXop \in ArithRROp$
        grd6 : $Rr(EXop) = Ra(pop)$
    with
        pr : pr = Regs
    then
        act1 : $EXALUoutput := ppc + Imm(pop)$
        act2 : $Cond := BoolOp(EXALUoutput \mapsto 0)$
        act3 : $EXop := pop$
        act4 : $TNextExecuted := ppcprime$
        act5 : $Regs(Rr(EXop)) := EXALUoutput$
    end
```

and the second, where there is no hazard and $Rr(EXop) \neq Ra(pop)$

Event $EXnoRAWNoBranchWB \mathrel{\widehat{=}}$

refines $EXNoBranch$

```
    any
        $pop, ppc, ppcprime$
    where
        grd1 : $pop \in BranchOp$
        grd2 : $ppc \in 0 \,..\, IMSIZE$
        grd3 : $BoolOp(Regs(Ra(pop)) \mapsto 0) = 0$
        grd4 : $ppcprime = ppc + 1$
        grd5 : $EXop \in ArithRROp$
        grd6 : $Rr(EXop) \neq Ra(pop)$
    with
```

```
        pr : $\mathtt{pr} = \mathtt{Regs}$
    then
        act1 : $EXALUoutput := ppc + Imm(pop)$
        act2 : $Cond := BoolOp(Regs(Ra(pop)) \mapsto 0)$
        act3 : $EXop := pop$
        act4 : $TNextExecuted := ppcprime$
        act5 : $Regs(Rr(EXop)) := EXALUoutput$
    end
```

Similary, two events are required for the case when the branch is taken. The hazard free event is shown here.

```
Event  $EXnoRAWBranchWB \mathrel{\widehat{=}}$
refines  $EXBranch$
    any
        $pop, ppc, ppcprime$
    where
        grd1 : $pop \in BranchOp$
        grd2 : $ppc \in 0 \,..\, IMSIZE$
        grd3 : $BoolOp(Regs(Ra(pop)) \mapsto 0) = 1$
        grd4 : $ppcprime = ppc + Imm(pop)$
        grd5 : $EXop \in ArithRROp$
        grd6 : $Rr(EXop) \neq Ra(pop)$
    with
        pr : $\mathtt{pr} = \mathtt{Regs}$
    then
        act1 : $EXALUoutput := ppc + Imm(pop)$
        act2 : $Cond := BoolOp(Regs(Ra(pop)) \mapsto 0)$
        act3 : $EXop := pop$
        act4 : $TNextExecuted := ppcprime$
        act5 : $Regs(Rr(EXop)) := EXALUoutput$
    end
```

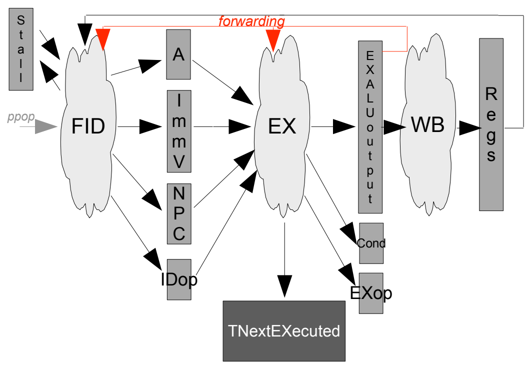


FIGURE 5.15: Third Refinement

5.4.4 The Third Refinement: a 3-stage pipeline

The IDEX pipeline registers are introduced, together with a 2-bit counter *Stall* to manage the pipeline stalling mechanism as shown in Figure 5.15.

Further RAW hazards are encountered in this refinement if the ID events load the value of the *Ra* register into the pipeline register *A* at the same time that an *arithmetic* instruction is writing back to the same register. These hazards are again dealt with using forwarding.

Stall is initialised to 0 and while no branch instruction is encountered stays at 0.

```
**Event** $IDEXWB \mathrel{\widehat{=}}$
**refines** $EXWB$
    **any**
        $ppop, pnpc, ppcprime$
    **where**
        grd1 : $IDop \notin BranchOp$
        grd2 : $ppop \notin BranchOp$
        grd3 : $Stall = 0$
        grd4 : $pnpc \in 0 \,..\, IMSIZE$
        grd5 : $EXop \in ArithRROp$
        grd6 : $ppcprime = pnpc$
    **with**
        pop : pop = IDop
    **then**
```

```
        act1 : $EXALUoutput :\in \mathbb{Z}$
        act2 : $Cond := 0$
        act3 : $EXop := IDop$
        act4 : $TNextExecuted := ppcprime$
        act5 : $Regs(Rr(EXop)) := EXALUoutput$
        act6 : $IDop := ppop$
        act7 : $A := Regs(Ra(ppop))$
        act8 : $ImmV := Imm(ppop)$
        act9 : $NPC := pnpc$
    end
```

When a branch operation is encountered, *Stall* is set to 2 which enables one of the two following events that establish whether the branch is taken or not, calculate the next value of PC, decrement *Stall* and set the speculatively fetched instruction to *NOP*.

```
Event  $IDStallEXNoBranchWB \mathrel{\widehat{=}}$
refines  $EXNoBranchWB$
    any
        $ppop, ppcprime$
    where
        grd1 : $IDop \in BranchOp$
        grd2 : $BoolOp(A \mapsto 0) = 0$
        grd3 : $ppcprime = NPC + 1$
        grd4 : $ppop \in Op$
        grd5 : $EXop \in ArithRROp$
        grd6 : $Stall > 0$
    with
        pop : pop = IDop
        ppc : ppc = NPC
    then
        act1 : $EXALUoutput := NPC + ImmV$
        act2 : $Cond := BoolOp(A \mapsto 0)$
        act3 : $EXop := IDop$
        act4 : $TNextExecuted := ppcprime$
        act5 : $Regs(Rr(EXop)) := EXALUoutput$
        act6 : $IDop := NOP$
        act7 : $Stall := Stall - 1$
        act8 : $A := Regs(Ra(NOP))$
        act9 : $ImmV := Imm(NOP)$
    end
Event  $IDStallEXBranchWB \mathrel{\widehat{=}}$
refines  $EXBranchWB$
    any
        $ppop, ppcprime$
    where
        grd1 : $IDop \in BranchOp$
        grd2 : $BoolOp(A \mapsto 0) = 1$
        grd4 : $ppcprime = NPC + Imm(IDop)$
        grd5 : $EXop \in ArithRROp$
        grd6 : $ppop \in Op$
        grd7 : $Stall > 0$
    with
        pop : pop = IDop
        ppc : ppc = NPC
    then
        act1 : $EXALUoutput := NPC + ImmV$
        act2 : $Cond := BoolOp(A \mapsto 0)$
        act3 : $EXop := IDop$
        act4 : $TNextExecuted := ppcprime$
        act5 : $Regs(Rr(EXop)) := EXALUoutput$
        act6 : $IDop := NOP$
        act7 : $Stall := Stall - 1$
        act8 : $A := Regs(Ra(NOP))$
        act9 : $ImmV := Imm(NOP)$
    end
```

The pipeline stalls for a further cycle to ensure that all the speculatively-fetched instructions have been flushed, represented by a stall event that decrements *Stall* to 0 so that normal pipeline operation can resume.

```
Event  IDStallEXWB ≙
refines  EXWB
    any
        ppop
    where
        grd1 : $IDop \notin BranchOp$
        grd2 : $EXop \in ArithRROp$
        grd3 : $ppop \in Op$
        grd4 : $Stall > 0$
    with
        pop : pop = IDop
    then
        act1 : $EXALUoutput :\in \mathbb{Z}$
        act2 : $Cond := 0$
        act3 : $EXop := IDop$
        act4 : $Regs(Rr(EXop)) := EXALUoutput$
        act5 : $IDop := NOP$
        act6 : $Stall := Stall - 1$
        act7 : $A := Regs(Ra(NOP))$
        act8 : $ImmV := Imm(NOP)$
    end
```

Three gluing invariants establish that the refinement is correct.

inv7 : $IDop \in BranchOp \Rightarrow A = Regs(Ra(IDop))$

inv8 : $IDop \in BranchOp \Rightarrow ImmV = Imm(IDop)$

inv9 : $Stall = 1 \Rightarrow IDop = NOP$

5.4.5 The Fourth Refinement: a 4-stage pipeline

It is at this stage that the program counter, PC, is introduced together with the IFID pipeline registers and the instruction memory *IMEM* as shown in Figure 5.16.

It is now necessary to introduce the gluing invariants that will prove that the program counter is updated in accordance with the abstract specification. This is done by linking the concrete variable *PC* to the variable *INPC*, which in turn is linked to *NPC* which

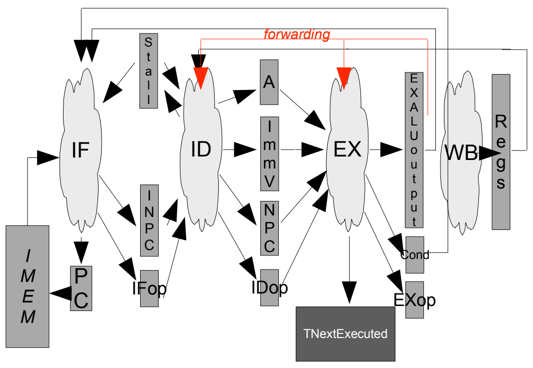


FIGURE 5.16: Fourth Refinement

finally is linked to the abstract variable *TNextExecuted*. Informally,

INPC always has the same value as PC.

If IDop is not a NOP then INPC holds the addresss that immediately succeeds NPC in Instruction Memory.

If IDop is not a NOP then NPC holds the address of the next instruction to be executed.

Formally,

$\texttt{inv4}: \ INPC = PC$

$\texttt{inv5}: \ IDop \neq NOP \Rightarrow INPC = NPC + 1$

$\texttt{inv6}: \ IDop \neq NOP \Rightarrow NPC = TNextExecuted$

It should be noted that, because there is no *no operation* instruction in the user-accessible DLX instruction set, *NOP* cannot occur in the input instruction stream.

5.4.6 The Fifth Refinement

Now that the gluing invariants have established, formally, the link between PC and *TNextExecuted*, the 5th refinement removes *TNextExecuted* since it is not needed in the implementation. *TNextExecuted* is a convenient modelling abstraction representing a notional PC for a non-pipelined ISA. Its relationship to real registers is given by the invariants *inv4-6* above.

Figure 5.17 shows the control flow of the pipelined branch operation as a state machine.

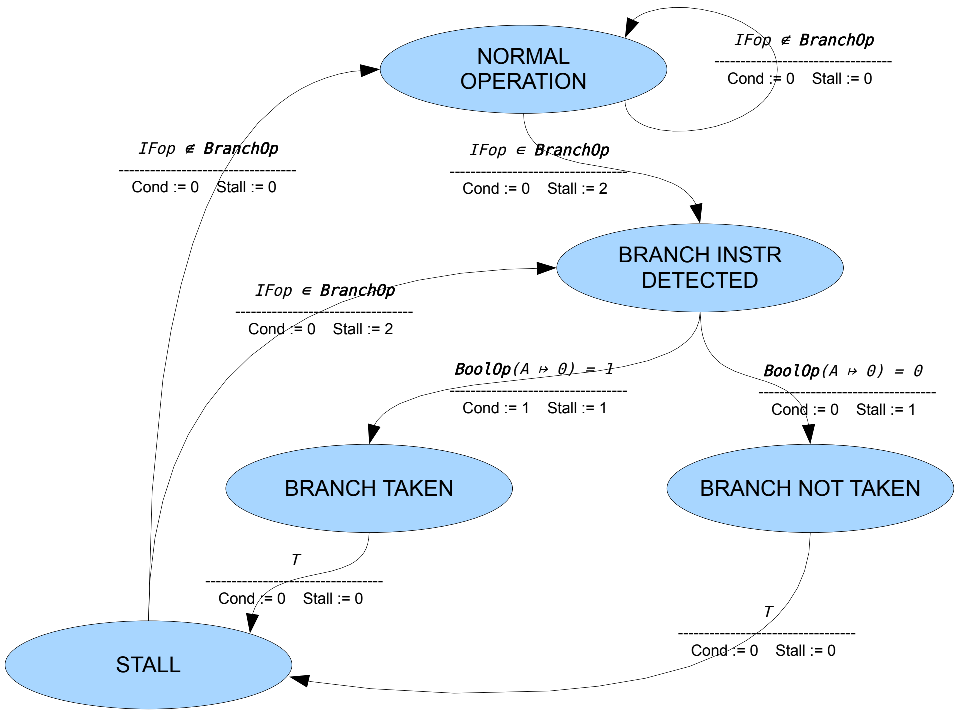


FIGURE 5.17: Fifth Refinement: Combined State Machine

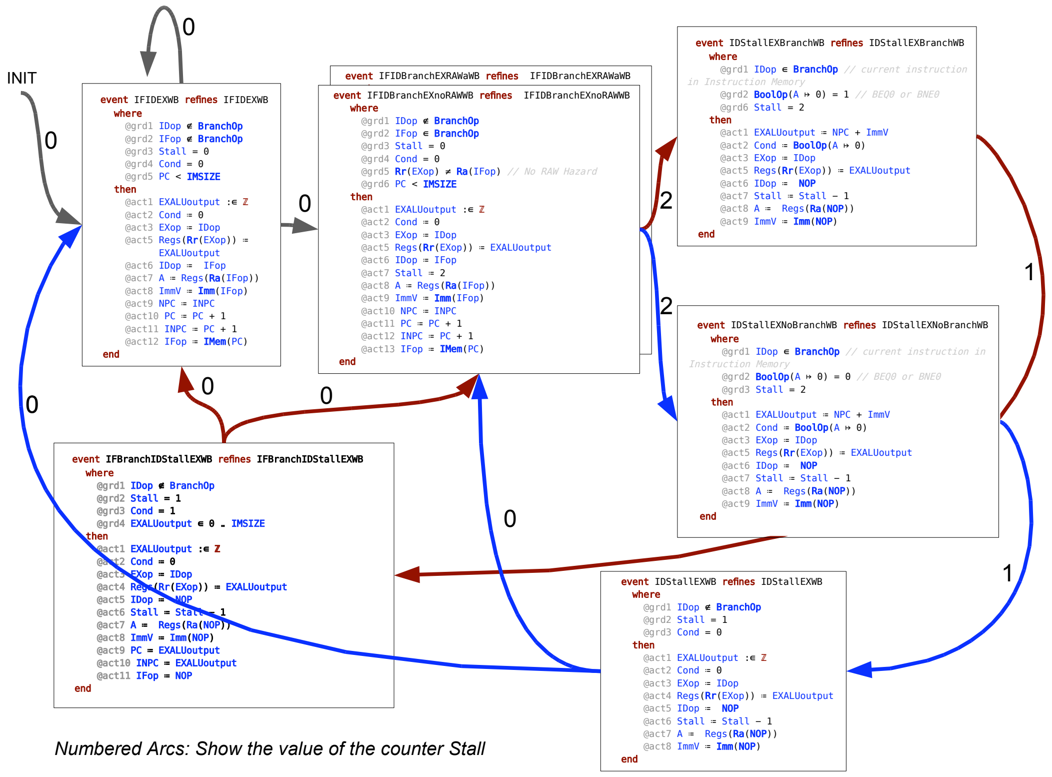
Under normal operation, in the absence of branching instructions, the value of the *Cond* register is always 0 and *Stall* remains at 0. When a branch instruction is detected in the decode stage, the *Stall* counter is set to 2. In the next cycle, the value of *Cond* is calculated. If it is set to 1 then the branch is taken. Otherwise, if *Cond* is set to 0, the branch is not taken. In both cases *Stall* is decremented to 1. The pipeline then stalls for one extra cycle to flush out the last incorrectly fetched instruction and *Stall* is decremented to 0. The pipeline then either resumes normal operation or, if the next instruction is a branch operation, it stalls again.

All the proof obligations generated are discharged automatically by the Rodin tool, as shown in Table 5.3.

	Total no. of proof obligations	Discharged Automatically
Abstract Model	4	4
1st Refinement	9	9
2nd Refinement	18	18
3rd Refinement	79	79
4th Refinement	76	76
5th Refinement	6	6

TABLE 5.3: Pipeline Proofs

Figure 5.18 illustrates further the operation of the concrete pipeline. Concrete events represent the transitions of the model's state. The numbered arcs are annotated with the value of *Stall* and point to the event or events that are enabled in the next pipeline state.



```
event IFIDEXWB refines IFIDEXWB
  where
    @grd1 IDop ∉ BranchOp
    @grd2 IFop ∉ BranchOp
    @grd3 Stall = 0
    @grd4 Cond = 0
    @grd5 PC < IMSIZE
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
    @act5 Regs(Rr(EXop)) ≔
          EXALUoutput
    @act6 IDop ≔  IFop
    @act7 A ≔ Regs(Ra(IFop))
    @act8 ImmV ≔ Imm(IFop)
    @act9 NPC ≔ INPC
    @act10 PC ≔ PC + 1
    @act11 INPC ≔ PC + 1
    @act12 IFop ≔ IMem(PC)
 end
```

```
event IFIDBranchEXRAWaWB refines  IFIDBranchEXRAWaWB
```

```
event IFIDBranchEXnoRAWWB refines  IFIDBranchEXnoRAWWB
  where
    @grd1 IDop ∉ BranchOp
    @grd2 IFop ∈ BranchOp
    @grd3 Stall = 0
    @grd4 Cond = 0
    @grd5 Rr(EXop) ≠ Ra(IFop) // No RAW Hazard
    @grd6 PC < IMSIZE
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
    @act5 Regs(Rr(EXop)) ≔ EXALUoutput
    @act6 IDop ≔ IFop
    @act7 Stall ≔ 2
    @act8 A ≔ Regs(Ra(IFop))
    @act9 ImmV ≔ Imm(IFop)
    @act10 NPC ≔ INPC
    @act11 PC ≔ PC + 1
    @act12 INPC ≔ PC + 1
    @act13 IFop ≔ IMem(PC)
 end
```

```
event IDStallEXBranchWB refines IDStallEXBranchWB
  where
    @grd1 IDop ∈ BranchOp // current instruction
in Instruction Memory
    @grd2 BoolOp(A ↦ 0) = 1 // BEQ0 or BNE0
    @grd6 Stall = 2
  then
    @act1 EXALUoutput ≔ NPC + ImmV
    @act2 Cond ≔ BoolOp(A ↦ 0)
    @act3 EXop ≔ IDop
    @act5 Regs(Rr(EXop)) ≔ EXALUoutput
    @act6 IDop ≔  NOP
    @act7 Stall ≔ Stall − 1
    @act8 A ≔  Regs(Ra(NOP))
    @act9 ImmV ≔ Imm(NOP)
 end
```

```
event IDStallEXNoBranchWB refines IDStallEXNoBranchWB
  where
    @grd1 IDop ∈ BranchOp // current instruction in
Instruction Memory
    @grd2 BoolOp(A ↦ 0) = 0 // BEQ0 or BNE0
    @grd3 Stall = 2
  then
    @act1 EXALUoutput ≔ NPC + ImmV
    @act2 Cond ≔ BoolOp(A ↦ 0)
    @act3 EXop ≔ IDop
    @act5 Regs(Rr(EXop)) ≔ EXALUoutput
    @act6 IDop ≔  NOP
    @act7 Stall ≔ Stall − 1
    @act8 A ≔  Regs(Ra(NOP))
    @act9 ImmV ≔ Imm(NOP)
 end
```

```
event IFBranchIDStallEXWB refines IFBranchIDStallEXWB
  where
    @grd1 IDop ∉ BranchOp
    @grd2 Stall = 1
    @grd3 Cond = 1
    @grd4 EXALUoutput ∈ 0 ‥ IMSIZE
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
    @act4 Regs(Rr(EXop)) ≔ EXALUoutput
    @act5 IDop ≔  NOP
    @act6 Stall ≔ Stall − 1
    @act7 A ≔  Regs(Ra(NOP))
    @act8 ImmV ≔ Imm(NOP)
    @act9 PC ≔ EXALUoutput
    @act10 INPC ≔ EXALUoutput
    @act11 IFop ≔ NOP
 end
```

```
event IDStallEXWB refines IDStallEXWB
  where
    @grd1 IDop ∉ BranchOp
    @grd2 Stall = 1
    @grd3 Cond = 0
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
    @act4 Regs(Rr(EXop)) ≔ EXALUoutput
    @act5 IDop ≔  NOP
    @act6 Stall ≔ Stall − 1
    @act7 A ≔  Regs(Ra(NOP))
    @act8 ImmV ≔ Imm(NOP)
 end
```

FIGURE 5.18: Fifth Refinement: Events

5.4.7 Pipeline Decomposition

The concrete representation of the branching instruction can now be decomposed into four machines, representing each of the IF, ID, EX and WB stage processes. The WB stage is represented by a single event, identical to that derived from the *ArithRR* development. The EX stage is represented by two events; one for the branching instruction,

```
Event  $EXBranch \mathrel{\widehat{=}}$
    when
        grd1 : $IDop \in BranchOp$
    then
        act1 : $EXALUoutput := NPC + ImmV$
        act2 : $Cond := BoolOp(A \mapsto 0)$
        act3 : $EXop := IDop$
    end
```

and the event that represents the behaviour of all other instructions.

```
Event  $EX \mathrel{\widehat{=}}$
    when
        grd1 : $IDop \notin BranchOp$
    then
        act1 : $EXALUoutput :\in \mathbb{Z}$
        act2 : $Cond := 0$
        act3 : $EXop := IDop$
    end
```

The EX events write to the $EXop$, $EXALUoutput$ and $Cond$ registers and read from the $IDop$, A, NPC and $ImmV$ registers.

The ID stage comprises an event for non-branching instructions, two branching events (one of which manages the data hazard) and a *stall* event.

