A General Decomposition Strategy for Verifying Register Renaming*

Hazem I. Shehata and Mark D. Aagaard

Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, CANADA

ABSTRACT

This paper describes a strategy for verifying data-hazard correctness of out-of-order processors that implement register-renaming. We define a set of predicates to characterize register-renaming techniques and provide a set of model-checking obligations that are sufficient to guarantee that a register-renaming technique satisfies data-hazard correctness. We demonstrate how two register renaming techniques (retirement-register-file and dual-RAT) instantiate our predicates, and present model checking results for the Dual-RAT technique, which is based on the Intel Pentium® 4 processor.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Verification; C.1.1 [**Computer Systems Organization**]: Register renaming; I.2.3 [**Deduction and Theorem Proving**]: Theorem proving and model checking

General Terms

Verification, Algorithms, Experimentation

Keywords

Register renaming, Formal design verification, Pipelined circuits.

1. INTRODUCTION

This paper describes a strategy for verifying data-hazard correctness for out-of-order processors that implement register renaming. We begin with a general definition of data-hazard correctness, and decompose it into thirty verification obligations that give sufficient conditions for correctness of register-renaming techniques. These conditions are designed to hold for register-renaming techniques in general, and to be model-checked efficiently.

PipeOk is a formal definition of correctness for pipelined circuits. It is based on the conventional concepts of structural hazards, control hazards, data hazards, and datapath functionality [1]. PipeOk guarantees both Burch-Dill flushing correctness [4] and flushpoint-equality correctness (i.e., testing that the implementation matches the specification whenever the implementation is in a flushed state).

PipeOk contains thirteen correctness properties: three for structural hazards, six for data hazards, one for datapath functionality, and three for flushing. Control hazards are incorporated into both structural and data hazards. Because the properties are relatively orthogonal in the behavior they describe, different properties are amenable to different verification and abstraction techniques, which helps improve the scalability of the verification. The PipeOk formalization of pipelines augments a state machine with pipeline-specific functions and predicates, such as the map between implementation variables and specification variables. The PipeOk decomposition into multiple properties is based upon characterizing the required behaviour of the functions and predicates.

We define a generic high-level-model of out-of-order processors that captures data-dependency behaviour for register renaming. Using this model, we instantiate the PipeOk functions and predicates with predicates that capture behaviour that is common to register renaming techniques in general, such as busy and valid bits for each physical storage location. Based on these register renaming predicates, we decompose the six data-hazard correctness properties of PipeOk into thirty model checking obligations. We have proved that any register-renaming technique that satisfies our thirty obligations is guaranteed to satisfy the six data-hazard correctness properties in PipeOk. All but one of our obligations is an invariant or a single-step property. The one remaining obligations can be further decomposed using implementation-specific information.

To validate the generality of our approach, we have used our register-renaming predicates and functions to model register-renaming techniques based on the Control Data 6600 scoreboard [15], the original Tomasulo algorithm [16], renaming with a reorder buffer and retirement register file (RRF), and the Intel Pentium® 4 processor [5]. We call this last technique "Dual RAT" register renaming, because two Register Alias Tables are used. Similar techniques were used on the MIPs R10000 [18] and Alpha 21264 [8]. In Section 3, we explain our models of the two most complex, and common, techniques: RRF and Dual-RAT.

To validate the effectiveness of our approach, we used our obligations to verify the Dual-RAT register renaming algorithm (Section 4). We chose the Dual-RAT technique, because of its novelty and because it posed new verification challenges in model checking complexity and in relating the implementation to the specification. Our model preserves behavior related to register renaming, but abstracts away behavior related to datapath computation, control hazards, and structural hazards. For an implementation with five physical registers and two architected registers, all but one of the obligations can be verified in under an hour total.

*This research was funded by Intel, the Semiconductor Research Corporation (SRC), the Natural Science and Engineering Research Council of Canada (NSERC), and a scholarship from the Egyptian government.

DAC 2004, June 7–11, 2004, San Diego, California, USA.

2. DATA-HAZARD CORRECTNESS

Of the thirteen properties in PipeOk, the six listed below are for data-hazard correctness. Together, these six properties guarantee that all producer-consumer data-dependencies in the specification are obeyed in the implementation [1].

RawHazOk Read-after-write orderings in the specification are preserved in the implementation.

WawHazOk Write-after-write orderings in the specification are preserved in the implementation.

WarHazOk Write-after-read orderings in the specification are preserved in the implementation.

SpecRdTotFun Each read operation in the specification corresponds to exactly one read operation in the implementation.

SpecWrTotFun Each write operation in the specification corresponds to at least one write operation in the implementation, and to at most one write operation per implementation storage location.

ImplWrTotFun Each write operation in the implementation that can be read corresponds to exactly one write operation in the specification.

The PipeOk correctness properties are written in terms of predicates and functions that are instantiated by each pipeline. The data-hazard properties are written in