```
Event  $ID \mathrel{\widehat{=}}$
    when
        grd1 : $IFop \notin BranchOp$
        grd2 : $Stall = 0$
    then
        act1 : $IDop := IFop$
        act2 : $A := Regs(Ra(IFop))$
        act3 : $ImmV := Imm(IFop)$
        act4 : $NPC := INPC$
    end
Event  $IDBranchNoRAW \widehat{=}$
    when
        grd1 : $IFop \in BranchOp$
        grd2 : $Stall = 0$
        grd3 : $Rr(EXop) \neq Ra(IFop)$
    then
        act1 : $IDop := IFop$
        act2 : $A := Regs(Ra(IFop))$
        act3 : $ImmV := Imm(IFop)$
        act4 : $NPC := INPC$
        act5 : $Stall := 2$
    end
Event  $IDBranchRAWa \widehat{=}$
    when
        grd1 : $IFop \in BranchOp$
        grd2 : $Stall = 0$
        grd3 : $Rr(EXop) = Ra(IFop)$
    then
        act1 : $IDop := IFop$
        act2 : $A := EXALUoutput$
        act3 : $ImmV := Imm(IFop)$
        act4 : $NPC := INPC$
        act5 : $Stall := 2$
    end
Event  $IDStall \widehat{=}$
    when
        grd1 : $IFop \in Op$
        grd2 : $Stall > 1$
    then
        act1 : $IDop := NOP$
        act2 : $A := Regs(Ra(NOP))$
        act3 : $ImmV := Imm(NOP)$
        act4 : $Stall := Stall - 1$
    end
```

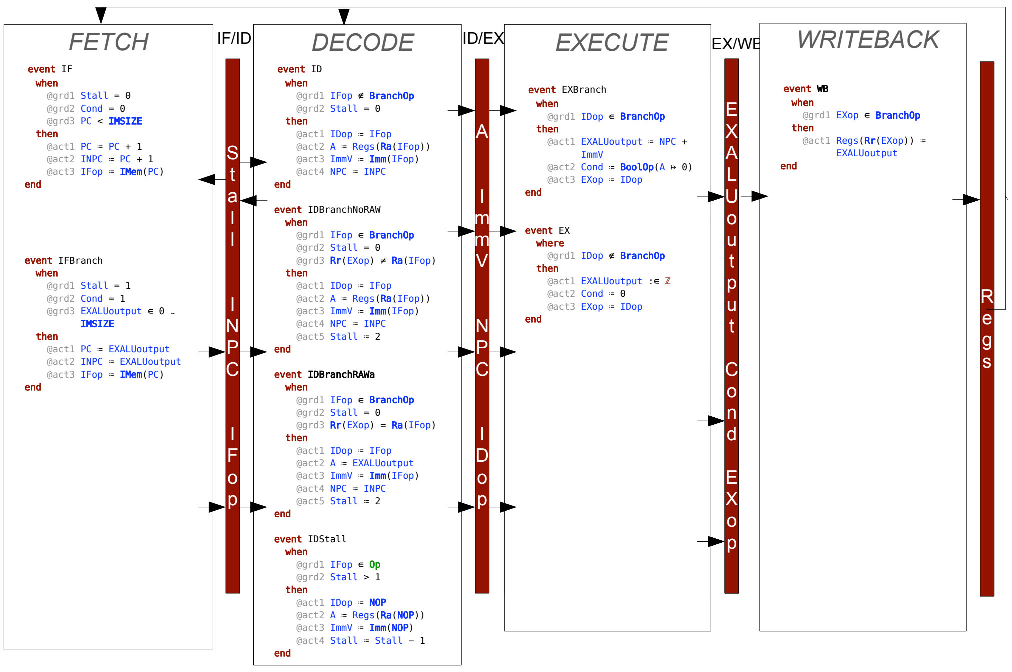
The ID events write to the $IDop$, A, NPC and $ImmV$ registers and read from the $Stall$, $IFop$ and $INPC$ registers.

The IF stage is represented by two events, which check the value of the *Cond* pipeline register and set the new value of PC accordingly.

```
Event IF $\widehat{=}$
    when
        grd1 : $Stall = 0$
        grd2 : $Cond = 0$
        grd3 : $PC < IMSIZE$
    then
        act1 : $PC := PC + 1$
        act2 : $INPC := PC + 1$
        act3 : $IFop := IMem(PC)$
    end
Event IFBranch $\widehat{=}$
    when
        grd1 : $Stall = 1$
        grd2 : $Cond = 1$
        grd3 : $EXALUoutput \in 0 .. IMSIZE$
    then
        act1 : $PC := EXALUoutput$
        act2 : $INPC := EXALUoutput$
        act3 : $IFop := IMem(PC)$
    end
```

The IF events write to the PC, $INPC$ and $IFop$ registers and read from the $EXALUoutput$ and PC registers.

All parameters have been resolved to concrete pipeline registers, and each process may be mapped directly to an RTL representation. The *DECODE* and *EXECUTE* processes comprise four and two mutually exclusive events respectively which are mapped to the branches in an RTL case statement. The final concrete pipeline representing the four processes which implement the branch instruction is shown in Figure 5.19.



FETCH

```
event IF
  when
    @grd1 Stall = 0
    @grd2 Cond = 0
    @grd3 PC < IMSIZE
  then
    @act1 PC ≔ PC + 1
    @act2 INPC ≔ PC + 1
    @act3 IFop ≔ IMem(PC)
end

event IFBranch
  when
    @grd1 Stall = 1
    @grd2 Cond = 1
    @grd3 EXALUoutput ∈ 0 ‥
          IMSIZE
  then
    @act1 PC ≔ EXALUoutput
    @act2 INPC ≔ EXALUoutput
    @act3 IFop ≔ IMem(PC)
end
```

DECODE

```
event ID
  when
    @grd1 IFop ∉ BranchOp
    @grd2 Stall = 0
  then
    @act1 IDop ≔ IFop
    @act2 A ≔ Regs(Ra(IFop))
    @act3 ImmV ≔ Imm(IFop)
    @act4 NPC ≔ INPC
end

event IDBranchNoRAW
  when
    @grd1 IFop ∈ BranchOp
    @grd2 Stall = 0
    @grd3 Rr(EXop) ≠ Ra(IFop)
  then
    @act1 IDop ≔ IFop
    @act2 A ≔ Regs(Ra(IFop))
    @act3 ImmV ≔ Imm(IFop)
    @act4 NPC ≔ INPC
    @act5 Stall ≔ 2
end

event IDBranchRAWa
  when
    @grd1 IFop ∈ BranchOp
    @grd2 Stall = 0
    @grd3 Rr(EXop) = Ra(IFop)
  then
    @act1 IDop ≔ IFop
    @act2 A ≔ EXALUoutput
    @act3 ImmV ≔ Imm(IFop)
    @act4 NPC ≔ INPC
    @act5 Stall ≔ 2
end

event IDStall
  when
    @grd1 IFop ∈ Op
    @grd2 Stall > 1
  then
    @act1 IDop ≔ NOP
    @act2 A ≔ Regs(Ra(NOP))
    @act3 ImmV ≔ Imm(NOP)
    @act4 Stall ≔ Stall − 1
end
```

EXECUTE

```
event EXBranch
  when
    @grd1 IDop ∈ BranchOp
  then
    @act1 EXALUoutput ≔ NPC +
          ImmV
    @act2 Cond ≔ BoolOp(A ↦ 0)
    @act3 EXop ≔ IDop
end

event EX
  where
    @grd1 IDop ∉ BranchOp
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
end
```

WRITEBACK

```
event WB
  when
    @grd1 EXop ∈ BranchOp
  then
    @act1 Regs(Rr(EXop)) ≔
          EXALUoutput
end
```

FIGURE 5.19: Branch 4-stage Pipeline

5.5 Pipeline Instruction Composition

The specifications of the *ArithRR* and *Branch* operations have been considered and refined separately. This loose decomposition of the problem domain has allowed the particular characteristics of these instructions to be modeled and refined independently. Further instructions from the ISA, such as the *Load* and *Store* instructions may be considered in a similar way.

Once all the instructions have been modeled, the concrete instruction implementations may be composed to derive a pipeline that supports all the instructions. The overall method that we have developed is illustrated, using the *ArithRR* and *Branch* instructions, in in Figure 5.20.

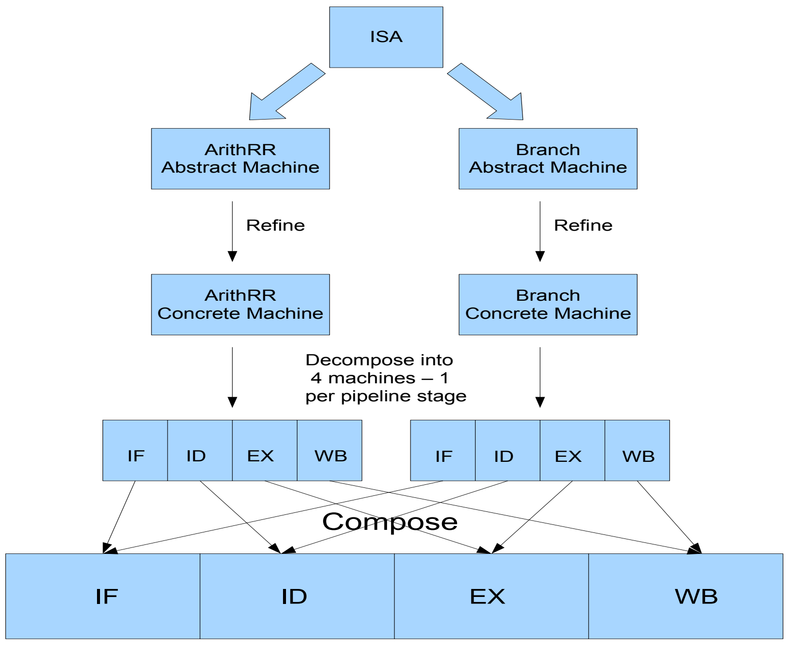
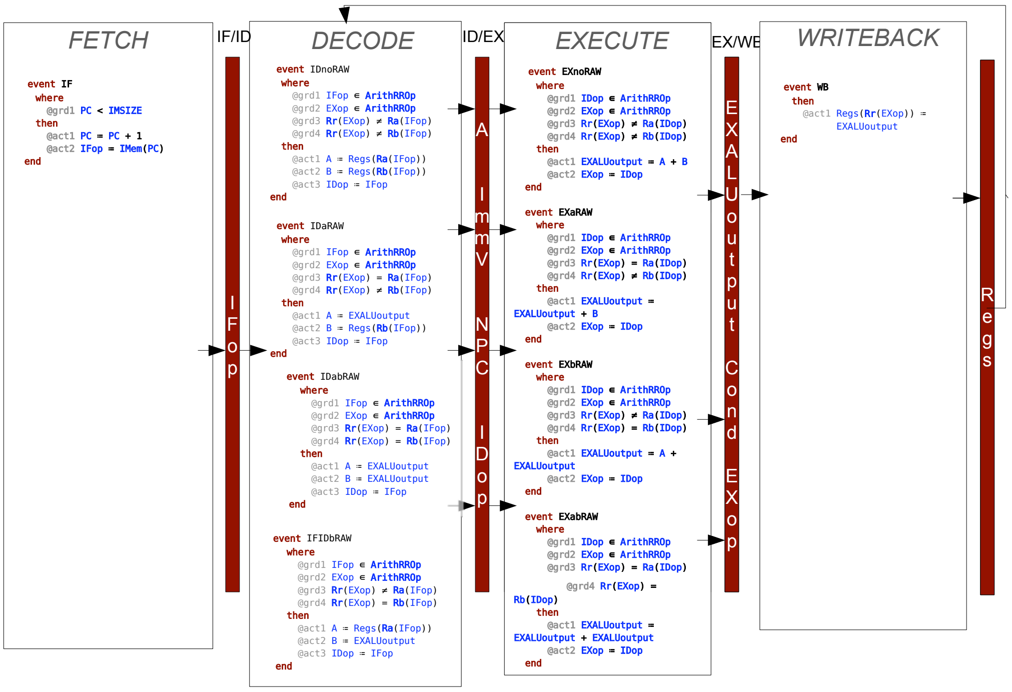


FIGURE 5.20: The Pipeline Development Flow

First, the task of implementing the ISA is decomposed, in the problem domain, into a set of tasks: one for each instruction. Event-B refinement with formal proof is then used to derive a concrete implementation, represented by a single Event-B machine, of each instruction. Shared event decomposition is then used to decompose each Event-B machine into four machines, one for each stage and representing the contribution of each instruction to each of the four pipeline stages. The contributions from each instruction to a given stage are then composed so that in the final step of the method a single, 4-stage pipeline comprising 4 processes, one per stage, is derived that represents the combined operation of the instructions.

Recall the implementations of the *ArithRR* instruction as shown in Figure 5.21 and the *Branch* instruction as shown in Figure 5.22.



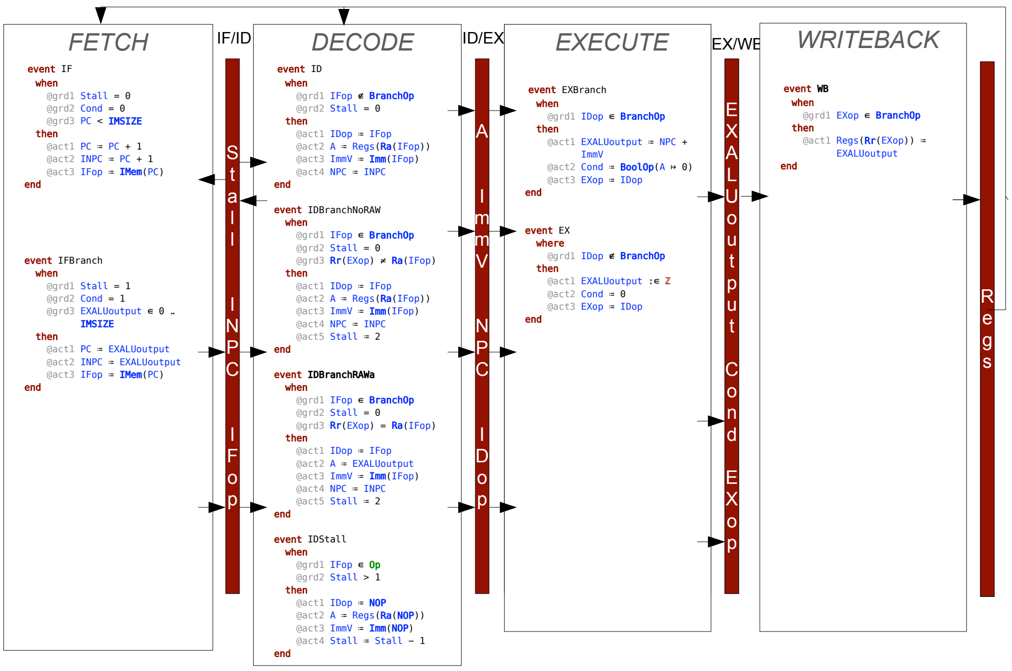
```
event IF
  where
    @grd1 PC < IMSIZE
  then
    @act1 PC ≔ PC + 1
    @act2 IFop ≔ IMem(PC)
end
```

```
event IDnoRAW
  where
    @grd1 IFop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) ≠ Ra(IFop)
    @grd4 Rr(EXop) ≠ Rb(IFop)
  then
    @act1 A ≔ Regs(Ra(IFop))
    @act2 B ≔ Regs(Rb(IFop))
    @act3 IDop ≔ IFop
end

event IDaRAW
  where
    @grd1 IFop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(IFop)
    @grd4 Rr(EXop) ≠ Rb(IFop)
  then
    @act1 A ≔ EXALUoutput
    @act2 B ≔ Regs(Rb(IFop))
    @act3 IDop ≔ IFop
end

  event IDabRAW
    where
      @grd1 IFop ∈ ArithRROp
      @grd2 EXop ∈ ArithRROp
      @grd3 Rr(EXop) = Ra(IFop)
      @grd4 Rr(EXop) = Rb(IFop)
    then
      @act1 A ≔ EXALUoutput
      @act2 B ≔ EXALUoutput
      @act3 IDop ≔ IFop
  end

event IFIDbRAW
  where
    @grd1 IFop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) ≠ Ra(IFop)
    @grd4 Rr(EXop) = Rb(IFop)
  then
    @act1 A ≔ Regs(Ra(IFop))
    @act2 B ≔ EXALUoutput
    @act3 IDop ≔ IFop
end
```

```
 event EXnoRAW
  where
    @grd1 IDop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) ≠ Ra(IDop)
    @grd4 Rr(EXop) ≠ Rb(IDop)
  then
    @act1 EXALUoutput ≔ A + B
    @act2 EXop ≔ IDop
 end

 event EXaRAW
  where
    @grd1 IDop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(IDop)
    @grd4 Rr(EXop) ≠ Rb(IDop)
  then
    @act1 EXALUoutput ≔
EXALUoutput + B
    @act2 EXop ≔ IDop
 end

 event EXbRAW
  where
    @grd1 IDop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) ≠ Ra(IDop)
    @grd4 Rr(EXop) = Rb(IDop)
  then
    @act1 EXALUoutput ≔ A +
EXALUoutput
    @act2 EXop ≔ IDop
 end

 event EXabRAW
  where
    @grd1 IDop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(IDop)

       @grd4 Rr(EXop) =
Rb(IDop)
  then
    @act1 EXALUoutput ≔
EXALUoutput + EXALUoutput
    @act2 EXop ≔ IDop
 end
```

```
event WB
  then
    @act1 Regs(Rr(EXop)) ≔
          EXALUoutput
end
```

FIGURE 5.21: ArithRR 4-stage Pipeline



```
event IF
  when
    @grd1 Stall = 0
    @grd2 Cond = 0
    @grd3 PC < IMSIZE
  then
    @act1 PC ≔ PC + 1
    @act2 INPC ≔ PC + 1
    @act3 IFop ≔ IMem(PC)
end

event IFBranch
  when
    @grd1 Stall = 1
    @grd2 Cond = 1
    @grd3 EXALUoutput ∈ 0 ‥
          IMSIZE
  then
    @act1 PC ≔ EXALUoutput
    @act2 INPC ≔ EXALUoutput
    @act3 IFop ≔ IMem(PC)
end
```

```
event ID
  when
    @grd1 IFop ∉ BranchOp
    @grd2 Stall = 0
  then
    @act1 IDop ≔ IFop
    @act2 A ≔ Regs(Ra(IFop))
    @act3 ImmV ≔ Imm(IFop)
    @act4 NPC ≔ INPC
end

event IDBranchNoRAW
  when
    @grd1 IFop ∈ BranchOp
    @grd2 Stall = 0
    @grd3 Rr(EXop) ≠ Ra(IFop)
  then
    @act1 IDop ≔ IFop
    @act2 A ≔ Regs(Ra(IFop))
    @act3 ImmV ≔ Imm(IFop)
    @act4 NPC ≔ INPC
    @act5 Stall ≔ 2
end

event IDBranchRAWa
  when
    @grd1 IFop ∈ BranchOp
    @grd2 Stall = 0
    @grd3 Rr(EXop) = Ra(IFop)
  then
    @act1 IDop ≔ IFop
    @act2 A ≔ EXALUoutput
    @act3 ImmV ≔ Imm(IFop)
    @act4 NPC ≔ INPC
    @act5 Stall ≔ 2
end

event IDStall
  when
    @grd1 IFop ∈ Op
    @grd2 Stall > 1
  then
    @act1 IDop ≔ NOP
    @act2 A ≔ Regs(Ra(NOP))
    @act3 ImmV ≔ Imm(NOP)
    @act4 Stall ≔ Stall − 1
end
```

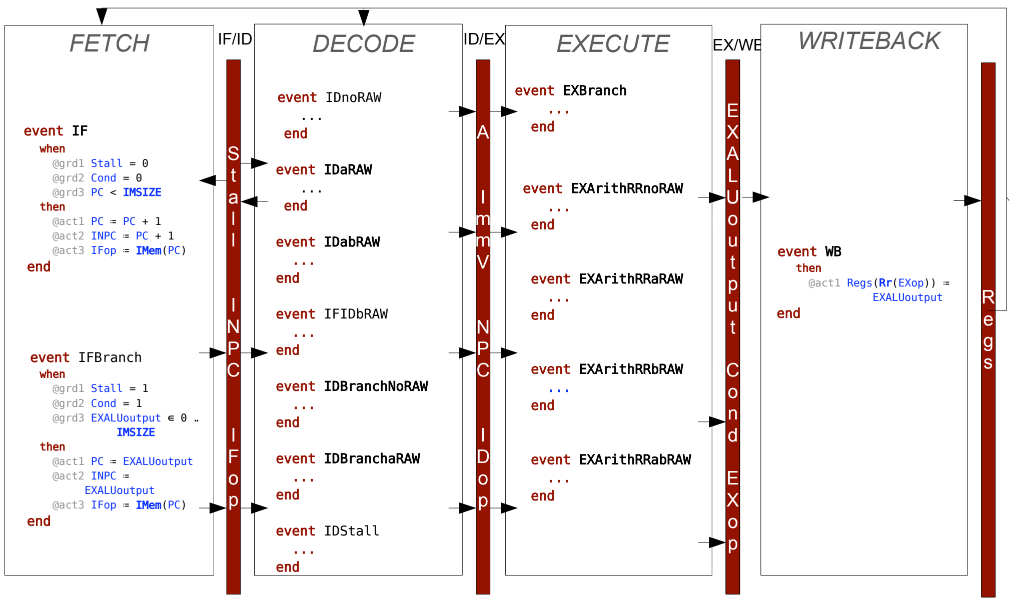
```
 event EXBranch
  when
    @grd1 IDop ∈ BranchOp
  then
    @act1 EXALUoutput ≔ NPC +
          ImmV
    @act2 Cond ≔ BoolOp(A ↦ 0)
    @act3 EXop ≔ IDop
end

event EX
  where
    @grd1 IDop ∉ BranchOp
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
end
```

```
event WB
  when
    @grd1 EXop ∈ BranchOp
  then
    @act1 Regs(Rr(EXop)) ≔
          EXALUoutput
end
```

FIGURE 5.22: Branch 4-stage Pipeline

Following the procedure described in [Silva and Butler, 2009], for each stage the initialisation events and the invariants are composed, and the variables merged. For the *FETCH* stage, the *IF* events from each instruction are composed and the *IFbranch* event, from the Branch refinement, included. For the *DECODE* stage, the four events from each instruction are included, resulting in eight mutually exclusive events. For the *EXECUTE* stage, the *EX* event from the Branch refinement is merged with the four events from the Arithmetic refinement, and the *EXBranch* included to give a total of five events. For the *WRITEBACK* stage the *WB* events from each instruction are merged, resulting in a single *WB* event in the composition.

The composed 4-stage pipeline comprises fifteen events as shown in Figure 5.23.



```
event IF
  when
    @grd1 Stall = 0
    @grd2 Cond = 0
    @grd3 PC < IMSIZE
  then
    @act1 PC ≔ PC + 1
    @act2 INPC ≔ PC + 1
    @act3 IFop ≔ IMem(PC)
end

event IFBranch
  when
    @grd1 Stall = 1
    @grd2 Cond = 1
    @grd3 EXALUoutput ∈ 0 ‥
            IMSIZE
  then
    @act1 PC ≔ EXALUoutput
    @act2 INPC ≔
          EXALUoutput
    @act3 IFop ≔ IMem(PC)
end
```

```
event IDnoRAW
  ...
end

event IDaRAW
  ...
end

event IDabRAW
  ...
end

event IFIDbRAW
  ...
end

event IDBranchNoRAW
  ...
end

event IDBranchaRAW
  ...
end

event IDStall
  ...
end
```

```
event EXBranch
  ...
end

event EXArithRRnoRAW
  ...
end

event EXArithRRaRAW
  ...
end

event EXArithRRbRAW
  ...
end

event EXArithRRabRAW
  ...
end
```

```
event WB
  then
    @act1 Regs(Rr(EXop)) ≔
              EXALUoutput
end
```

Figure 5.23: Composed 4-stage Pipeline

Each instruction of the ISA can be developed independently, addressing the particular architectural considerations of that instruction, and then composed formally with the other instructions. Each stage of the composed pipeline is represented by a set of mutually exclusive events that maps directly to an HDL description which can be RTL, Bluespec or CAL.

5.6 Measuring Pipeline Complexity at the Specification Level

The simplicity of the final pipeline is deceptive. Although the processes representing each stage are not in themselves complex, reflecting the inherent efficiency and elegance of the DLX design, the real complexity of the pipeline derives from that the fact the pipeline stage processes are running concurrently and interacting via shared variables with feedback.

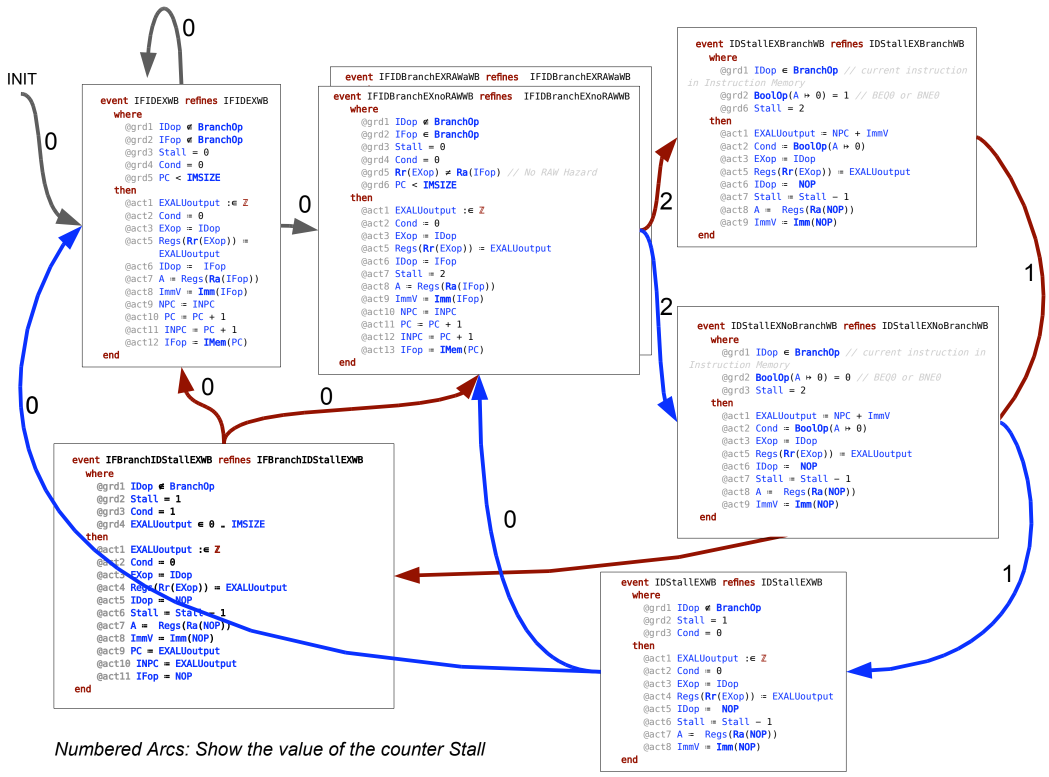
In a current RTL development flow, the implementation will be handed to a verification engineer who must verify the design against a test plan. The difficulty comes in developing a credible test plan that reflects the complexity of the design. The ISA specification on its own cannot be used as the basis for the test plan; the design engineer needs to understand the detailed pipeline implementation. In practice, the test engineer must derive the behaviour of the *combined state machine* from the individual state machines that represent each pipeline stage. Just as ensuring full code coverage of each individual process is insufficient, ensuring full arc coverage of each of the interacting state machines is also insufficient. In general, generating a combined state machine is impractical as it is extremely difficult to decide which of all the possible combined transitions are actually valid.

In the Event-B, proof-based method described, design complexity is exposed explicitly in the design process, the combined state machine is visible from an early stage of the design, the effect on complexity of design decisions can be seen immediately and design alternatives can be explored and measured. Proof-based refinement, with invariant preservation and convergence, can be seen to obviate the need for unit tests.

5.6.1 Combined State Machine Arc Coverage

Recall the concrete implementation (prior to decomposition) of the branch instruction pipeline feedback loop as shown in Figure 5.24.

The combined state machine representing the three pipeline stages and its valid transitions are revealed explicitly. The eleven arcs are labeled with the value of the counter *stall*; the verification engineer must ensure that all eleven are covered by the tests. These tests for the downstream process can therefore be derived directly from the final combined state machines revealed in this Event-B model, ensuring that full arc coverage can be targeted.



```
event IFIDEXWB refines IFIDEXWB
  where
    @grd1 IDop ∉ BranchOp
    @grd2 IFop ∉ BranchOp
    @grd3 Stall = 0
    @grd4 Cond = 0
    @grd5 PC < IMSIZE
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
    @act5 Regs(Rr(EXop)) ≔
          EXALUoutput
    @act6 IDop ≔  IFop
    @act7 A ≔ Regs(Ra(IFop))
    @act8 ImmV ≔ Imm(IFop)
    @act9 NPC ≔ INPC
    @act10 PC ≔ PC + 1
    @act11 INPC ≔ PC + 1
    @act12 IFop ≔ IMem(PC)
 end
```

```
event IFIDBranchEXRAWaWB refines  IFIDBranchEXRAWaWB
```

```
event IFIDBranchEXnoRAWWB refines  IFIDBranchEXnoRAWWB
  where
    @grd1 IDop ∉ BranchOp
    @grd2 IFop ∈ BranchOp
    @grd3 Stall = 0
    @grd4 Cond = 0
    @grd5 Rr(EXop) ≠ Ra(IFop) // No RAW Hazard
    @grd6 PC < IMSIZE
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
    @act5 Regs(Rr(EXop)) ≔ EXALUoutput
    @act6 IDop ≔ IFop
    @act7 Stall ≔ 2
    @act8 A ≔ Regs(Ra(IFop))
    @act9 ImmV ≔ Imm(IFop)
    @act10 NPC ≔ INPC
    @act11 PC ≔ PC + 1
    @act12 INPC ≔ PC + 1
    @act13 IFop ≔ IMem(PC)
 end
```

```
event IDStallEXBranchWB refines IDStallEXBranchWB
  where
    @grd1 IDop ∈ BranchOp // current instruction
in Instruction Memory
    @grd2 BoolOp(A ↦ 0) = 1 // BEQ0 or BNE0
    @grd6 Stall = 2
  then
    @act1 EXALUoutput ≔ NPC + ImmV
    @act2 Cond ≔ BoolOp(A ↦ 0)
    @act3 EXop ≔ IDop
    @act5 Regs(Rr(EXop)) ≔ EXALUoutput
    @act6 IDop ≔  NOP
    @act7 Stall ≔ Stall − 1
    @act8 A ≔  Regs(Ra(NOP))
    @act9 ImmV ≔ Imm(NOP)
 end
```

```
event IDStallEXNoBranchWB refines IDStallEXNoBranchWB
  where
    @grd1 IDop ∈ BranchOp // current instruction in
Instruction Memory
    @grd2 BoolOp(A ↦ 0) = 0 // BEQ0 or BNE0
    @grd3 Stall = 2
  then
    @act1 EXALUoutput ≔ NPC + ImmV
    @act2 Cond ≔ BoolOp(A ↦ 0)
    @act3 EXop ≔ IDop
    @act5 Regs(Rr(EXop)) ≔ EXALUoutput
    @act6 IDop ≔  NOP
    @act7 Stall ≔ Stall − 1
    @act8 A ≔  Regs(Ra(NOP))
    @act9 ImmV ≔ Imm(NOP)
 end
```

```
event IFBranchIDStallEXWB refines IFBranchIDStallEXWB
  where
    @grd1 IDop ∉ BranchOp
    @grd2 Stall = 1
    @grd3 Cond = 1
    @grd4 EXALUoutput ∈ 0 ‥ IMSIZE
  then
    @act1 EXALUoutput :∈ Z
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
    @act4 Regs(Rr(EXop)) ≔ EXALUoutput
    @act5 IDop ≔  NOP
    @act6 Stall ≔ Stall − 1
    @act7 A ≔  Regs(Ra(NOP))
    @act8 ImmV ≔ Imm(NOP)
    @act9 PC ≔ EXALUoutput
    @act10 INPC ≔ EXALUoutput
    @act11 IFop ≔ NOP
 end
```

```
event IDStallEXWB refines IDStallEXWB
  where
    @grd1 IDop ∉ BranchOp
    @grd2 Stall = 1
    @grd3 Cond = 0
  then
    @act1 EXALUoutput :∈ ℤ
    @act2 Cond ≔ 0
    @act3 EXop ≔ IDop
    @act4 Regs(Rr(EXop)) ≔ EXALUoutput
    @act5 IDop ≔  NOP
    @act6 Stall ≔ Stall − 1
    @act7 A ≔  Regs(Ra(NOP))
    @act8 ImmV ≔ Imm(NOP)
 end
```

FIGURE 5.24: Combined State Machine Arcs

5.6.2 Combined State Machine Path Coverage

Although combined state machine Arc Coverage is a desirable, but often unattainable, goal in design verification, it is in itself not sufficient to ensure full functional coverage. All valid *paths* through the combined state machine must also be covered and identifying these paths is extremely difficult. It is a major benefit of a proof-based method that there is no need to be concerned about the paths of the combined state machine; invariant preservation and convergence are proved for all possible interleavings of the composed events.

In general it is difficult to document the combined state machine with diagrams such as that shown in Figure 5.24. A better approach it to use *event refinement diagrams* [Butler, 2009] which also reveal the refinement hierarchy. The uppermost node of the diagram represents an event from the abstract machine. This node is connected to one or more events below it which represent the first refinement. These events may themselves be refined by events below them in the tree. An event refinement diagram, annotated to show the value of the *Stall* variable before (left) and after (right) the event has been evaluated, is used in Figure 5.25 to illustrate the case where a branch is taken, but no RAW hazard is incurred. The dashed line denotes that the event *EX* refines *skip*. The solid lines denote that, for instance, *EXNoBranch* refines *NoBranch*. The *XOR* denotes that one and only one of the events below it are enabled at a time.

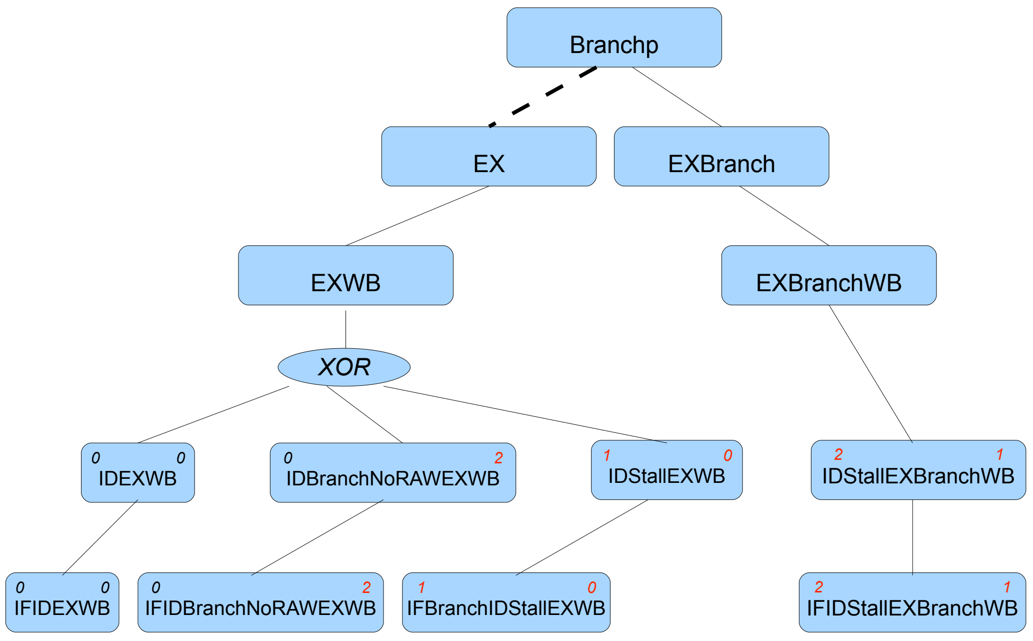


Figure 5.25: Branch Refinement Diagram: 4th Refinement, No RAW

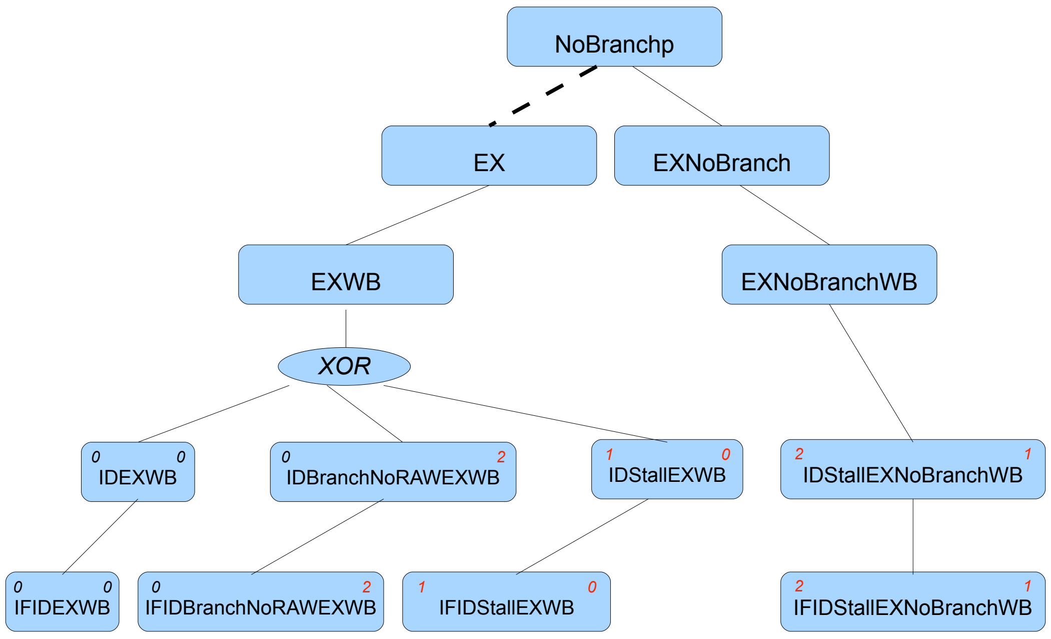


Figure 5.26: NoBranch Refinement Diagram: 4th Refinement, No RAW

At the fourth level of refinement it can be seen that the pipeline, in normal operation and in the absence of branch instructions, executes the event *IFIDEXWB*. The value of the *Stall* remains at zero. When a branch instruction is encountered, the event *IFIDBranchNoRAWEXWB* is enabled and *Stall* is set to 2 to indicate that the pipeline will now stall. This enables the event *IFIDStallEXBranchWB* which determines that the branch will be taken and decrements *Stall*. *IFBranchIDStallEXWB* is now enabled to complete the pipeline stall and decrement the value of the counter to 0 so that normal operation can resume. Figure 5.26 documents the case where there is no hazard and the branch is not taken.

A total of four such diagrams, to reflect the combination of cases where a RAW hazard is encountered or not and where the branch is taken or not, document concisely the complexity of the combined state machine, its transitions and their associated convergence.

5.7 Component Re-use in Pipeline Specifications

Once the key concepts of pipeline specification in this method have been developed, to manage for instance stalling and forwarding, it is desirable that the efforts of this specification work can be re-used in subsequent pipeline designs. Exploring further the component-based composition described in the specification of the Even Number Generator in Section 4.4.6 on Page 53, the *ArithRR* instruction is revisited to show how specification components representing a pipeline stage can be introduced into the refinement.

Recall the abstract model of the *ArithRR* instruction represented by the event

```
Event  $ArithRR \mathrel{\widehat{=}}$
    any
        $pop$
    where
        grd1 : $pop \in ArithRROp$
    then
        act1 : $Regs(Rr(pop)) := Regs(Ra(pop)) + Regs(Rb(pop))$
    end
```

with the microarchitecture shown in Figure 5.27.

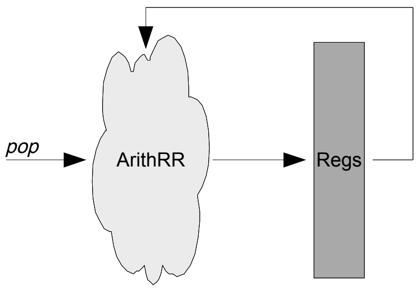
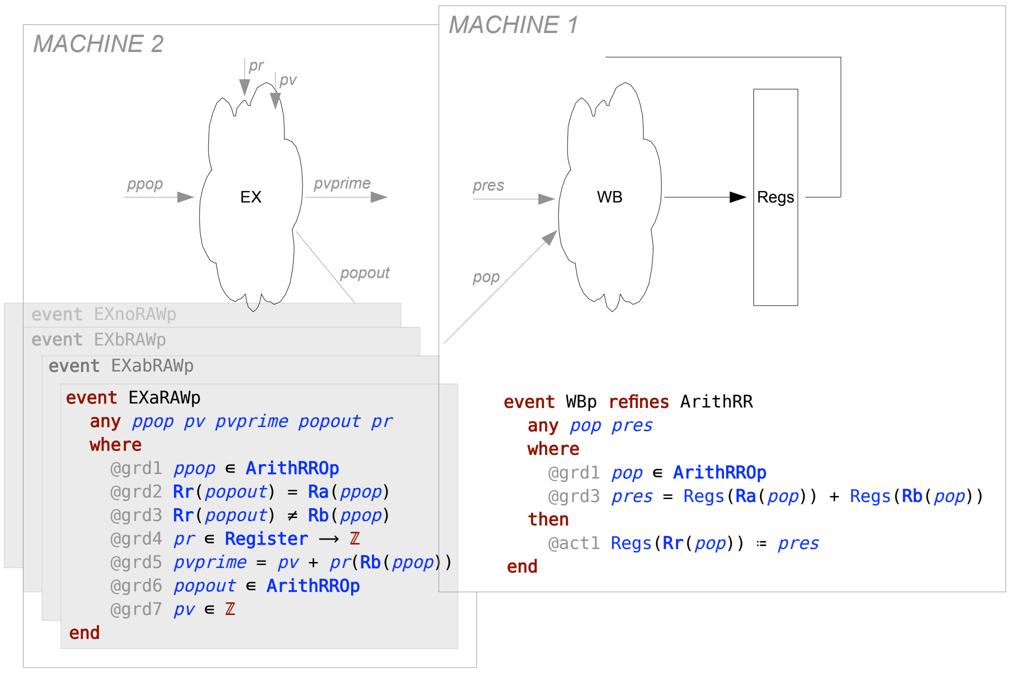


FIGURE 5.27: Abstract Machine: Microarchitecture

Now consider the two component specifications of Figure 5.28.

The two *machines* representing each of these abstract specifications can be composed formally to create a single machine that must be proved to be a correct refinement of the original abstract specification. The abstract machine is refined by *Machine 1.* The event



```
event EXnoRAWp
event EXbRAWp
event EXabRAWp
event EXaRAWp
  any ppop pv pvprime popout pr
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 Rr(popout) = Ra(ppop)
    @grd3 Rr(popout) ≠ Rb(ppop)
    @grd4 pr ∈ Register → ℤ
    @grd5 pvprime = pv + pr(Rb(ppop))
    @grd6 popout ∈ ArithRROp
    @grd7 pv ∈ ℤ
end
```

```
event WBp refines ArithRR
  any pop pres
  where
    @grd1 pop ∈ ArithRROp
    @grd3 pres = Regs(Ra(pop)) + Regs(Rb(pop))
  then
    @act1 Regs(Rr(pop)) ≔ pres
end
```

FIGURE 5.28: First Refinement: Components

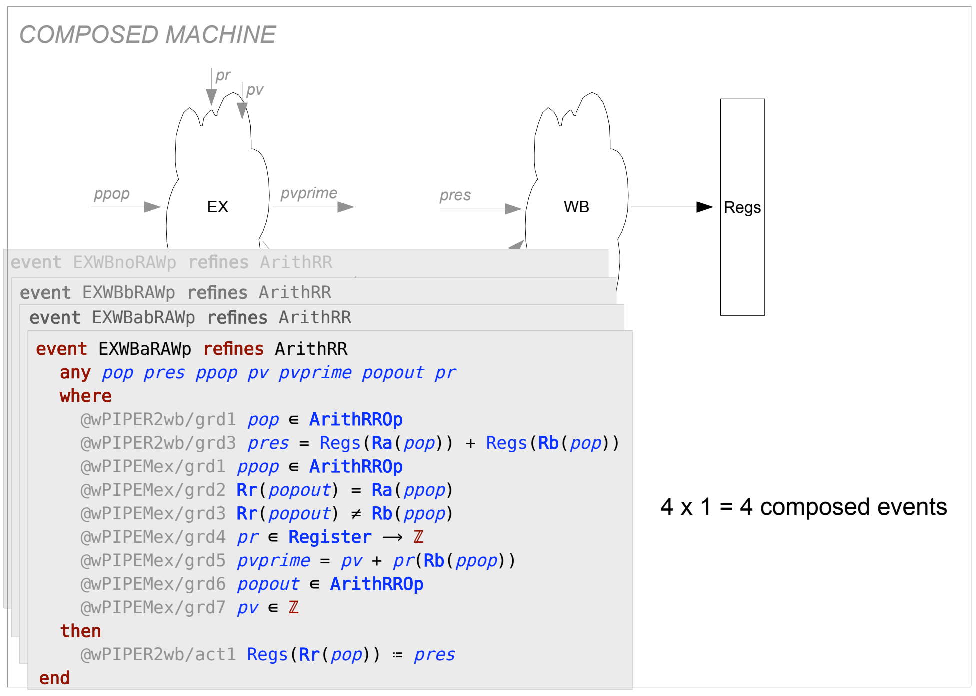
representing the WriteBack stage, *WBp*, refines the abstract event *ArithRR* and the four events *EX*RAWp* refine *skip*. The initial composition, using Rodin's composition plug-in, results in four composed events as show in Figure 5.29.

In a further refinement, the variables *EXALUoutput* and *EXop* are now introduced, the parameters of the composed machine are *instantiated* using witnesses and the gluing invariant introduced as shown in Figure 5.30. The proof obligations are discharged automatically.

The composition process is repeated, introducing the parameterised specification of the Instruction Decode stage as shown in Figure 5.31. Four Instruction Decode events are composed with the four events from the previous refinement to give a total of sixteen events.

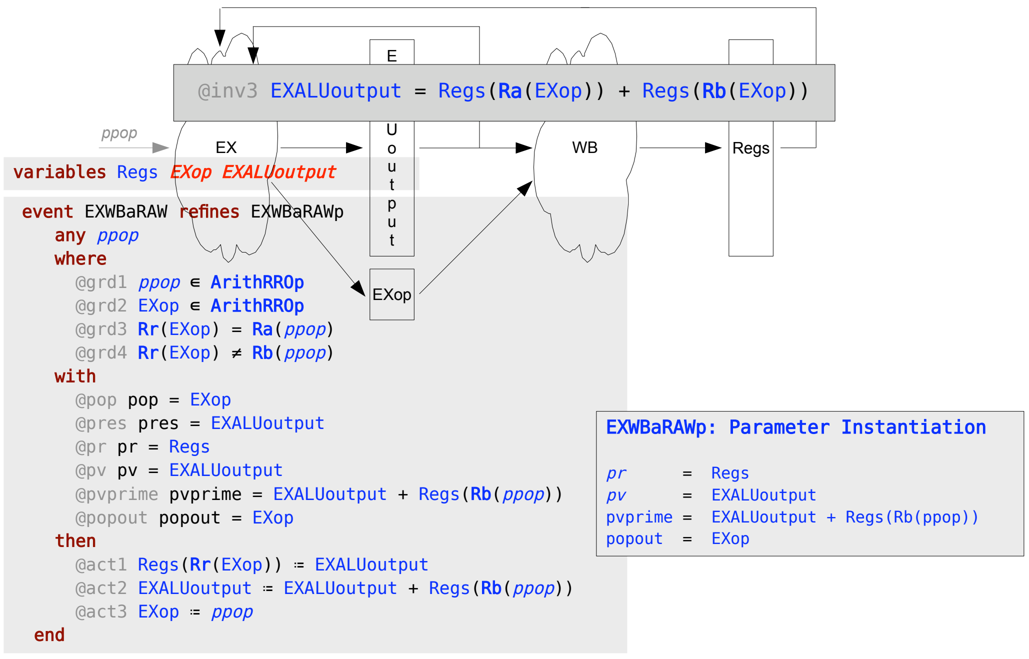
In the third refinement the variables *A*, *B* and *EXop* are introduced, the parameters of the composed machine are again instantiated using witnesses and the gluing invariants introduced as shown in Figure 5.32. Again, the proof obligations are discharged automatically.

Now that the full feedback loop of the pipeline has been modelled and proved, the events can be de-composed so that there is a machine to represent each pipeline stage, in the same way as described for the *ArithRR* instruction in Section 5.3.6 on Page 85.



```
event EXWBnoRAWp refines ArithRR
event EXWBbRAWp refines ArithRR
event EXWBabRAWp refines ArithRR
event EXWBaRAWp refines ArithRR
  any pop pres ppop pv pvprime popout pr
  where
    @wPIPER2wb/grd1 pop ∈ ArithRROp
    @wPIPER2wb/grd3 pres = Regs(Ra(pop)) + Regs(Rb(pop))
    @wPIPEMex/grd1 ppop ∈ ArithRROp
    @wPIPEMex/grd2 Rr(popout) = Ra(ppop)
    @wPIPEMex/grd3 Rr(popout) ≠ Rb(ppop)
    @wPIPEMex/grd4 pr ∈ Register ⟶ ℤ
    @wPIPEMex/grd5 pvprime = pv + pr(Rb(ppop))
    @wPIPEMex/grd6 popout ∈ ArithRROp
    @wPIPEMex/grd7 pv ∈ ℤ
  then
    @wPIPER2wb/act1 Regs(Rr(pop)) ≔ pres
end
```

FIGURE 5.29: First Refinement: Component Composition



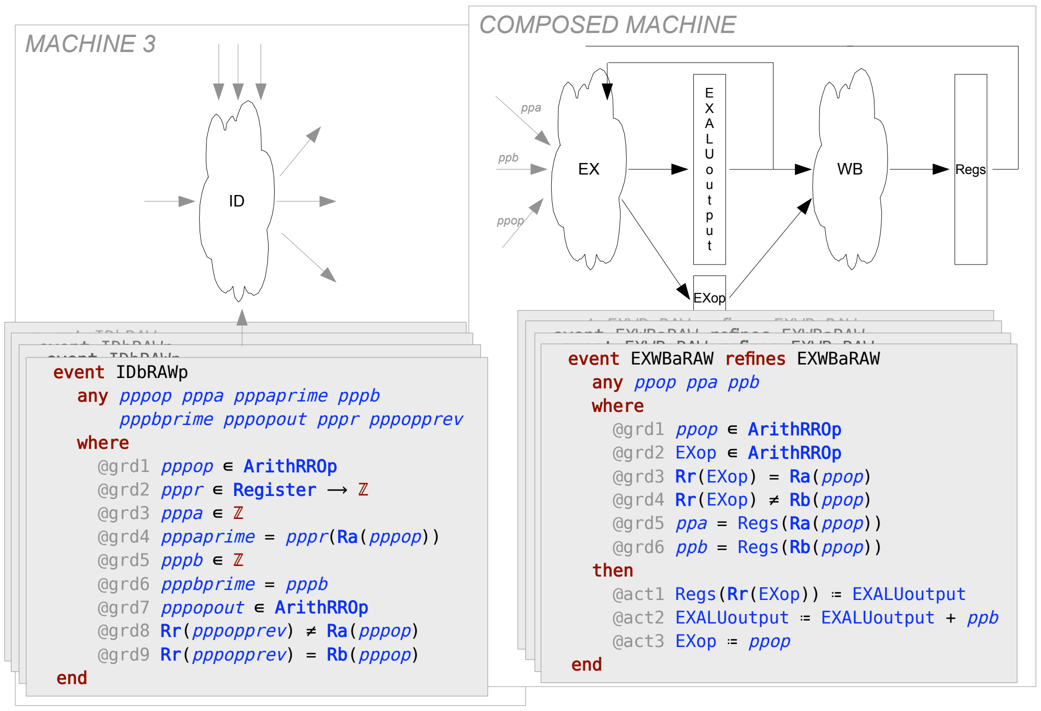
```
@inv3 EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop))
```

```
variables Regs EXop EXALUoutput
```

```
event EXWBaRAW refines EXWBaRAWp
  any ppop
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(ppop)
    @grd4 Rr(EXop) ≠ Rb(ppop)
  with
    @pop pop = EXop
    @pres pres = EXALUoutput
    @pr pr = Regs
    @pv pv = EXALUoutput
    @pvprime pvprime = EXALUoutput + Regs(Rb(ppop))
    @popout popout = EXop
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 EXALUoutput ≔ EXALUoutput + Regs(Rb(ppop))
    @act3 EXop ≔ ppop
end
```

```
EXWBaRAWp: Parameter Instantiation

pr       =  Regs
pv       =  EXALUoutput
pvprime  =  EXALUoutput + Regs(Rb(ppop))
popout   =  EXop
```

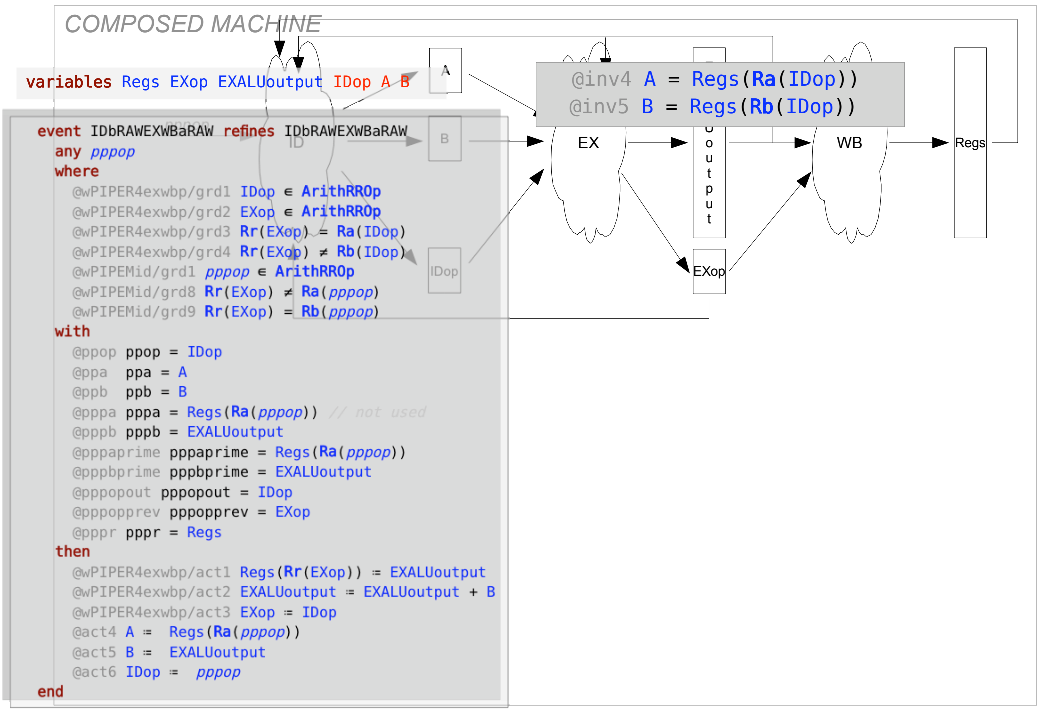
FIGURE 5.30: Second Refinement: Parameter Instantiation



```
event IDbRAWp
  any pppop pppa pppaprime pppb
      pppbprime pppopout pppr pppopprev
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 pppr ∈ Register ⟶ ℤ
    @grd3 pppa ∈ ℤ
    @grd4 pppaprime = pppr(Ra(pppop))
    @grd5 pppb ∈ ℤ
    @grd6 pppbprime = pppb
    @grd7 pppopout ∈ ArithRROp
    @grd8 Rr(pppopprev) ≠ Ra(pppop)
    @grd9 Rr(pppopprev) = Rb(pppop)
 end
```

```
event EXWBaRAW refines EXWBaRAW
  any ppop ppa ppb
  where
    @grd1 ppop ∈ ArithRROp
    @grd2 EXop ∈ ArithRROp
    @grd3 Rr(EXop) = Ra(ppop)
    @grd4 Rr(EXop) ≠ Rb(ppop)
    @grd5 ppa = Regs(Ra(ppop))
    @grd6 ppb = Regs(Rb(ppop))
  then
    @act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @act2 EXALUoutput ≔ EXALUoutput + ppb
    @act3 EXop ≔ ppop
end
```

FIGURE 5.31: Third Refinement: Component Composition



```
variables Regs EXop EXALUoutput IDop A B
```

```
@inv4 A = Regs(Ra(IDop))
@inv5 B = Regs(Rb(IDop))
```

```
event IDbRAWEXWBaRAW refines IDbRAWEXWBaRAW
  any pppop
  where
    @wPIPER4exwbp/grd1 IDop ∈ ArithRROp
    @wPIPER4exwbp/grd2 EXop ∈ ArithRROp
    @wPIPER4exwbp/grd3 Rr(EXop) = Ra(IDop)
    @wPIPER4exwbp/grd4 Rr(EXop) ≠ Rb(IDop)
    @wPIPEMid/grd1 pppop ∈ ArithRROp
    @wPIPEMid/grd8 Rr(EXop) ≠ Ra(pppop)
    @wPIPEMid/grd9 Rr(EXop) = Rb(pppop)
  with
    @ppop ppop = IDop
    @ppa  ppa = A
    @ppb  ppb = B
    @pppa pppa = Regs(Ra(pppop)) // not used
    @pppb pppb = EXALUoutput
    @pppaprime pppaprime = Regs(Ra(pppop))
    @pppbprime pppbprime = EXALUoutput
    @pppopout pppopout = IDop
    @pppopprev pppopprev = EXop
    @pppr pppr = Regs
  then
    @wPIPER4exwbp/act1 Regs(Rr(EXop)) ≔ EXALUoutput
    @wPIPER4exwbp/act2 EXALUoutput ≔ EXALUoutput + B
    @wPIPER4exwbp/act3 EXop ≔ IDop
    @act4 A ≔  Regs(Ra(pppop))
    @act5 B ≔  EXALUoutput
    @act6 IDop ≔   pppop
end
```

FIGURE 5.32: Third Refinement: Parameter Instantiation

5.7.1 Parameters and Witnesses in Component Re-use

Witnesses, described in Section 2.9 on Page 19 were introduced into Event-B to make proof easier. They do also, however, provide a powerful mechanism for importing re-usable event *templates* into an Event-B development. In our method, a re-usable component has *outputs* that are concrete variables and *inputs* that are parameters. These parameters represent the *environment* within which the component is guaranteed to implement its specified behaviour, represented by Event-B *invariants*, in terms of its output variables. The parameterized environment can be represented in the *guards* of the events in a powerful and flexible way, which can broaden considerably the scope for re-use of that component by defining a *family* of behaviours which meets the component's input specification.

When a component is introduced into an Event-B refinement, its output *variables* are bound using witnesses to the *parameters* of the refinement and its input *parameters* are bound, again using witnesses, to the *variables* that already exist or are freshly introduced at this refinement level. In this way, communication is established between the component and the existing model. Alternatively, the parameters of the component may be left to be bound in a subsequent refinement; the parameters become part of the environment specification of the model. The developer does not need to modify the component, only to provide the witnesses that constrain the parameters to the *particular* variables of the development. The Rodin tool proves that the variables meet the parameterized specification.

It is a fundamental requirement of a re-use method that the developer should not have to modify a component to meet a particular need. Event-B parameters and witnesses provide a mechanism that meets this requirement.

5.8 A Review of the Pipeline Development Method

Even for a simple and elegant pipeline architecture such as DLX it was a daunting proposition to attempt to derive, formally, a correct, concrete implementation from the processor ISA. It was therefore necessary to decompose the *problem*. The first issue that needed to be addressed was whether it was actually possible to *model* the ISA and the derived pipeline in Event-B. This issue splits into two; the modelling of *data* and the modelling of *control*.

5.8.1 Modelling Data

The key observation here was that the data needed to be only *concrete enough* to be amenable to high-level or RTL synthesis. It is one of the major strengths of modern synthesis techniques that they relieve the designer of the need to worry about the low-level data representation. Event-B types, supported by the structuring conventions that facilitate the representation of *records* was found to be sufficient to represent the instructions and registers of the processor.

5.8.2 Modelling Control

The next step was to ensure that each concrete pipeline stage operation could be modelled with Event-B events; to show a direct mapping between the specified operations and these events was an important step in validating the approach. At this stage, however, it became clear that the processor behaviour could not be modelled as a straightforward interleaving of these events and that it would be necessary to model explicitly the inherent simultaneity of the pipeline.

5.8.3 Advancing from Modelling to Proving: Problem Decomposition

Now it was possible, using multiple refinement steps, to derive a concrete implementation of the pipeline, but not to *prove* that the implementation implemented its specification. It was therefore important to decompose the problem further and consider a single instruction at a time. The Arithmetic instruction was chosen first and a *strong* abstract specification sought. In this case the specification of the effect that the instruction has on the processor register file, for a notional abstract machine that executes in a single cycle, was settled upon. It was then possible to refine this to a two-stage abstract machine with the introduction of just two pipeline variables and focus on using the tool to help discover the appropriate gluing invariants. At this stage either *stalling* or *forwarding* could have been chosen to address the RAW hazards shown up by the failing proof obligations, but since these proof obligations pointed directly to the forwarding solution, this was chosen.

The Branch instruction was then chosen so that the pipeline stalling technique could be developed. Again, a strong abstract specification, which defined the effect of the instruction on the program counter of an abstract single-cycle machine, was chosen.

5.8.4 Automatic Proof

It is highly desirable for a proof-based method that the proof-obligations are discharged automatically without manual user intervention. Where proofs cannot be discharged, it is not always obvious to the designer whether the problem is with the model or the tool. It is therefore important that the developer of the *method* ensures that automatic proof is feasible, and it was not considered an option in our work to require manual proof intervention. In retrospect, striving for automatic proof had other major benefits, not least in ensuring that each refinement step represented a clear, understandable increment addressing a single modelling issue.

5.8.5 Managing Architectural Complexity

A further benefit of taking small, incremental steps is that any increase is complexity is noticed and can be managed immediately. The method ensures that all the simultaneous pipeline activity must be addressed at every refinement step and makes it easy to explore alternative architectures; the designer can move back up the refinement hierarchy, as far as it is necessary, to chose an alternative refinement approach. Event-B de-composition and re-composition have also been shown to be invaluable in managing pipeline complexity.

5.8.6 Measuring Architectural Complexity

An interesting by-product of our work has been the establishment of a relationship between formal proof and *functional coverage*; in particular combined state machine arc and path coverage. During the development of the pipeline, *faults* were deliberately injected into the models by, for instance, deliberately changing the value of a variable and then re-running the provers. Sensitivity to faults is a measure of the *strength* of the formal model and if a variable can take faulty values without affecting the proofs, then that variable can also be considered not to be *covered.* The relationship between model strength and coverage is an interesting area for further research.

5.8.7 Memory Accesses

The DLX microarchitecture assumes that memory *loads* and *stores* occur in a single cycle. Although straightforward to model, this does not reflect the actual memory access characteristics of an SoC microprocessor. A more representative memory access example is considered in the next chapter.

Chapter 6

Memory Accesses: Managing Component Latency

In modern SoC designs a large area of the SoC can be devoted to on-chip memory and memory accesses for Synchronous Dynamic RAM (SDRAM) can take several clock cycles, although it is possible to make a read request on every clock cycle. In addition, where a microprocessor architecture is capable of issuing multiple instructions in a single cycle which can complete in an arbitrary order it must be shown that the re-order buffer maintains the original issue order.

The contribution of this chapter is to extend the method that has been developed in earlier chapters to model RAM latency and manage out-of-order completion. Modeling SDRAM, in the context of an Internet Protocol (IP) address look-up module, is the subject of detailed analysis in [Arvind et al., 2004] and the same example is used here to demonstrate the Event-B modelling techniques that we have developed in the extension of the method.

6.0.8 Modelling Synchronous Memory

For a synchronous RAM with a latency of N cycles, an auxiliary N-*bit* shift register is introduced to keep track of the valid memory accesses, as shown in Figure 6.1.

On every clock cycle, if a read request is initiated, then a $'1'$ is written to the shift register. Otherwise, if there is no read request, then a $'0'$ is written. At the same time, on every clock cycle, the last bit of the shift register is read. If it is a $'1'$, then the value on the output data line of the RAM represents the valid result of a read request made N cycles earlier, and this result can be written to the output FIFO. If it is a $'0'$, the result is ignored.

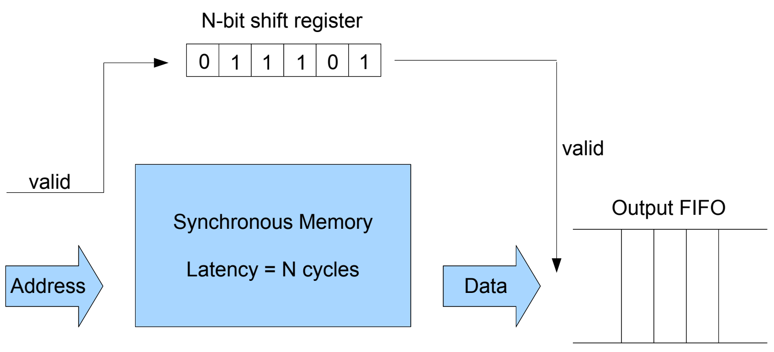


FIGURE 6.1: Managing RAM Latency

To model the memory latency in Event-B, a partial function *latency table* is introduced that maps a unique token representing a memory read to a constant natural number that represents the number of clock cycles that will elapse before the data associated with the memory read will be ready.

inv1 : $latency\_table \in \mathbb{N} \nrightarrow \mathbb{N}$

inv2 : $last\_token \in \mathbb{N}$

The variable *last_token* is used to ensure that the next, incremental token value is associated with each new lookup.

A further table maps the unique token to the memory address.

inv3 : $address\_table \in \mathbb{N} \nrightarrow Address$

The SDRAM maps an address to a value,

inv4 : $RAM \in Address \nrightarrow Value$

and the output FIFO maps a natural number to a *Value* and is accessed using a write and read index, *wr* and *rd* respectively.

inv5 : $output\_fifo \in \mathbb{N} \nrightarrow Value$

The microarchitecture is shown in Figure 6.2.

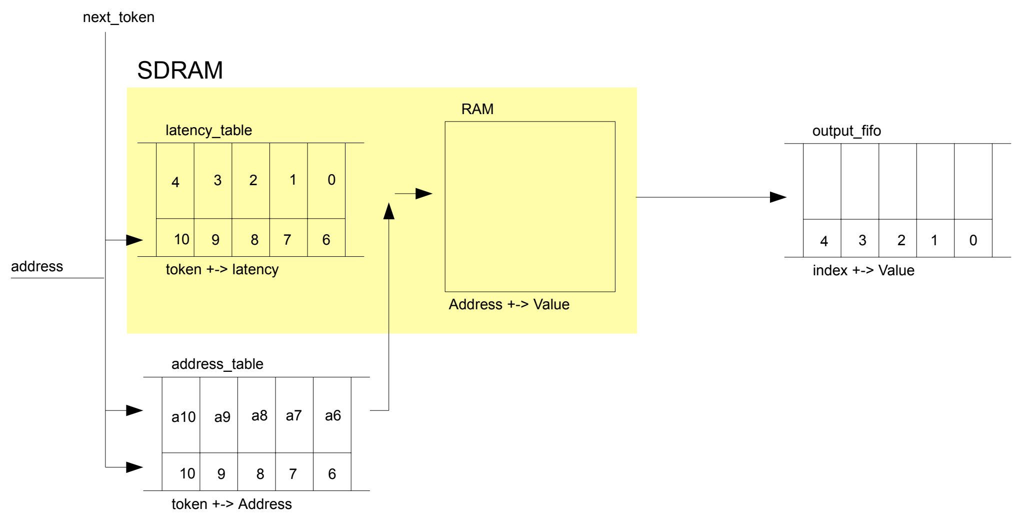


Figure 6.2: SDRAM lookup

When a read request is initiated, a new entry for the associated read address is added to the latency table with the natural number set to the memory latency. At the same time, a lambda function is used to decrement the value associated with any previous read address in the table,

act2 : $latency\_table := (\lambda i \cdot i \in dom(latency\_table) \wedge latency\_table(i) > 0 \mid latency\_table(i) - 1) \lhd\!\!- \{tok \mapsto Latency\}$

where *tok* is the token associated with the latest read request and *Latency* is the SDRAM latency in clock cycles.

If at a clock cycle no read request is initiated, then the values associated with outstanding read requests must be still be decremented.

act2 : $latency\_table := (\lambda i \cdot i \in dom(latency\_table) \wedge latency\_table(i) > 0 \mid latency\_table(i) - 1)$

When the value associated with an outstanding read request reaches zero, then the look-up can complete.

```
Event  CompleteLookup $\widehat{=}$
    any
        tok
    where
        grd1 : $tok \in dom(latency\_table) \wedge latency\_table(tok) = 0$
    then
        act1 : $output\_fifo(wr) := RAM(address\_table(tok))$
    end
```

The token *tok* is used to find the corresponding RAM address in the address table and the value at that address is writen to the output FIFO.

This method for modeling latency is now used in an Event-B model of an IP look-up pipeline.

6.0.9 The IP Look-up Circular Pipeline

The IP look-up module in a router is responsible for conducting a *Longest Prefix Match* of an IP address with the entries in a look-up table. (An IP address is a sequence of numbers separated by dots.) The look-up table has a sparse tree representation in memory and therefore multiple memory reads may be needed to get a match. To maximise throughput, a circular pipeline is employed so that a memory read request can be made at each clock cycle. The number of memory reads required will vary according to the makeup of the IP address and an out-of-order completion buffer is therefore needed to ensure that the results are made available in the correct sequence. The architecture of the IP Address Lookup model is shown in Figure 6.3.

When an IP address is passed to the look-up module, it is associated with a numerical token that denotes its place in the look-up queue. The first sub-address is placed in the address table and then used as an index to the top-level node of the table in memory. The latency table ensures that the result of the look-up is correctly synchronised. If the sub-address points to a leaf, then the look-up result is placed in the completion buffer at the location denoted by the IP address token. Otherwise, the reference to the sub-tree is re-circulated and the next sub-address read and used as an index into the sub-tree. This continues until a leaf is reached or it is found that the address is invalid. The entry in the completion buffer is updated accordingly and the next IP address is processed.

The model performs three fundamental actions: *enter*, *recirculate* and *completelookup*. Since this model is pipelined, it is also necessary to consider the simultaneous actions as described by the state machine of Table 6.1. The action *enter* is enabled when a new IP address is presented to the module

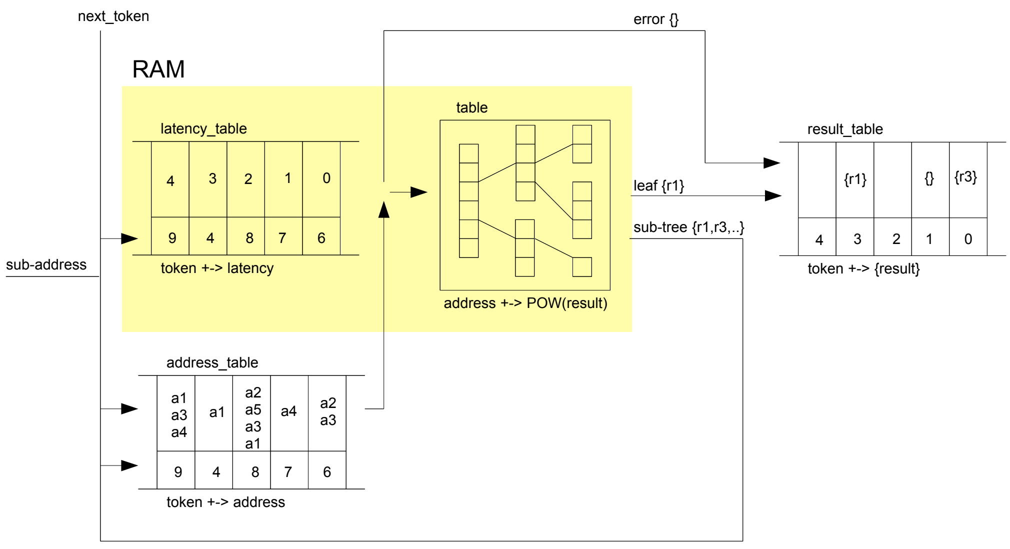


Figure 6.3: IP Lookup Circular Pipeline

Event *enter* $\widehat{=}$

refines *enter*

any

$tok, subaddr$

where

grd1 : $tok \in \mathbb{N}$

grd2 : $subaddr \in Subaddress$

grd3 : $tok = last\_token + 1$

grd6 : $table\_in\_place = TRUE$

grd7 : $tok \notin dom(latency\_table)$

grd8 : $\forall x \cdot x \in dom(latency\_table) \Rightarrow latency\_table(x) > 0$

then

act1 : $address\_table := address\_table \mathbin{\lhd\!\!-} \{tok \mapsto \{0 \mapsto subaddr\}\}$

act2 : $last\_token := tok$

act3 : $latency\_table := (\lambda i \cdot i \in dom(latency\_table) \wedge latency\_table(i) > 0 |$
$latency\_table(i) - 1) \mathbin{\lhd\!\!-} \{tok \mapsto 4\}$

end

The address table maps a token to an ordered list of IP sub-addresses.

`inv1` : $address\_table \in \mathbb{N} \rightarrowtail\!\!\!\!\!\!+ (\mathbb{N} \rightarrowtail\!\!\!\!\!\!+ Subaddress)$

The new token is given the next free value,

`grd3` : $tok = last\_token + 1$

a guard ensures that no memory access is completing in this cycle,

`grd8` : $\forall x \cdot x \in dom(latency\_table) \Rightarrow latency\_table(x) > 0$

the token is associated with the first sub-address of the IP address and added to the address table,

`act1` : $address\_table := address\_table \mathbin{\lhd\!\!\!-} \{tok \mapsto \{0 \mapsto subaddr\}\}$

and the token is associated with the RAM latency (in this model the value is 4 cycles) while, simultaneously, all other values in the latency table are decremented by one.

`act3` : $latency\_table := (\lambda i \cdot i \in dom(latency\_table) \wedge latency\_table(i) > 0 |$
$latency\_table(i) - 1) \mathbin{\lhd\!\!\!-} \{tok \mapsto 4\}$

It is also necessary to model the case where there is no address lookup pending. The action *idle* simply decrements the values in the latency table to ensure that the latency is correctly modelled.

Event *idle* $\widehat{=}$

refines *idle*

when

`grd1` : $table\_in\_place = TRUE$

`grd3` : $\forall x \cdot x \in dom(latency\_table) \Rightarrow latency\_table(x) > 0$

then

`act1` : $latency\_table := \lambda i \cdot i \in dom(latency\_table) \wedge latency\_table(i) > 0 |$
$latency\_table(i) - 1$

end

The action *completelookup* is enabled when a result is available to put in the completion buffer. It is composed with either an *idle* or *enter* action. For instance,

Event $enter\_complete\_lookup \;\widehat{=}$

refines *enter*

any

$tok, subaddr, ctok$

where

grd1 : $tok \in \mathbb{N}$

grd2 : $subaddr \in Subaddress$

grd4 : $ctok \in \mathbb{N}$

grd5 : $tok = last\_token + 1$

grd8 : $table\_in\_place = TRUE$

grd9 : $tok \notin dom(latency\_table)$

grd10 : $ctok \in dom(latency\_table)$

grd11 : $ctok \in dom(address\_table)$

grd13 : $address\_table(ctok) \in dom(table)$

grd14 : $latency\_table(ctok) = 0$

grd15 : $card(table(address\_table(ctok))) = 1$

then

act1 : $address\_table := address\_table \Leftarrow \{tok \mapsto \{0 \mapsto subaddr\}\}$

act2 : $last\_token := tok$

act3 : $latency\_table := (\lambda i \cdot i \in dom(latency\_table) \wedge latency\_table(i) > 0 | latency\_table(i) - 1) \Leftarrow \{tok \mapsto 4\}$

act4 : $result\_table := result\_table \Leftarrow \{ctok \mapsto table(address\_table(ctok))\}$

end

In this composed event *tok* is the *entering* token and *ctok* is the *completing* token.

The value in the latency table for the token has reached zero,

grd14 : $latency\_table(ctok) = 0$

and therefore the result table can be updated.

act4 : $result\_table := result\_table \Leftarrow \{ctok \mapsto table(address\_table(ctok))\}$

The completing token *ctok* is used to get the sub-address from the address table, this sub-address is then used to get the result from the look-up table *table* and the result is written to the completion buffer *result_table*.

An extra action, *completelookuperr* is added to deal with invalid addresses, again composed with either an *idle* or *enter* action. For instance,

```
Event  idle_complete_lookup_err ≙
refines  idle
    any
        ctok
    where
        grd1 : $ctok \in \mathbb{N}$
        grd2 : $table\_in\_place = TRUE$
        grd3 : $ctok \in dom(latency\_table)$
        grd4 : $ctok \in dom(address\_table)$
        grd6 : $address\_table(ctok) \notin dom(table)$
        grd7 : $latency\_table(ctok) = 0$
    then
        act1 : $result\_table := result\_table \mathbin{<\!\!+} \{ctok \mapsto \varnothing\}$
        act2 : $latency\_table := \lambda i \cdot i \in dom(latency\_table) \land latency\_table(i) > 0 |$
              $latency\_table(i) - 1$
    end
```

The event *recirculate* is enabled when the result of the lookup is not a leaf. In this case, the reference to the sub-tree is re-circulated and the next sub-address read and used as an index into the sub-tree.

```
Event  recirculate ≙
refines  recirculate
    any
        tok, subaddr, next
    where
        grd1 : $table\_in\_place = TRUE$
        grd2 : $tok \in \mathbb{N}$
        grd4 : $tok \in dom(latency\_table)$
```

grd6 : $tok \in dom(address\_table)$

grd7 : $address\_table(tok) \in dom(table)$

grd3 : $latency\_table(tok) = 0$

grd5 : $card(table(address\_table(tok))) > 1$

grd8 : $subaddr \in Subaddress$

grd9 : $next \in \mathbb{N}$

grd10 : $next = card(dom(address\_table(tok)))$

then

act1 : $address\_table := address\_table \Leftarrow \{tok \mapsto address\_table(tok) \Leftarrow \{next \mapsto subaddr\}\}$

act2 : $latency\_table := (\lambda i \cdot i \in dom(latency\_table) \land latency\_table(i) > 0 | latency\_table(i) - 1) \Leftarrow \{tok \mapsto 4\}$

act3 : $result\_table := result\_table \Leftarrow \{tok \mapsto table(address\_table(tok))\}$

end

Current State				**Action**		**Next State**		
Res	Leaf	Valid				Res	Leaf	Valid
F	$*$	$*$	$\rightarrow$	*idle/enter*	$\rightarrow$	T/F	T/F	T/F
T	F	T	$\rightarrow$	*recirculate*	$\rightarrow$	T/F	T/F	T/F
T	$*$	F	$\rightarrow$	*idle/enter* $\|$ *completelookuperr*	$\rightarrow$	T/F	T/F	T/F
T	T	T	$\rightarrow$	*idle/enter* $\|$ *completelookup*	$\rightarrow$	T/F	T/F	T/F

TABLE 6.1: IP Lookup State Machine

Row 1: If the result (indicated by the boolean *Res*) of a memory lookup is not yet available due to memory latency, or there are no lookups pending, then the model will either *idle* if there is no pending lookup or *enter* the next pending IP address. (Lookup of the next address can proceed even while an earlier address is still being processed.)

Row 2: When the result of the memory lookup becomes available and a leaf node has not been encountered, precedence is given to this lookup in progress over pending new lookups and the action *recirculate* is enabled.

Row 3: If the memory lookup fails because the address is invalid, then an empty result is written to the completion buffer and, simultaneously, the next address can be processed if there is one pending.

Row 4: When the result of the memory lookup becomes available and a leaf is encountered, the valid result is written to the completion buffer and, simultaneously, the next address can be processed if there is one pending.

The model is developed with three refinement stages and all proof obligations are discharged automatically as shown in in Table 6.2.

	Total no. of proof obligations	Discharged Automatically	Discharged Manually	Not Discharged
Abstract Model	28	28	0	0
First Refinement	7	7	0	0
Second Refinement	11	11	0	0
Third Refinement	21	21	0	0

Table 6.2: IP Lookup Proofs

This section has demonstrated how Event-B can be used to model the fixed latency of synchronous RAM in a hardware component. It has also demonstrated how multiple memory accesses can be managed simultaneously and how a re-order buffer can be introduced to ensure that the look-up results are processed in the correct order. These techniques can be transferred to the modelling of *super-scalar* SoC pipelined microprocessors.

For applications where latency is variable, a *latency-insenstitive* approach provides an effective solution. Modeling latency-insensitive protocols is the topic of Chapter 7.

Chapter 7

Developing SoC Sub-Systems

This chapter covers the emerging latency-insensitive protocols which can de-couple sub-system design from the complex timing interaction that can occur between components and the implications of latency-insensitive design for an Event-B based method.

The contribution described in this chapter is the development of a high-level representation in Event-B of a low-level, latency-insensitive protocol that can be used at the specification level to verify that a sub-system of components is a correct refinement of its abstract specification, that each component conforms to the protocol and that the sub-system does not deadlock.

7.1 Managing Design Complexity

Once a methodology is in place for the specification, refinement and architectural exploration of SoC components, the "consequent challenge is addressing the communication and synchronization issues that arise while assembling predesigned components." [Carloni and Sangiovanni-Vincentelli, 2002]

The requirements of an SoC inter-component communication mechanism are many-fold. First, it must address the issue of latency which has an increasing impact on SoC design methods. Second, it must decouple the components so that function and communication can be considered as *orthogonal concerns* [Keutzer et al., 2000]. Increases in component complexity result in a corresponding exponential increase in verification complexity. Third, it must work with existing tool flows which are predominantly synchronous, and fourth it should not in itself impose unreasonable performance and power-consumption overheads.

7.2 Asynchronous Design and Transaction-Level Modelling

Introducing uni-directional FIFOs as a communication mechanism between components addresses both latency and complexity concerns. This mechanism forms the foundation of the SystemC Transaction Level Modeling methodology [Ghenassia, 2006].

Strictly enforcing FIFO-based communication ensures that there are no shared control variables and the only way that communication between components can occur is through *message-passing*. However, to ensure that the functionality of a sub-system depends only on the order in which messages are received, and not on their timing, it is essential that every component is *stallable* [Carloni and Sangiovanni-Vincentelli, 2002]; if an input FIFO is empty or an output FIFO is full, then a component must retain its internal state until it is possible for communication to recommence. The price of the reduced complexity and independence from latency issues that FIFO-based communication exacts is that it must be verified that the components can stall correctly and, in addition, that the stalling mechanism cannot lead to inadvertent sub-system deadlock or livelock. This additional verification requirement is better addressed using formal methods rather than simulation.

Although an exponential merging of events is required for each individual pipelined component, as shown in earlier chapters, the events of different components do not need to be merged. The FIFO de-coupling means that the order in which the events of different components are activated does not matter, and the interleaving semantics of Event-B are precisely those that are required. So, for a 10-stage pipeline represented by 3 events per stage, the total number of potential combined events that must be considered is $3^{10} = 59049$. If the pipeline register between the fifth and sixth stage is replaced with a FIFO, splitting the pipeline in two, the total number of events that must be considered is

$$3^5 + 3^5 = 243 + 243 = 486$$

Splitting a pipeline to manage latency issues has a commensurate effect on model complexity, with the proviso that each component of the pipeline must be shown to be stallable.

Although FIFO-based communication addresses the issues of latency and complexity, the asynchronous nature of the communication means that it doesn't fit well within synchronous tool chains. This has the knock-on effect that the synthesised output cannot be as efficient, in terms of performance and power consumption, as for a purely synchronous design since it is much more difficult to optimise across component boundaries.

FIFO-based communication is a mechanism, not a method. It imposes the burden that it diverges from the well-established synchronous design methods and is more suitable for communication between large sub-systems which may operate in different clock domains than for communication within sub-systems.

7.3 Unit-Transaction Level Design

Unit-Transaction Level design (UTL) [Asanovic, 2007], builds on the notion of FIFO-based communication with the constraint that each component must be a *transactor* (*trans*actional *actor*) which has private local state, buffered uni-directional input and output channels and a set of *guarded atomic actions* (transactions) that can read from its input channels, modify local state and write to its output channels. A scheduler is associated with each transactor which determines the next transaction to be executed, although this scheduling function could be implemented implicitly within the transactions themselves. A transaction is chosen non-deterministically for execution from the set of ready transactions.

The UTL method has been designed expressly for use with Bluespec high-level synthesis, but with one important constraint. Individual Bluespec descriptions are written for each transactor; the object-oriented mechanism that Bluespec provides for hierarchical composition is not used and is replaced by explicit FIFO-based communication. Transactors are constrained in size according to the guidelines recommended in [Sylvester and Keutzer, 2001] so that conventional logic synthesis can be used on the RTL output from Bluespec for each transactor in turn.

7.4 Latency-Insensitive Design

The theory of latency-insensitive design [Carloni et al., 2001], [Carloni et al., 1999] is proposed as "the foundation of a correct-by-construction methodology for SoC design" [Carloni and Sangiovanni-Vincentelli, 2002]. Latency-insensitive design introduces the notion of a communication shell which wraps each synchronous, stallable component with a set of buffers that manage point-to-point, uni-directional data links between the components of a sub-system. The behaviour of a sub-system composed of components that conform to the latency-insensitive protocol is independent of the communication latencies between the components.

It is an important benefit of this approach that the sub-system itself is also synchronous. In an asynchronous system, the delay between two successive messages is arbitrary. In a latency-insensitive system, this delay is constrained to be a multiple of the clock period. Latency insensitive design therefore exhibits the beneficial properties of asynchronous design, but with the considerable advantage that, because all inter-component communication is synchronous, it can be used in a well-established synchronous design tool flow, with all the advantages of the optimised synthesis facilities that such a flow provides.

For each input of a stallable component the wrapper implements 3 registers as shown in Figure 7.1.

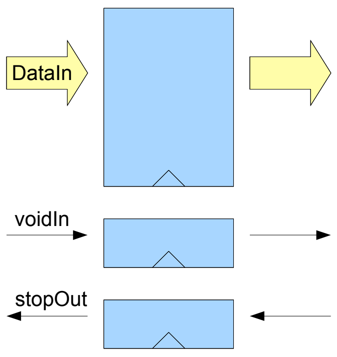


FIGURE 7.1: Latency Insensitive Protocol

The *DataIn* register holds the input data. The single bit *voidIn* register takes the value $'1'$ if the input data is valid and $'0'$ if not. The single bit *stopOut* register is set to $'1'$ by the receiving component if it is stalled and $'0'$ otherwise. A corresponding set of 3 registers is implemented for each output. If a component can be stalled with a gated clock then the wrapper can be generated automatically.

Where the point-to-point communication between two components cannot meet the clock period constraint, then one or more *relay stations* may be inserted without affecting the sub-system functionality. Relay stations are simple, stallable components which comply with the latency-insensitive protocol and simply hold the data value passed to them for a clock period before passing it on. A point-to-point path with relay stations acts as a pipeline and therefore can minimise the effect on throughput.

7.5 Synchronous Elastic Design

Synchronous ELastic Flow (SELF) [Krstic et al., 2006], [Cortadella et al., 2006b], [Cortadella et al., 2006a] builds on the theory of latency-insensitive design by defining an highly efficient communication protocol that imposes a very low overhead in terms of area, performance and power consumption. The protocol is shown in Figure 7.2.

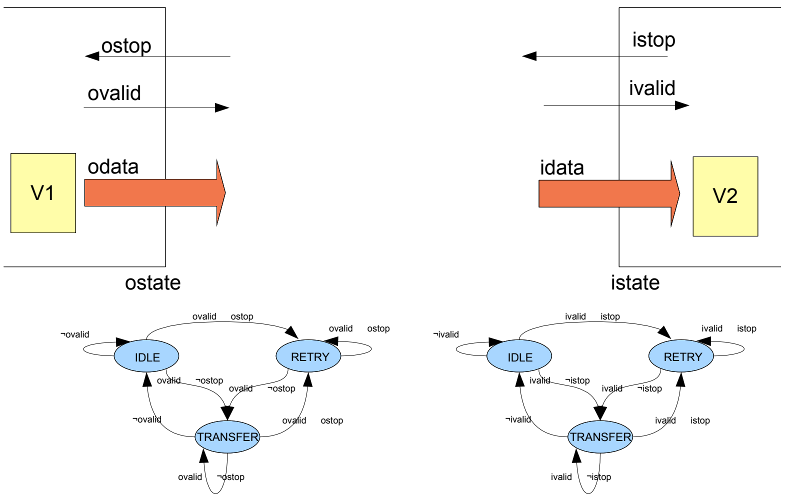


FIGURE 7.2: The SELF Protocol

Each *elastic component* must be stallable and implement the SELF protocol on each of its inputs and outputs. Two control signals, *valid* and *stop*, determine the three possible states that an output or input channel can take as shown in Table 7.1.

State	Signal Conditions	Actions
TRANSFER	$valid \wedge \neg stop$	receiver accepts valid data from sender
IDLE	$\neg valid$	sender has no valid data
RETRY	$valid \wedge stop$	valid data from sender is not accepted by receiver

TABLE 7.1: SELF State Machine

Two elastic components are connected with an *elastic buffer* as shown in Figure 7.3.

The buffer is essentially a 2-entry FIF0, which can be implemented using 2 latches, and a simple mechanism for propagating the values of *valid* and *stop* between the components.

Synchronous Elastic Design combines the de-coupling benefits of asynchronous design with the efficiency that can be achieved using synchronous tool flows and is therefore a strong candidate for use in SoC sub-system design.

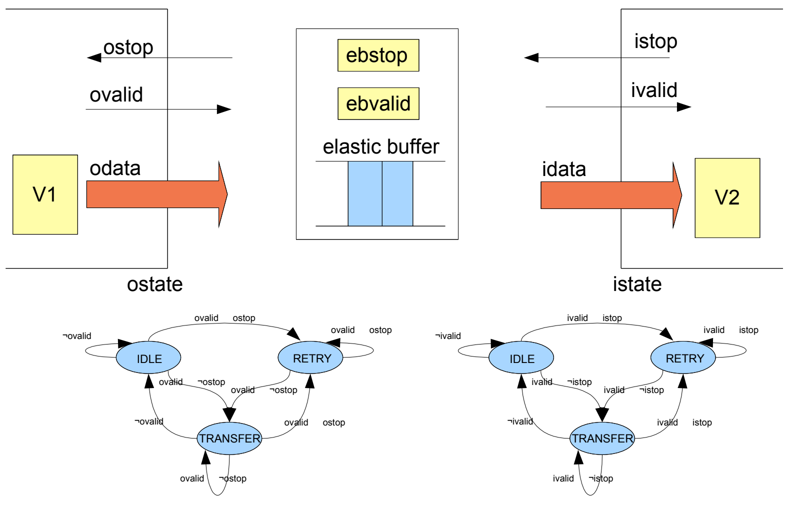


FIGURE 7.3: Connecting 2 Elastic Components

7.6 Microarchitectural Exploration: Introducing Synchronous Elastic Buffering with Event-B

If a component design is seen to be getting too complex and potential timing issues arise, the microarchitectural designer will need to consider breaking the component into sub-components. By introducing synchronous elastic buffers to replace pipeline registers, the designer is able to address the timing issues in a way that de-couples the functionality from the timing.

We present a method, illustrated by examples, that enables the designer to prove that an alternative, buffered sub-system design is functionally equivalent to the original component design which used shared register communication.

In the first example, we show how the output of the *Execute* stage in a microprocessor pipeline may be buffered instead of written directly to pipeline registers. Two refinements of the abstract pipeline are shown; one which introduces registers in the traditional way and another which introduces synchronous elastic buffers instead of the registers. The Rodin tool is then used to prove that, provided that appropriate forwarding mechanisms are introduced, both microarchitectural alternatives are correct refinements of the abstract pipeline. The use of forwarding with synchronous buffering is a valuable technique because it allows the designer to manage, for instance, arithmetic operations that take a variable number of cycles to complete but with the performance benefits that access to the forwarded value can provide.

In the second example, synchronous elastic buffering is used to implement a distributed stalling mechanism in a microprocessor pipeline which provides an efficient alternative to forwarding that allows the pipeline implementation to sustain higher clock rates.

7.6.1 The Synchronous Elastic Buffer: An Abstract Specification in Event-B

[Cortadella et al., 2006c] presents an abstract model of a synchronous elastic buffer, based on an unbounded FIFO B, indexed by two variables wr and rd as shown in in Figure 7.4.

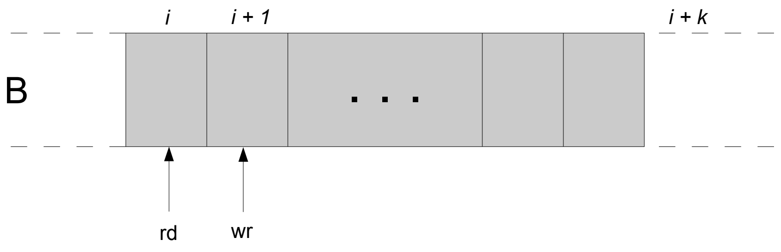


FIGURE 7.4: Abstract Synchronous Elastic Buffer

B can be modelled in Event-B as

inv1 : $B \in \mathbb{N} \rightarrow Op$

inv2 : $rd \in \mathbb{N}$

inv3 : $wr \in \mathbb{N}$

The value to buffered, in this case a microprocessor opcode is indexed by two natural numbers, the read index rd and the write index wr. Values enter the buffer at the location pointed to by the write index and are read from the buffer at the location pointed to by the read index. Since the synchronous elastic buffer is of size two,

inv4 : $wr = rd + 1$

and the variables are initialised thus.

act1 : $B := \mathbb{N} \times \{NOP\}$

act2 : $rd := 0$

act3 : $wr := 1$

[Cortadella et al., 2006c] uses the DLX pipeline to illustrate how elasticity can be introduced into microarchitectural design. Here, the *ArithRR* instruction

```
Event  $ArithRR \mathrel{\widehat{=}}$
    any
        $pop$
    where
        grd1 : $pop \in ArithRROp$
    then
        act1 : $Regs(Rr(pop)) := Regs(Ra(pop)) + Regs(Rb(pop))$
    end
```

will be used to explore two alternative pipeline implementations with synchronous buffering to manage RAW hazards. The microarchitecture of the abstract machine is shown in Figure 7.5.

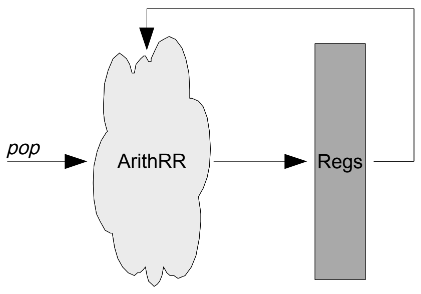


FIGURE 7.5: Abstract Machine: Microarchitecture

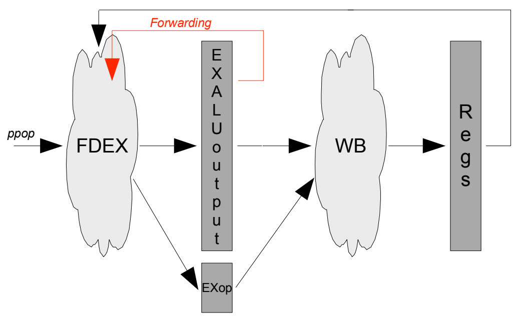


FIGURE 7.6: Refined Machine with forwarding: Microarchitecture

7.6.2 Synchronous Buffering with Forwarding

Recall the first refinement of the *ArithRR* model with the microarchitecture shown in Figure 7.6.

In this alternative refinement of the abstract model, the registers *EXALUoutput* and *EXop* are replaced with the synchronous elastic buffers *vbuf* and *obuf* respectively as shown in Figure 7.7.

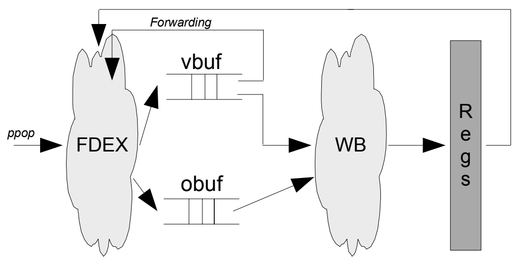


FIGURE 7.7: Refined Machine with buffers and forwarding: Microarchitecture

Both buffers have read indices (*vrd* and *ord* respectively) and write indices (*vwr* and *owr* respectively) which are all incremented at each event execution to ensure synchronous operation. The combined event, for instance, that deals with a RAW hazard on the source register *Ra*

Event $EXWBaRAW \mathrel{\widehat{=}}$

refines $ArithRR$

 any

  $ppop$

 where

  grd1 : $obuf(ord) \in ArithRROp$

  grd2 : $ppop \in ArithRROp$

  grd3 : $Rr(obuf(ord)) = Ra(ppop)$

  grd4 : $Rr(obuf(ord)) \neq Rb(ppop)$

 with

  pop : $\mathtt{pop} = \mathtt{obuf}(\mathtt{ord})$

 then

  act1 : $Regs(Rr(obuf(ord))) := vbuf(vrd)$

  act2 : $ord := owr$

  act3 : $vrd := vwr$

  act4 : $vbuf(vwr) := vbuf(vrd) + Regs(Rb(ppop))$

  act5 : $obuf(owr) := ppop$

  act6 : $vwr := vwr + 1$

  act7 : $owr := owr + 1$

 end

can be compared with the same event in the register-based refinement,

Event $EXWBaRAW \mathrel{\widehat{=}}$

refines $ArithRR$

 any

  $ppop$

 where

  grd1 : $EXop \in ArithRROp$

  grd2 : $ppop \in ArithRROp$

  grd3 : $Rr(EXop) = Ra(ppop)$

  grd4 : $Rr(EXop) \neq Rb(ppop)$

 with

  pop : $\mathtt{pop} = \mathtt{EXop}$

```
then
    act1 : $Regs(Rr(EXop)) := EXALUoutput$
    act2 : $EXALUoutput := EXALUoutput + Regs(Rb(ppop))$
    act3 : $EXop := ppop$
end
```

The new gluing invariant is

inv9 : $vbuf(vrd) = Regs(Ra(obuf(ord))) + Regs(Rb(obuf(ord)))$

compared with the original

inv3 : $EXALUoutput = Regs(Ra(EXop)) + Regs(Rb(EXop))$

and all proof obligations are discharged automatically. In this case, 88 proof obligations are discharged compared with the 33 in the original, register-based solution.

7.6.3 Synchronous Elastic Buffering with Stalling

Introducing synchronous elastic buffering allows the designer to take advantage of the elastic buffering protocol and implement stalling in a distributed manner, obviating the need for a centralised stall counter. When a RAW hazard is detected for instance, instead of using forwarding, the *Valid* control signals are used effectively to stall the pipeline for a single cycle. A *bubble* of invalid data is flushed out through the pipeline and then normal operation can resume.

In order to implement this stalling mechanism it is not only necessary to introduce buffering between the *Execute* and *Writeback* stages but also buffering in a backwards direction within the feedback loop which writes values back to the register file; if values travelling in a forward direction are stalled because of a hazard, then the delay caused must be compensated for when the values are written back. The stalling protocol, described below, ensures that writing back to the register file is correctly synchronised in this presence of RAW harards.

The tradeoff is the cost of an extra cycle when a hazard is encountered against the extra forwarding logic complexity, but with the added advantage that the pipeline will be able to sustain higher clock frequencies because of the shorter signal paths introduced by buffering the write back path. The micro-architecture is shown in Figure 7.8.

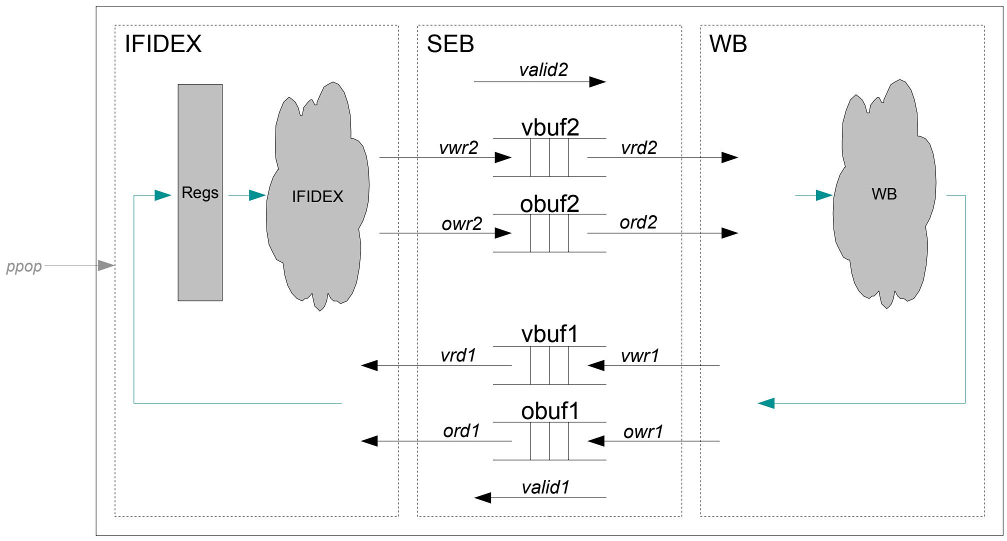


FIGURE 7.8: Refined Machine with synchronous elastic buffers: Microarchitecture

In this refinement of the abstract ISA specification, events are introduced which refine the abstract event *ArithRR*. The actions that update the synchronous buffer indices,

act6 : $ord1 := owr1$

act7 : $vrd1 := vwr1$

act8 : $vwr1 := vwr1 + 1$

act9 : $owr1 := owr1 + 1$

act10 : $ord2 := owr2$

act11 : $vrd2 := vwr2$

act12 : $vwr2 := vwr2 + 1$

act13 : $owr2 := owr2 + 1$

are common to all the events and are omitted for clarity.

Under normal operation, in the absence of hazards, *Valid1* and *Valid2* are *TRUE* and the event *EXWMBnoRAW* is enabled. The guards grd5 : $Rr(obuf1(ord1)) \neq Ra(pppop)$

grd6 : $Rr(obuf1(ord1)) \neq Rb(pppop)$

check that the target register *Rr* of the instruction buffered in the *Writeback* stage is not about to overwrite either of the source registers, *Ra* and *Rb*, of the incoming instruction *pppop* and the guards

grd7 : $Rr(obuf2(ord2)) \neq Ra(pppop)$

grd8 : $Rr(obuf2(ord2)) \neq Rb(pppop)$

check that the source registers, *Ra* and *Rb*, of the incoming instruction *pppop* do not need the result stored in the target register *Rr* of the instruction buffered in the *Execute* stage.

Event *EXWBnoRAW* $\widehat{=}$

refines *ArithRR*

any

pppop

where

grd1 : $pppop \in ArithRROp$

grd2 : $obuf1(ord1) \in ArithRROp$

grd3 : $obuf2(ord2) \in ArithRROp$

grd4 : $Valid1 = TRUE$

grd5 : $Rr(obuf1(ord1)) \neq Ra(pppop)$

grd6 : $Rr(obuf1(ord1)) \neq Rb(pppop)$

grd7 : $Rr(obuf2(ord2)) \neq Ra(pppop)$

grd8 : $Rr(obuf2(ord2)) \neq Rb(pppop)$

grd9 : $Valid2 = TRUE$

with

pop : $\mathtt{pop} = \mathtt{obuf1(ord1)}$

then

act1 : $Regs(Rr(obuf1(ord1))) := vbuf1(vrd1)$

act2 : $obuf1(owr1) := obuf2(ord2)$

act3 : $vbuf1(vwr1) := vbuf2(vrd2)$

act4 : $vbuf2(vwr2) := Regs(Ra(pppop)) + Regs(Rb(pppop))$

```
        act5 : $obuf2(owr2) := pppop$
            ... //Update buffer indices
    end
```

The control signals *Valid1* and *Valid2* remain *TRUE*, the values of the source registers *Ra* and *Rb* are added together and written to the buffer *vbuf2*, the result from the previous cycle, stored in buffer *vbuf2*, is written to the buffer *vbuf1*, the result from the cycle before that, stored in buffer *vbuf1* is written back to the Register File and all the buffer read and write indexes are incremented.

To illustrate the stalling mechanism, consider the case where a RAW hazard is encountered on register *Ra* of the new instruction *pppop* because the pipeline is about write a value back to that location, as detected by the guard

grd5 : $(Rr(obuf1(ord1)) = Ra(pppop))$

of the event *EXWBRAWa*

```
Event  $EXWBRAWa \widehat{=}$
refines  $ArithRR$
    any
        $pppop$
    where
        grd1 : $pppop \in ArithRROp$
        grd2 : $obuf1(ord1) \in ArithRROp$
        grd3 : $obuf2(ord2) \in ArithRROp$
        grd4 : $Valid1 = TRUE$
        grd5 : $Rr(obuf1(ord1)) = Ra(pppop)$
        grd6 : $Rr(obuf1(ord1)) \neq Rb(pppop)$
        grd7 : $Rr(obuf2(ord2)) \neq Ra(pppop)$
        grd8 : $Rr(obuf2(ord2)) \neq Rb(pppop)$
        grd9 : $Valid2 = TRUE$
    with
        pop : pop = obuf1(ord1)
    then
        act1 : $Regs(Rr(obuf1(ord1))) := vbuf1(vrd1)$
        act2 : $obuf2(owr2) := pppop$
        act3 : $obuf1(owr1) := obuf2(ord2)$
        act4 : $vbuf1(vwr1) := vbuf2(vrd2)$
            ... //Update buffer indices
        act14 : $Valid2 := FALSE$
    end
```

The control signal *Valid2* is set to *FALSE*, invalidating the value in *vbuf2*, which retains its previous value. The results from the earlier instruction are handled in the normal way and *Valid1* remains *TRUE*.

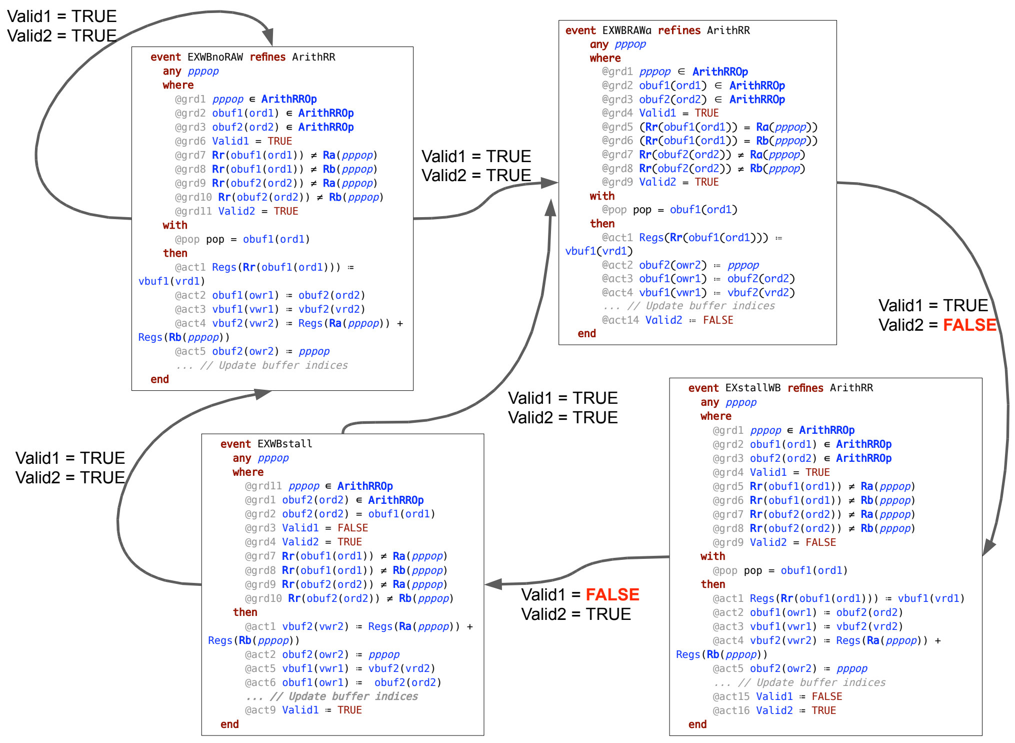
The event *EXstallWB* is now enabled, which writes the new result to the buffer *vbuf2* and sets *Valid2* to *TRUE*, transfers the invalid result from *vbuf2* to *vbuf1* and sets *Valid1* to *FALSE*, and writes the value in *vbuf1* back to the Register File.

```
Event  $EXstallWB \widehat{=}$
refines  $ArithRR$
    any
        $pppop$
    where
        grd1 : $pppop \in ArithRROp$
        grd2 : $obuf1(ord1) \in ArithRROp$
        grd3 : $obuf2(ord2) \in ArithRROp$
        grd4 : $Valid1 = TRUE$
        grd5 : $Rr(obuf1(ord1)) \neq Ra(pppop)$
        grd6 : $Rr(obuf1(ord1)) \neq Rb(pppop)$
        grd7 : $Rr(obuf2(ord2)) \neq Ra(pppop)$
        grd8 : $Rr(obuf2(ord2)) \neq Rb(pppop)$
        grd9 : $Valid2 = FALSE$
    with
        pop : pop = obuf1(ord1)
    then
        act1 : $Regs(Rr(obuf1(ord1))) := vbuf1(vrd1)$
        act2 : $obuf1(owr1) := obuf2(ord2)$
        act3 : $vbuf1(vwr1) := vbuf2(vrd2)$
        act4 : $vbuf2(vwr2) := Regs(Ra(pppop)) + Regs(Rb(pppop))$
        act5 : $obuf2(owr2) := pppop$
            ... //Update buffer indices
        act14 : $Valid1 := FALSE$
        act15 : $Valid2 := TRUE$
    end
```

Finally, the event *WXWBStall* which refines *skip* is enabled which, because *Valid1* is *FALSE*, does not perform the Register File write back The *bubble* of invalid data has now been flushed from the pipeline, *Valid1* is set to *TRUE* and normal pipeline operation resumes.

```
Event  $EXWBstall \widehat{=}$
    any
        $pppop$
    where
        grd1 : $pppop \in ArithRROp$
        grd2 : $obuf2(ord2) \in ArithRROp$
        grd3 : $obuf2(ord2) = obuf1(ord1)$
        grd4 : $Valid1 = FALSE$
        grd5 : $Valid2 = TRUE$
        grd6 : $Rr(obuf1(ord1)) \neq Ra(pppop)$
        grd7 : $Rr(obuf1(ord1)) \neq Rb(pppop)$
        grd8 : $Rr(obuf2(ord2)) \neq Ra(pppop)$
        grd9 : $Rr(obuf2(ord2)) \neq Rb(pppop)$
    then
        act1 : $vbuf2(vwr2) := Regs(Ra(pppop)) + Regs(Rb(pppop))$
        act2 : $obuf2(owr2) := pppop$
        act5 : $vbuf1(vwr1) := vbuf2(vrd2)$
        act6 : $obuf1(owr1) := obuf2(ord2)$
        act9 : $Valid1 := TRUE$
            ... //Update buffer indices
    end
```

The stalling mechanism is shown in Figure 7.9. Concrete events represent the transitions of the model's state. The numbered arcs are annotated with the values of *Valid1* and *Valid2* and point to the event or events that are enabled in the next pipeline state.



```
event EXWBnoRAW refines ArithRR
  any pppop
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 obuf1(ord1) ∈ ArithRROp
    @grd3 obuf2(ord2) ∈ ArithRROp
    @grd6 Valid1 = TRUE
    @grd7 Rr(obuf1(ord1)) ≠ Ra(pppop)
    @grd8 Rr(obuf1(ord1)) ≠ Rb(pppop)
    @grd9 Rr(obuf2(ord2)) ≠ Ra(pppop)
    @grd10 Rr(obuf2(ord2)) ≠ Rb(pppop)
    @grd11 Valid2 = TRUE
  with
    @pop pop = obuf1(ord1)
  then
    @act1 Regs(Rr(obuf1(ord1))) ≔
vbuf1(vrd1)
    @act2 obuf1(owr1) ≔ obuf2(ord2)
    @act3 vbuf1(vwr1) ≔ vbuf2(vrd2)
    @act4 vbuf2(vwr2) ≔ Regs(Ra(pppop)) +
Regs(Rb(pppop))
    @act5 obuf2(owr2) ≔ pppop
    ... // Update buffer indices
end
```

```
event EXWBRAWa refines ArithRR
    any pppop
    where
      @grd1 pppop ∈ ArithRROp
      @grd2 obuf1(ord1) ∈ ArithRROp
      @grd3 obuf2(ord2) ∈ ArithRROp
      @grd4 Valid1 = TRUE
      @grd5 (Rr(obuf1(ord1)) = Ra(pppop))
      @grd6 (Rr(obuf1(ord1)) = Rb(pppop))
      @grd7 Rr(obuf2(ord2)) ≠ Ra(pppop)
      @grd8 Rr(obuf2(ord2)) ≠ Rb(pppop)
      @grd9 Valid2 = TRUE
    with
      @pop pop = obuf1(ord1)
    then
      @act1 Regs(Rr(obuf1(ord1))) ≔
vbuf1(vrd1)
      @act2 obuf2(owr2) ≔ pppop
      @act3 obuf1(owr1) ≔ obuf2(ord2)
      @act4 vbuf1(vwr1) ≔ vbuf2(vrd2)
      ... // Update buffer indices
      @act14 Valid2 ≔ FALSE
  end
```

```
event EXstallWB refines ArithRR
  any pppop
  where
    @grd1 pppop ∈ ArithRROp
    @grd2 obuf1(ord1) ∈ ArithRROp
    @grd3 obuf2(ord2) ∈ ArithRROp
    @grd4 Valid1 = TRUE
    @grd5 Rr(obuf1(ord1)) ≠ Ra(pppop)
    @grd6 Rr(obuf1(ord1)) ≠ Rb(pppop)
    @grd7 Rr(obuf2(ord2)) ≠ Ra(pppop)
    @grd8 Rr(obuf2(ord2)) ≠ Rb(pppop)
    @grd9 Valid2 = FALSE
  with
    @pop pop = obuf1(ord1)
  then
    @act1 Regs(Rr(obuf1(ord1))) ≔ vbuf1(vrd1)
    @act2 obuf1(owr1) ≔ obuf2(ord2)
    @act3 vbuf1(vwr1) ≔ vbuf2(vrd2)
    @act4 vbuf2(vwr2) ≔ Regs(Ra(pppop)) +
Regs(Rb(pppop))
    @act5 obuf2(owr2) ≔ pppop
    ... // Update buffer indices
    @act15 Valid1 ≔ FALSE
    @act16 Valid2 ≔ TRUE
end
```

```
event EXWBstall
  any pppop
  where
    @grd11 pppop ∈ ArithRROp
    @grd1 obuf2(ord2) ∈ ArithRROp
    @grd2 obuf2(ord2) = obuf1(ord1)
    @grd3 Valid1 = FALSE
    @grd4 Valid2 = TRUE
    @grd7 Rr(obuf1(ord1)) ≠ Ra(pppop)
    @grd8 Rr(obuf1(ord1)) ≠ Rb(pppop)
    @grd9 Rr(obuf2(ord2)) ≠ Ra(pppop)
    @grd10 Rr(obuf2(ord2)) ≠ Rb(pppop)
  then
    @act1 vbuf2(vwr2) ≔ Regs(Ra(pppop)) +
Regs(Rb(pppop))
    @act2 obuf2(owr2) ≔ pppop
    @act5 vbuf1(vwr1) ≔ vbuf2(vrd2)
    @act6 obuf1(owr1) ≔  obuf2(ord2)
    ... // Update buffer indices
    @act9 Valid1 ≔ TRUE
end
```

FIGURE 7.9: Synchronous Elastic Buffers: Stalling mechanism

Two gluing invariants are required to prove that this is a correct refinement.

$$\texttt{inv14}: \; Valid1 = TRUE \Rightarrow vbuf1(vrd1) = Regs(Ra(obuf1(ord1))) + Regs(Rb(obuf1(ord1)))$$

$$\texttt{inv15}: \; Valid2 = TRUE \Rightarrow vbuf2(vrd2) = Regs(Ra(obuf2(ord2))) + Regs(Rb(obuf2(ord2)))$$

From Figure 7.9 it can be seen that all possible combinations of *Valid1* and *Valid2* are feasible except one: $Valid1 = FALSE$ and $Valid2 = FALSE$.

The invariant

$$\texttt{inv20}: \; \neg(Valid1 = FALSE \wedge Valid2 = FALSE)$$

is therefore introduced and proved, showing that this case cannot occur and that pipeline cannot deadlock.

In this case 195 proof obligations are discharged automatically.

7.6.4 Shared Event Pipeline Decomposition

When pipeline communication is via shared registers, a two-way, *shared event* decomposition is used to decompose the combined events into two processes. Where synchronous elastic buffers are introduced to de-couple the pipeline stages, a three-way, *shared event* decomposition [Butler, 2009] as described in Section 2.9 on Page 18 is used. The microarchitecture of the partitioning is shown in Figure 7.10.

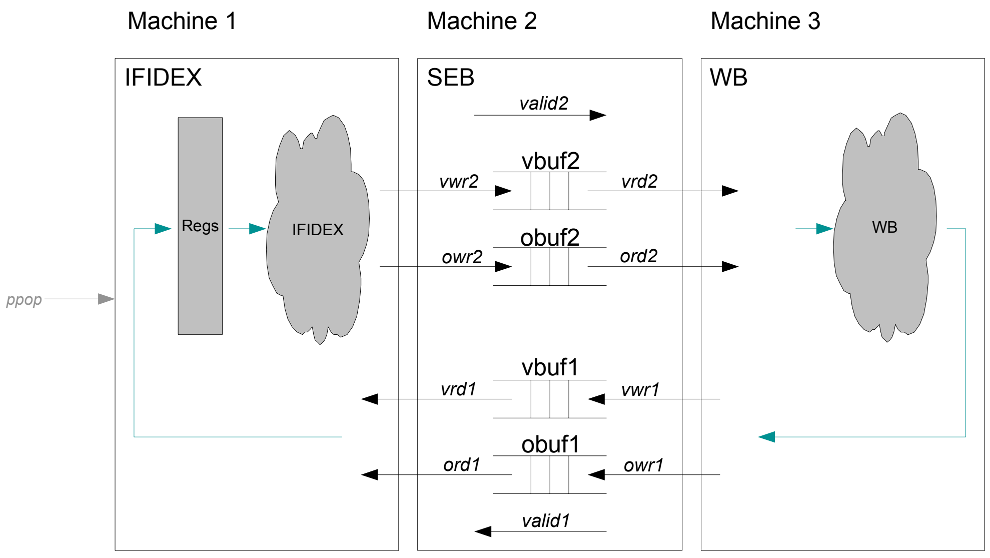


FIGURE 7.10: Synchronous elastic buffer decomposition: Microarchitecture

The *Execute* stage of *Machine 1* and *WriteBack* stage of *Machine 3* do not communicate directly with each other; all communication is via a shared event, *SEBtransfer*, in a separate Event-B machine (*Machine 2*) which represents the synchronous elastic buffers and which increments the buffer indices.

```
Event  SEBtransfer ≙
    then
        act1 : ord1 := owr1
        act2 : vrd1 := vwr1
        act3 : vwr1 := vwr1 + 1
        act4 : owr1 := owr1 + 1
        act5 : ord2 := owr2
        act6 : vrd2 := vwr2
        act7 : vwr2 := vwr2 + 1
        act8 : owr2 := owr2 + 1
    end
```

7.6.5 A Review of the use of Synchronous Elastic Design with Event-B

Despite the increase in modelling complexity introduced by the incorporation of latency-insensitive buffers into the pipeline, increasing the number of proof obligations from 33 to 195 when compared with the original shared register model, these proof obligations are still discharged automatically. Stalling is inherently a more complex solution to the problem of RAW hazards than forwarding, but on the other hand the distributed stalling mechanism that latency-insensitive design enables is inherently less complex than the centralised mechanism using to implement the branch instruction.

It is a great benefit of our method that the gluing invariants discovered in the shared variable implementation can be re-used with minor and well-defined modifications which would be amenable to automatic tool support. In addition, the introduction of the extra actions needed to update the buffer indices could also be automated.

The microarchitectural designer has the choice as to whether to use forwarding or stalling techniques and shared register or buffered communication. The trade-offs between verification complexity, power consumption and performance can be taken into account at this early stage. Forwarding, though easier to implement, may not meet the performance requirements.

Components that become too large or complex can be partitioned to form sub-systems within which communication between components is both latency-insensitive and efficient. Track length constraints preclude the design of long pipelines with only shared register inter-stage communication. Introducing latency-insensitve buffers breaks up the pipeline into more manageable sections without affecting its functionality. The exploration is done at the specification level with the support of Event-B refinement and automatic proof.

The designer can also use these latency-insensitive design techniques to model instructions which in some SoC microprocessor architectures can have variable latency, such as memory accesses and arithmetic instructions.

Chapter 8

Conclusions

The complexity of modern System-on-Chip (SoC) hardware stretches the existing design and verification flows, languages and tools to the limit of their capabilities [Asanovic et al., 2006], [Sylvester and Keutzer, 2001]. Verification takes a larger and larger proportion of the overall effort and it is often very late in the design process that timing issues, resulting from the very small feature sizes of modern silicon processes, are encountered and can only be corrected by substantial re-design [Carloni and Sangiovanni-Vincentelli, 2002].

Although semi-conductor companies express the desire to raise the design process to the Electronic System Level (ESL) [Asanovic et al., 2006] in order to improve the predictability of the verification closure process, the approach that the EDA industry has taken to address this need has been fragmentary, no clear standards have emerged, and the tried and proven RTL design methodology still forms the significant bedrock of any modern SoC design flow. There is a clear need to enhance existing design flows to be better able to manage this increased complexity, without losing the well-established benefits that have driven successful synchronous design.

In a current RTL development flow, the implementation will be handed to a verification engineer who must verify the design against a test plan. The difficulty comes in developing a credible test plan that reflects the complexity of the design. An ISA specification, for instance, on its own cannot be used as the basis for the test plan; the test engineer needs to understand the detailed pipeline implementation. In practice, the test engineer must derive the behaviour of the *combined state machine* from the individual state machines that represent each pipeline stage. Just as ensuring full code coverage of each individual process is insufficient, ensuring full arc coverage of each of the interacting state machines is also insufficient. In general, generating a combined state machine is impractical as it is extremely difficult to decide which of all the possible combined transitions are actually valid.

Property checking, rather than using tests, takes a set of temporal properties [Cohen et al., 2004], [Sutherland et al., 2004], derived from the specification and checks these properties against the synthesised gate-level description. Property checking depends on having a comprehensive set of properties to represent the desired behaviour. It is very difficult to establish whether sufficient properties have been written or not, and *property coverage* [Chockler et al., 2001], [Hoskote et al., 1999] is a topic of ongoing research. Without a measurable outcome, property checking will not become an indispensable component in the SoC flow.

In our Event-B, proof-based method, design complexity is exposed explicitly in the design process, the combined state machine is visible from an early stage of the design, the effect on complexity of design decisions can be seen immediately and design alternatives can be explored and measured. Proof-based refinement, with invariant preservation and convergence, can be seen to obviate the need for unit tests.

Although combined state machine Arc Coverage is a desirable, but often unattainable, goal in design verification, it is in itself not sufficient to ensure full functional coverage. All valid *paths* through the combined state machine must also be covered and identifying these paths is extremely difficult. It is a major benefit of our proof-based method that there is no need to be concerned about the paths of the combined state machine; invariant preservation and convergence are proved for all possible interleavings of the composed events.

In our approach, an enhancement of the System-on-Chip hardware design and verification flow, incorporating synthesis, has been explored which exploits the synergy between the Event-B method and the rule-driven approach of guarded atomic action, high-level synthesis. Since both these methods are based fundamentally on guarded atomic actions, it has been possible to define a augmented flow whereby a formal and systematic approach to high-level specification with refinement and supporting architectural exploration can be provided by Event-B, resulting in a concrete specification that can be mapped directly to a TRS description for high-level synthesis to RTL and therefore close the gap between specification and implementation. It has also been shown that, in the case of microprocessor pipelines, it is possible to derive a concrete specification that can be mapped directly to RTL.

A general approach to specifying and refining SoC components, in particular those with pipelined architectures, has been presented which focuses on the issues that arise when feedback is present in the pipeline. The method begins with an abstract Event-B model representing the specification of the required behaviour. The abstract specification represents a high-level view of the hardware which executes in a single cycle. A refinement of the abstract model is then introduced which represents the behaviour as a two-stage pipeline. The two stages communicate via shared registers and gluing invariants are introduced to prove that the two-stage pipeline implements the abstract specification.

This model executes in two cycles, with overlapping execution of the pipeline stages. This two-stage model is then further refined until all pipeline stages within a pipeline feedback loop have a concrete implementation. Once the invariants have been proved, which shows that the overlapping execution within the feedback loop has been implemented correctly, shared event decomposition is used to decompose the pipeline model into a set of models, one for every stage of the pipeline. Models representing the stages outside the feedback loop can then be introduced. Each pipeline stage model represents a hardware process of the final pipeline implementation, and the invariants represent the properties of that pipeline.

This general approach is then applied, using the *register/register arithmetic* and *branch* instructions as examples, to show that the ISA specification of the instruction can be refined systematically to a pipelined model that can be proved to implement its ISA specification. The final, concrete models of each instruction comprise a set of Event-B machines, one for each pipeline stage. The machines representing each particular stage are then composed formally to provide a complete, concrete model of the pipeline. The method ensures, through the introduction of gluing invariants at each stage, that microarchitectural considerations are addressed early in the design flow. Different microarchitectures may be explored and verified at the specification level. Stepwise refinement allows the developer to manage the multiplicity of cases caused by pipeline data hazards.

A method for incorporating a latency-insensitive design protocol for inter-component communication has also been developed, which retains the benefits of asynchronous communication but within a synchronous tool flow. This does, however, require that components are stallable and do not cause sub-system deadlock. The DLX register/register arithmetic instruction has been used to show that synchronous elastic buffers can be used to implement a distributed pipeline stalling mechanism, as an efficient alternative to forwarding, to manage data hazards which has been proved to be free from deadlock.

Three goals were set for our work which, if achieved, would represent significant, original contributions to the area of microelectronic design.

First, that alternative pipeline architectures within components could be developed from a high-level specification, compared and verified systematically.

Second, that a high-level model for latency-insensitive communication could be developed that could be used at the specification level to verify formally that communicating components obey the latency-insensitive protocol and will not cause deadlock.

Third, that the specification could be refined to a level of abstraction that matches that required for input to high-level or RTL synthesis.

All three goals have been achieved. In our *correct-by-construction* approach, Event-B is used to represent the abstract hardware specification with the TRS-style descriptions familiar to hardware designers who use Bluespec or CAL. The abstract specification is then refined systematically to reflect the architectural decisions of the designer. It has been shown how the designer is able to deal with one architectural consideration at a time, refining a particular aspect of the design to a concrete representation while leaving the rest of the representation abstract. At each refinement step the Rodin tool helps the designer to discover the gluing invariants that must be proved to demonstrate that the concrete representation is a correct refinement of the abstract. These invariants are fundamental properties of the design that can be translated directly into PSL or SVA descriptions and used downstream in the flow for RTL formal and simulation-based verification. Where performance or complexity considerations require that a component be split into a sub-system of communicating components, it has been shown how an alternative refinement, incorporating synchronous elastic buffers, can be derived that also meets the abstract specification. The refinement process continues until a concrete representation of the specification has been derived that is suitable for either TRS or RTL synthesis.

The verification effort has been raised to the specification level because the concrete representation has been proved formally to implement the abstract specification. Verification is made manageable because it is performed incrementally within the design flow. The hardware and its associated properties are described in an event-based language which is a natural vehicle for synchronous hardware description and all the associated proof obligations are generated and proved by the Rodin tool environment.

The fourth, and overriding goal, was that the method developed could fit seamlessly within a modern SoC verification flow. It has been a drawback of many formal hardware verification developments that they sit as an adjunct to the main flow, can be useful in finding bugs, but require the design to be translated into specialist representations for analysis by specialist scientists. A modern verification flow requires a measurable outcome that provides the confidence required for the design to be signed off. A mechanism that may or may not find additional bugs does not necessarily increase that confidence. Code and functional coverage measurements provide the confidence required for sign-off in current, simulation-based flows. We have developed, systematically, models that have been proved to be a correct implementation of the abstract specification for all possible paths through the combined state machine that represents the concurrent behaviour of a component. The component can therefore to be said to have achieved full path coverage and this measurement can be correlated directly with path coverage in a simulation-based flow. It is therefore possible with our method to combine the coverage results from formal and simulation-based verification into a single metric for design sign-off.

The work presented has focused on microprocessor pipelines, but these are specialisations of what constitutes the general mechanism for representing complex hardware designs, *communicating state machines*, which communicate via shared registers or by message-passing with buffers. A pipeline stage is a state machine that communicates with neighbouring stages using the pipeline stage registers. Our method can therefore be used to develop implementations of the components that form the building blocks of a modern hardware design. For each type of component, the challenge will to identify the abstract specification that best embodies the essential characteristics of that component and to develop the refinement strategy that allows a concrete representation to be derived formally with automatic proof. A library of parameterised templates for each component type can then be assembled for subsequent use in the mainstream design flow. Such templates would best be developed in a small, leading-edge group with highly-experienced designers or in the central research and development department of a larger semiconductor company, using existing designs. Having incurred the initial cost of developing the templates, the return on investment would be realised in the mainstream flow with greatly improved efficiency in reaching coverage targets when compared to traditional, simulation-based methods and the increased confidence that formally-based coverage results would deliver.

As far as SoC processor pipelines are concerned, these will inevitably become more complex while attempting to retain the low power consumption benefits of the simple, five-stage pipelines that dominate current SoC designs. Super-scalar implementations with multi-issue instruction capabilities will require the multiple memory access and out-of-order completion capabilities demonstrated in our IP lookup circular buffer development. Instructions with variable latency can be managed elegantly using synchronous, latency-insensitive design.

The long-term benefit of incorporating our method into the verification flow will be to bridge the specification gap that currently exists, to enable the verification process to begin at the specification level and to allow both functional and performance considerations to be addressed through architectural exploration at a much earlier stage in the flow than is currently possible.

References

J.R. Abrial. The Event-B Modelling Notation. 2007. URL http://deploy-eprints.ecs.soton.ac.uk/11/3/notation-1.5.pdf.

JR Abrial. Rigorous Open Development Environment for Complex Systems: event B language. 2005.

J.R. Abrial. Event model decomposition. 2009.

J.R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae, XXI*, 2006.

J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. *B*, 98:83–128, 1998.

J.R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *International Conference on Formal Engineering Methods (ICFEM)*, 2006.

P. Alexander and P. Baraona. Formal methods at the systems level. In *Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on*, 1997. URL http://ieeexplore.ieee.org/iel3/4942/13802/00638303.pdf?arnumber=638303.

ARM. Specification Rev 2.0. *ARM Limited*, 1999.

ARM. AMBA AXI Protocol Specification. 2003.

Arvind and J.C. Hoe. Micro-architecture Exploration and Synthesis via TRS's. Technical report, MIT, April 1999.

Arvind and X. Shen. Using Term Rewriting Systems to Design and Verify Processors. *IEEE Micro*, 19(3):36–46, 1999.

Arvind, R. Nikhil, D. Rosenband, and N. Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. *CSAIL, April*, 2004.

K. Asanovic. Transactors for Parallel Hardware and Software Co-Design. *International High Level Design Validation and Test Workshop, IEEE, November*, 2007.

K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, 18(2006-183):19, 2006.

F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

P. Baraona and P. Alexander. VSPEC: A Language for Digital System Specification. *Proceedings of the Al and Systems Engineering Workshop*, pages 19–27, 1994.

Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl von Platen, Marco Mattavelli, and Mickaël Raulet. Opendf: a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*, 36(5): 29–35, 2008. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1556444.1556449.

N. Bombieri, F. Fummi, and G. Pravadelli. On the evaluation of transactor-based verification for reusing TLM assertions and testbenches at RTL. *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 1007–1012, 2006a.

N. Bombieri, F. Fummi, and G. Pravadelli. A Methodology for Abstracting RTL Designs into TL Descriptions. *Proc. of ACM/IEEE MEMOCODE*, pages 145–152, 2006b.

E. Borger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. *ZUM'97, the Z Formal Specification Notation: 10th International Conference of Z Users, Reading, UK, April 3-4, 1997: Proceedings*, 1997.

R. Brown. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

J.R. Burch and D.L. Dill. Automatic verification of Pipelined Microprocessor Control. *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 68–80, 1994.

M. Butler. Decomposition structures for Event-B. *Integrated Formal Methods iFM2009, Springer, LNCS*, 5423, 2009.

L.P. Carloni and A.L. Sangiovanni-Vincentelli. Coping with latency in SOC design. *Micro, IEEE*, 22(5):24–35, 2002.

L.P. Carloni, K.L. McMillan, A. Saldanha, and A.L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency insensitive design. *Proc. Intl. Conf. on Computer-Aided Design*, 1999.

L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(9):1059–1076, 2001.

H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. *Lecture Notes in Computer Science*, 2031:528+, 2001. URL citeseer.ist.psu.edu/chockler02coverage.html.

E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Lecture Notes In Computer Science*, pages 52–71, 1981.

E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001. URL citeseer.ist.psu.edu/clarke01bounded.html.

B. Cline. Forte announces cynthesizer 3.3. Website, May 2007a. URL http://www.forteds.com/news/pr052507.pdf.

B. Cline. Transaction-level modeling gains further momentum, 2007b. URL http://www.chipdesignmag.com/print.php?articleId=813?issueId=20.

B. Cohen, S. Venkataramanan, and A. Kumari. *Using PSL/Sugar for formal and dynamic verification: Guide to Property Specification Language for Assertion-based Verification*. VhdlCohen Publ., 2004.

J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 657–662, New York, NY, USA, 2006a. ACM. ISBN 1-59593-381-6. doi: http://doi.acm.org/10.1145/1146909.1147077.

J. Cortadella, M. Kishinevsky, and B. Grundmann. Specification and design of synchronous elastic circuits. In *Proc. International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, pages 16–21, February 2006b.

J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and design of synchronous elastic circuits. In *TAU'06: Proceedings of the ACM/IEEE International Workshop on Timing Issues 2006*. Citeseer, 2006c.

N.H. Dave. *Designing a processor in Bluespec*. Masters thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2005.

S. Devadas, A. Ghosh, and K. Keutzer. *Logic synthesis*. McGraw-Hill, Inc. New York, NY, USA, 1994.

A. Donlin. Transaction Level Modeling: Flows and Use Models. *CODES+ ISSS*, 4:75–80, 2004.

R. Drechsler and S. Horeth. Gatecomp: Equivalence checking of digital circuits in an industrial environment. *Int'l Workshop on Boolean Problems*, pages 195–200, 2002.

S.A. Edwards. The Challenges of Hardware Synthesis from C-Like Languages. *Design, Automation, and Test in Europe: Proceedings of the conference on Design, Automation and Test in Europe-*, 1:66–67, 2005.

N. Evans and M. Butler. A Proposal for Records in Event-B. *FM*, pages 21–27, 2006.

H. Foster, E. Marschner, and Y. Wolfsthal. IEEE 1850 PSL: The next generation, 2005.

D. Gajski, J. Zhu, and R. Domer. Essential Issues in Codesign. Technical report, Technical report ICS-97-26, University of California, Irvine, 1997, 1997.

D. Gebhardt and K.S. Stevens. Elastic Flow in an Application Specific Network-on-Chip. *Electronic Notes in Theoretical Computer Science*, 200(1):3–15, 2008.

D. Geer. Chip makers turn to multicore processors. *IEEE Computer*, 38(5):11–13, 2005.

F. Ghenassia. *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2006.

M. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press New York, NY, USA, 1993.

M. Gordon, J. Hurd, and K. Slind. Executing the formal semantics of the accellera property specification language by mechanised theorem proving, 2003. URL citeseer.ist.psu.edu/gordon03executing.html.

M.G. Hadjinicolaou, G. Musgrave, and R.B. Hughes. Graphical specification of digital systems using interval temporal logic. In *ASIC Conference and Exhibit, 1994. Proceedings., Seventh Annual IEEE International*, 1994. URL http://ieeexplore.ieee.org/iel2/3197/9098/00404589.pdf?tp=&isnumber=&arnumber=404589.

S. Hallerstede. Justifications for the Event-B Modelling Notation. In *B 2007: Formal Specification and Development in B*, 2007.

F. Haque and J. Michelson. *Art of Verification with VERA*. Verification Central, 2001.

A. Hartstein and T. Puzak. The optimum pipeline depth considering both power and performance. *ACM Trans. Archit. Code Optim.*, 1(4):369–388, 2004. ISSN 1544-3566. doi: http://doi.acm.org/10.1145/1044823.1044824.

J. Henkel. Closing the SoC design gap. *Computer*, 36(9):119–121, 2003.

J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

J.C. Hoe. Synthesis of Operation-Centric Hardware Descriptions. *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 511–519, 2000.

J.C. Hoe. Operation-centric hardware description and synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(9):1277–1288, 2004.

J.C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. *Proceedings of the IFIP TC10/WG10. 5 Tenth International Conference on Very Large Scale Integration: Systems on a Chip*, pages 595–619, 1999.

Y. Hollander, M. Morley, and A. Noy. The e language: A fresh separation of concerns. *Proceedings of TOOLS*, 38, 2001.

P. Horowitz and W. Hill. *The Art of Electronics.* Cambridge University Press, 1989.

Y. Hoskote, T. Kam, P.H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. *Proc. 36th Design automation conference*, pages 300–305, 1999.

D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004.

Y. Huygen. Eda consortium report. Web, April 2007. URL http://www.edac.org/downloads/pressreleases/07-04-9_MSS_Q4_2007_ReleaseFINAL.pdf.

R.B. Jones, D.L. Dill, and J.R. Burch. Efficient validity checking for processor verification. In *IEEE International Conference on Computer-Aided Design*, San Jose, California, USA, 1995. URL citeseer.ist.psu.edu/jones95efficient.html.

J.J. Joyce, G. Birtwistle, and M. Gordon. Proving a Computer Correct in Higher Order Logic, Report No. 100. *Computer Laboratory, Cambridge University*, 1986.

J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press. URL citeseer.ist.psu.edu/burch90symbolic.html.

M. Kaufmann and J. Moore. Industrial proofs with acl2. Technical report, University of Texas, 2004.

K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

D. Kroening and W.J. Paul. Automated pipeline design. In *Proceedings of the 38th conference on Design automation*, pages 810–815. ACM New York, NY, USA, 2001.

S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous Elastic Networks. *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, pages 19–30, 2006.

T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai. A framework for object oriented hardware specification, verification, and synthesis. *Proceedings of the Design Automation Conference (DAC'2001)*, pages 413–418, 2001.

R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and DM Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92, 2003.

P. Lieverse, P. van der Wolf, and E. Deprettere. A trace transformation technique for communication refinement. In *Proceedings 9th International Symposium on Hardware/Software Codesign (CODES'2001)*, pages 134–139, Copenhagen, Denmark, April 25–27 2001. URL citeseer.ist.psu.edu/lieverse01trace.html.

P. Manolios. Correctness of pipelined machines. *Formal Methods in Computer-Aided Design–FMCAD*, 1954:161–178, 2000.

P. Manolios and S. Srinivasan. A computationally efficient method based on commitment refinement maps for verifying pipelined machines models. *ACM-IEEE International Conference on Formal Methods and Models for Codesign*, pages 189–198, 2005a.

P. Manolios and S.K. Srinivasan. Verification of executable pipelined machines with bit-level interfaces. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, page 862. IEEE Computer Society, 2005b.

B. Meyer, E.T. Hochschule, and S. Zurich. The grand challenge of trusted components. *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 660–667, 2003.

W. Mueller, J. Ruf, D. Hofmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The Simulation Semantics of SystemC. *Proc. of DATE 2001*, 2001.

M.C. Ng, M. Vijayaraghavan, N. Dave, G. Raghavan, and J. Hicks. From WiFi to WiMAX: Techniques for High-Level IP Reuse across Different OFDM Protocols. *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 71–80, 2007.

R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, 2004.

R.S. Nikhil. Composable Guarded Atomic Actions: a Bridging Model for SoC Design. *Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on*, pages 23–28, 2007.

OSCI-TLMSubgroup. Systemc tlm 2 draft. Technical report, 2007. URL `http://www.systemc.org/web/sitedocs/TLM_2_0.html`.

D.L. Perry. *VHDL*. McGraw-Hill New York, 1994.

J. Plosila and K. Sere. Action systems in pipelined processor design. In *Proceedings Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 156–166. Citeseer, 1997.

G. Prabhu. Computer architecture tutorial, 1 2000. URL `http://www.cs.iastate.edu/~prabhu/Tutorial/title.html`.

V. R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986. URL `citeseer.ist.psu.edu/pratt86modelling.html`.

R. Kumar, C. Blumenroehr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design-A classification and survey. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166, pages 294–299, Palo Alto, CA, USA, 1996. Springer Verlag. URL `citeseer.ist.psu.edu/kumar96formal.html`.

D. Ragan, P. Sandborn, and P. Stoaks. A Detailed Cost Model for Concurrent Use With Hardware/Software Co-Design. *Proceedings of the Design Automation Conference*, pages 269–274, 2002.

A. Rose, S. Swan, J. Pierce, and J.M. Fernandez. Transaction Level Modeling in SystemC. *OSCI TLM-WG*, 2005.

D.L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. *Proc. 41st DAC (June 2004)*, 2004.

D.L. Rosenband and Arvind. Hardware synthesis from guarded atomic actions with performance specifications. *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 784–791, 2005.

G. Russell. *CAD for VLSI*. Van Nostrand Reinhold, 1985.

N. Shankar. Lazy compositional verification. *Lecture Notes in Computer Science*, 1536: 541–564, 1998.

R. Silva and M.J. Butler. Supporting reuse mechanisms for developments in event-b: Composition. Technical report, University of Southampton, 2009.

R. Silva, C. Pascal, T.S. Hoang, and M. Butler. Decomposition tool for Event-B. 2010.

P. Stanford and P. Mancuso. *EDIF: Electronic Design Interchange Format: Version 200.* Electronic Industries Association, Engineering Department, 1990.

I.E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

S. Sutherland, S. Davidmann, and P. Flake. *System Verilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Kluwer Academic Pub, 2004.

S. Swan. SystemC transaction level models and RTL verification. *Proceedings of the 43rd annual conference on Design automation*, pages 90–92, 2006.

D. Sylvester and K. Keutzer. Impact of small process geometries on microarchitectures in systems on a chip. *Proceedings of the IEEE*, 89(4):467–489, 2001.

S. Tahar and R. Kumar. Formal Verification of Pipeline Conflicts in RISC Processors. *Proc. European Design Automation Conference (EURO-DAC94), Grenoble, France, September*, pages 285–289, 1994.

M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2): 25–35, 2002. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2002.997877.

D.E. Thomas and P.R. Moorby. *The Verilog (r) Hardware Description Language.* Kluwer Academic Publishers, 2002.

M.Y. Vardi and P. Wolper. Automata theoretic techniques for modal logics of programs: (extended abstract). In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 446–456, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-133-4. doi: http://doi.acm.org/10.1145/800057.808711.

T. W. Williams. *VLSI testing.* Elsevier Science Publishers BV Amsterdam, The Netherlands, The Netherlands, 1986.

Appendix A

Published Papers

On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification

Colley, J and Butler, M

Refinement Based Methods for the Construction of Dependable Systems
Dagstuhl, Germany
September 2009

On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification

John Colley[1], Michael Butler[2]

[1] University of Southampton,
School of Electronics and Computer Science
Southampton, SO17 1BJ, UK
jlc05r@ecs.soton.ac.uk
[2] University of Southampton,
School of Electronics and Computer Science
Southampton, SO17 1BJ, UK
mjb@ecs.soton.ac.uk

Abstract. Microprocessor pipelining is a well-established technique that improves performance and reduces power consumption by overlapping instruction execution. Verifying, however, that an implementation meets this ISA specification is complex and time-consuming. One of the key verification issues that must be addressed is that of overlapping instruction execution. This can introduce hazards where, for instance, a new instruction reads the value from a register which will be written by an earlier instruction that has not yet completed. Using Event-B's support for refinement with automated proof, a method is explored where the abstract machine represents directly an instruction from the ISA that specifies the effect that the instruction has on the microprocessor register file. Refinement is then used systematically to derive a concrete, pipelined execution of that instruction. Microarchitectural considerations are raised to the specification level and design choices can be verified much earlier in the flow. The method proposed therefore has the potential to be integrated into an existing high-level synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

Keywords. Event-B, Pipeline, ISA

1 Introduction

Microprocessor pipelining is a well-established technique that improves performance and reduces power consumption by overlapping instruction execution. Modern System-on-Chip microprocessors used for mobile applications have very stringent power consumption requirements and are typically based on the 5-stage DLX microprocessor [1]. From the Instruction Set Architecture (ISA) specification, a pipelined microarchitecture is developed that implements the specification. Verifying, however, that an implementation meets this ISA specification is complex and time-consuming. Current verification techniques are predominantly test based within a Register Transfer Level (RTL) simulation and synthesis flow.

One of the key verification issues that must be addressed is that of overlapping instruction execution. This can introduce hazards where, for instance, a new instruction reads the value from a register which will be written by an earlier instruction that has not yet completed. These are termed Read-After-Write (RAW) data hazards [1]. The presence of hazards depends on the instruction mix presented to the microprocessor and pseudo-random test generation techniques have been used in an attempt to achieve adequate test coverage of instruction combinations [2], [3] .

Formal techniques, using both model checking and theorem proving, have been used in microprocessor verification, but as an adjunct to the simulation-based flow. These techniques are applied after the design is completed in the hope of detecting errors not discovered by testing. Higher-level hardware description languages such as Bluespec [4] and CAL [5], which provide an automatic synthesis route to RTL, can speed up the design process, but it is the verification costs that dominate in the overall flow and the bulk of the verification must still be done at the Register Transfer Level.

Event-B [6], [7] is a proof-based modelling language and method that enables the development of specifications using refinement. The Rodin platform [8] is the Eclipse-based IDE that provides support for Event-B refinement and mathematical proof. Using Event-B's support for refinement with automated proof, a method is explored where the abstract machine represents directly an instruction from the ISA that specifies the effect that the instruction has on the microprocessor register file. Refinement is then used systematically to derive a concrete, pipelined execution of that instruction. At each refinement step the importance is shown of addressing the inherent simultaneity that characterises the pipelined behaviour and, in particular, the effects that feedback has in pipeline construction.

To illustrate the method, the *register/register arithmetic* instruction of a typical System-on-Chip (SoC) microprocessor is chosen that can exhibit RAW data hazards with overlapping execution. The technique, termed *forwarding*, where intermediate values are fed back to a stage that needs them, is employed in modern microprocessors to provide a very efficient means of managing RAW hazards [1]. Debugging the forwarding logic has, however, been found to be difficult and expensive [9] . With the introduction of appropriate invariants in our approach, it is shown that the concrete, pipelined refinement will not preserve these invariants unless the RAW hazards are detected and managed appropriately.

The concrete Event-B model implements forwarding in a way that corresponds directly to the techniques used in microprocessor design and is proved, automatically, in the Rodin environment to be a correct refinement of the abstract ISA specification. Thus, microarchitectural considerations are raised to the specification level and design choices can be verified much earlier in the flow. The concrete model also has a direct correspondence to an equivalent hardware description in the high-level languages Bluespec and CAL, which like Event-B are based on guarded atomic actions. The method proposed therefore has the potential to be integrated into an existing high-level synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

2 An Overview of Event-B

In Event-B, an abstract model comprises a *machine* that specifies the high-level behaviour and a *context*, made up of sets, constants and their properties, that represents the type environment for the high-level machine. The machine is represented as a set of *state variables*, v and a set of events, *guarded atomic actions*, which modify the state. If more than one action is enabled, then one is chosen non-deterministically for *execution*, an observable transition on the state variables which must preserve an *invariant* on the variables, *I(v)*. A more concrete representation of the machine may then be created which refines the abstract machine, and the abstract context may be extended to support the types required by the refinement. *Gluing invariants* are used to verify that the concrete machine is a correct refinement of the abstract. Gluing invariants give rise to proof obligations for pairs of abstract and corresponding concrete events. Events may also have parameters which take, non-deterministically, the values that will make the guards in which they are referenced true.

An event can be represented by the *generalized substitution*,

$$\boxed{\textbf{any } x \textbf{ where } P(x,v) \textbf{ then } v := F(x,v) \textbf{ end}}$$

where x represents the event parameters and v represents the value of the machine state variables. Informally, this event can be fired provided that the guard *P(x, v)* can be satisfied for some value x. The details are explained in [10] .

3 Modelling the Arithmetic Instruction

3.1 The Abstract ISA Model

The structure of a *register/register arithmetic* instruction associates the opcode with a destination register *Rr* and two source registers *Ra* and *Rb*. The Event-B *context*, *PIPEC*, for the arithmetic instruction therefore defines a set of operations *Op*, the type *Register*, the subset of operations that are of type *register/register arithmetic*, *ArithRRop*, and the relationship between the fields of the arithmetic instruction and their associated registers. The conventions of [11] are followed to model operation fields. The context also defines *No Operation*, *NOP*.

```
CONTEXT PIPEC
SETS
    Op
CONSTANTS
    Register
    Rr
    Ra
    Rb
    NOP
    ArithRROp
AXIOMS
    axm1 : $Register \subseteq \mathbb{N}$
    axm2 : $Rr \in Op \rightarrow Register$
    axm3 : $Ra \in Op \rightarrow Register$
    axm4 : $Rb \in Op \rightarrow Register$
    axm5 : $ArithRROp \subseteq Op$
    axm6 : $NOP \in Op$
    axm7 : $NOP \notin ArithRROp$
END
```

The abstract machine, *PIPEM*, defines the register file *Regs* and a single event *ArithRR* that specifies the effect that execution of the instruction has on the register file. For simplicity, the addition operation is shown, but this can more generally be represented by an uninterpreted function [12] without affecting the proof approach used. The parameter *pop* specifies the environment for the event; given an instruction of type *ArithRROp*, the state of the register file will be updated according to that instruction.

```
MACHINE PIPEM
SEES PIPEC
VARIABLES
    Regs
INVARIANTS
    inv1 : $Regs \in Register \rightarrow \mathbb{Z}$
EVENTS
Initialisation
    begin
        act1 : $Regs := Register \times \{0\}$
    end
Event ArithRR $\widehat{=}$
    any
        $pop$
    where
        grd1 : $pop \in ArithRROp$
    then
        act1 : $Regs(Rr(pop)) := Regs(Ra(pop)) + Regs(Rb(pop))$
    end
END
```

The microarchitecture of the abstract machine is shown in Figure 1.

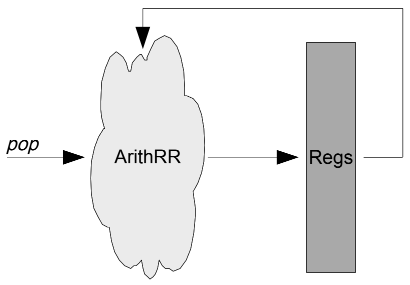


Fig. 1. Abstract Machine: Microarchitecture

3.2 The First Refinement: a 2-stage pipeline

A 2-stage pipeline is now introduced which refines the abstract machine. The second pipeline stage is a concrete representation of the *Write Back (WB)* stage while the first stage is still abstract, representing the *Fetch/Decode/Execute* operations of the pipeline.

MACHINE PIPER1
REFINES PIPEM
SEES PIPEC
VARIABLES
    `Regs`
    `EXop`
    `ALUout`
INVARIANTS
    `inv1 :` $EXop \in Op$
    `inv2 :` $ALUout \in \mathbb{Z}$
    `inv3 :` $ALUout = Regs(Ra(EXop)) + Regs(Rb(EXop))$
EVENTS
Event $FDEXWB \mathrel{\widehat{=}}$
refines *ArithRR*
    any
        ppop
    where
        `grd1 :` $EXop \in ArithRROp$
        `grd2 :` $ppop \in ArithRROp$
        `grd3 :` $Rr(EXop) \neq Ra(ppop)$

```
        grd4 : $Rr(EXop) \neq Rb(ppop)$
    with
        pop : pop = EXop
    then
        act1 : $Regs(Rr(EXop)) := ALUout$
        act2 : $ALUout := Regs(Ra(ppop)) + Regs(Rb(ppop))$
        act3 : $EXop := ppop$
    end
END
```

Two new variables, $ALUout$ and $EXop$ are introduced to represent the $EXWB$ pipeline registers. The parameter *pop* of the abstract *ArithRR* event is bound to the concrete register $EXop$ using an Event-B *witness* and a new parameter *ppop* represents the environment of the refined event, *FDEXWB*. The *FDEXWB* event models the simultaneous execution of both pipeline stages. The microarchitecture of the refined machine is shown in Figure 2.

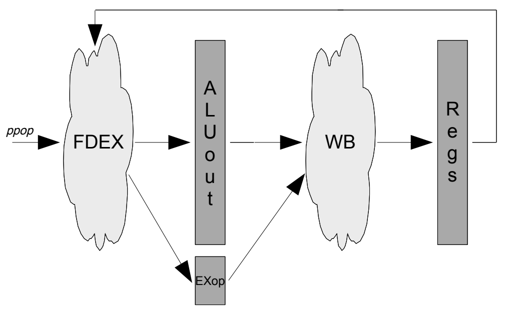


Fig. 2. Refined Machine: Microarchitecture

It is now necessary to introduce the *gluing invariant* to establish that this is a correct refinement of the abstract machine. To preserve the meaning of the abstract specification, the new variable $ALUout$ must always have the value *Regs(Ra(EXop)) + Regs(Rb(EXop))*, as represented by the invariant *inv3*. The Rodin prover, however, shows that this invariant is not preserved by the refined machine. The abstract *FDEX* pipeline stage simultaneously reads the register file while the *WB* stage is writing to it. If the location being read is the same as that being written, a *Read After Write (RAW)* data hazard will be encountered and the wrong value will read by the first pipeline stage. This inherent feedback in the pipelined implementation must be addressed explicitly if it is to meet its specification.

3.3 Detecting the RAW Hazard

The abstract *FDEX* pipeline stage may only read from the source registers *Ra* and *Rb* if they do not coincide with the target register *Rr* of the previous instruction, represented by *Rr(EXop))*. Two new guards are introduced into the refined event to meet this requirement.

grd1 : ...
grd2 : ...
grd3 : $Rr(EXop) \neq Ra(ppop)$
grd4 : $Rr(EXop) \neq Rb(ppop)$

The Rodin prover now shows that the invariant $ALUout = Regs(Ra(EXop)) + Regs(Rb(EXop))$ is preserved by the refined machine.

3.4 Dealing Correctly with the RAW Hazard

It is now necessary to deal with the cases where a hazard is encountered on register *Ra* alone, on register *Rb* alone and on both registers *Ra* and *Rb*. In each case, the required value(s) can be read from the *ALUout* register. This corresponds directly to the *forwarding* technique used in microprocessor design. Three extra events are introduced to deal with each case. For instance, for the hazard on register *Ra*, the guards of the event are

grd3 : $Rr(EXop) = Ra(ppop)$
grd4 : $Rr(EXop) \neq Rb(ppop)$

and the associated action now reads the value of *Ra* from *ALUout*.

act2 : $ALUout := ALUout + Regs(Rb(ppop))$

The Rodin prover shows that, for each case, the invariant is preserved. The microarchitecture of the modified refined machine is shown in Figure 3.

3.5 Further Refinements

The refinement process can continue, systematically, until all the pipeline stages are represented in concrete form. At each step, the gluing invariants will ensure that the refinement implements its predecessor.

In the second refinement, the concrete Execute *(EX)* stage is introduced together with the *IDEX* pipeline registers. The registers *A* and *B* store the values of *Ra* and *Rb* respectively. Four events in the abstract *Fetch/Decode* stage are needed to deal with the possible data hazard combinations and two new gluing invariants,

inv1 : $A = Regs(Ra(IDop))$
inv2 : $B = Regs(Rb(IDop))$

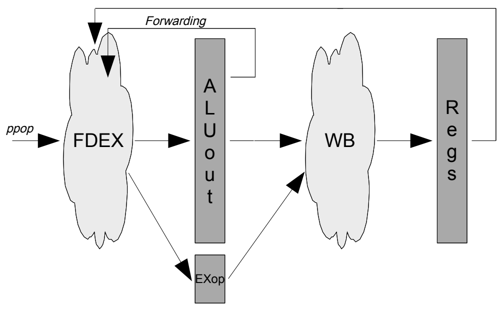


Fig. 3. Refined Machine with forwarding: Microarchitecture

ensure that the data hazards are dealt with correctly. When combined with the four *EXWB* events, this gives a total of sixteen events.

In the third refinement, the concrete Instruction Fetch *IF* and Instruction Decode *ID* are established.

To generalise this approach for uninterpreted arithmetic operations, the action

$\texttt{act1}: \ Regs(Rr(p)) := Regs(Ra(p)) + Regs(Rb(p))$

can be replaced with

$\texttt{act1}: \ Regs(Rr(p)) := fop(Regs(Ra(p)) \mapsto Regs(Rb(p)))$

where

$\texttt{grd1}: \ fop = func(p)$

and

$\texttt{axm8}: \ func \in Op \rightarrow Register$

is a field of the arithmetic instruction. The proofs with arbitrary arithmetic operations are still automatic.

The final, concrete pipeline is represented by sixteen events and all the proof obligations generated are discharged automatically by the Rodin tool, as shown in Table 1.

4 Related Work

Early work in the formal verification of microprocessors was focused on simple, non-pipelined processors described at the Register Transfer Level (RTL). In [13]

	Total no. of proof obligations	Discharged Automatically
Abstract Model	3	3
1st Refinement	33	33
2nd Refinement	192	192
3rd Refinement	115	115

Table 1. Pipeline Proofs

the RTL is represented in the ML programming language and the HOL proof assistant system [14] used to discharge the proofs.

In [12] and [15] the representation of the processor is raised to the Instruction Set Architecture (ISA) level and the techniques described focus on the formal verification of the control logic of first a 3-stage pipelined ALU and then the full 5-stage DLX processor. ALU operations are represented as uninterpreted functions. In order to show that the pipelined processor will behave in the same way as a notional non-pipelined version, the concept of pipeline *flushing* is introduced. *Stall* instructions are introduced at the pipeline input to ensure that each instruction is completed before the next is initiated. The notion of *refinement maps* are introduced in [16] and [17] to extend the flushing concepts of Burch and Dill to more complex 3 and 10-stage pipelines, using the ACL2 functional programming language and theorem prover [18].

[19] focuses its attention on the formalization of the pipeline hazards that can occur when multiple instructions are executed at once in the DLX pipeline. Structural, data and control hazards are represented and checked using the HOL verification system [14]. Incremental design techniques with refinement are described in [20] to show that a notional DLX pipeline that executes one instruction at a time can be refined to a pipeline that executes 5 instructions at each clock cycle and manages structural hazards does not encounter a sequence of instructions that would incur data or control hazards. This pipeline is then further refined to model the data and control hazards. Abstract State Machines (ASMs) are used to represent the DLX instructions. In [9], a tool that takes a sequential model of the DLX pipeline, which is assumed to be correct, and adds the forwarding logic is described. The tool also provides a proof of correctness for the generated hardware. Our approach is the only one that starts with an abstract ISA specification and proves, systematically, that the concrete, concurrent pipeline model derived from the ISA implements that specification.

5 Conclusions

A method has been explored, using the *register/register arithmetic* instruction as an example, to show that the ISA specification of the instruction can be refined systematically to a pipelined model that can be proved to implement its ISA specification. The method ensures, through the introduction of gluing invariants at each stage, that microarchitectural considerations are addressed early in the design flow. Different microarchitectures may be explored and verified at the specification level. Stepwise refinement allows us to manage the multiplicity of cases caused by pipeline data hazards. The models have been developed using the Rodin Platform and all the generated proof obligations are discharged automatically by the tool.

Current work is focused on managing the effect of branch instructions on correct pipeline execution. The techniques described have been used to prove that the pipeline program counter is updated correctly according to the branch instruction ISA specification. Gluing invariants are being developed to ensure that instructions that have been fetched speculatively are not executed when a branch is encountered.

A disadvantage of our approach is that we need to specify separate pipeline stages with a single event. We are exploring a technique that uses refinement and decomposition to create separate events for each stage once the gluing invariants have been proved.

In common with Bluespec and CAL, Event-B is based on guarded atomic actions. The method proposed therefore has the potential to be integrated into an existing high-level synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

References

1. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann (2006)
2. Hollander, Y., Morley, M., Noy, A.: The e language: A fresh separation of concerns. Proceedings of TOOLS **38** (2001)
3. Haque, F., Michelson, J.: Art of Verification with VERA. Verification Central (2001)
4. Nikhil, R.: Bluespec System Verilog: efficient, correct RTL from high level specifications. Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on (2004) 69–70
5. Bhattacharyya, S.S., Brebner, G., Janneck, J.W., Eker, J., von Platen, C., Mattavelli, M., Raulet, M.: Opendf: a dataflow toolset for reconfigurable hardware and multicore systems. SIGARCH Comput. Archit. News **36** (2008) 29–35
6. Abrial, J., Mussat, L.: Introducing dynamic constraints in B. B **98** (1998) 83–128
7. Hallerstede, S.: Justifications for the Event-B Modelling Notation. In: B 2007: Formal Specification and Development in B. (2007)
8. Abrial, J., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: International Conference on Formal Engineering Methods (ICFEM). (2006)
9. Kroening, D., Paul, W.: Automated pipeline design. In: Proceedings of the 38th conference on Design automation, ACM New York, NY, USA (2001) 810–815
10. Abrial, J.: Rigorous Open Development Environment for Complex Systems: event B language. (2005)
11. Evans, N., Butler, M.: A Proposal for Records in Event-B. FM (2006) 21–27
12. Burch, J., Dill, D.: Automatic verification of Pipelined Microprocessor Control. Proceedings of the 6th International Conference on Computer Aided Verification (1994) 68–80

13. Joyce, J., Birtwistle, G., Gordon, M.: Proving a Computer Correct in Higher Order Logic, Report No. 100. Computer Laboratory, Cambridge University (1986)
14. Gordon, M., Melham, T.: Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press New York, NY, USA (1993)
15. Jones, R., Dill, D., Burch, J.: Efficient validity checking for processor verification. In: IEEE International Conference on Computer-Aided Design, San Jose, California, USA (1995)
16. Manolios, P.: Correctness of pipelined machines. Formal Methods in Computer-Aided Design–FMCAD **1954** (2000) 161–178
17. Manolios, P., Srinivasan, S.: A computationally efficient method based on commitment refinement maps for verifying pipelined machines models. ACM-IEEE International Conference on Formal Methods and Models for Codesign (2005) 189–198
18. Kaufmann, M., Moore, J.: Industrial proofs with acl2. Technical report, University of Texas (2004)
19. Tahar, S., Kumar, R.: Formal Verification of Pipeline Conflicts in RISC Processors. Proc. European Design Automation Conference (EURO-DAC94), Grenoble, France, September (1994) 285–289
20. Borger, E., Mazzanti, S.: A Practical Method for Rigorously Controllable Hardware Design. ZUM'97, the Z Formal Specification Notation: 10th International Conference of Z Users, Reading, UK, April 3-4, 1997: Proceedings (1997)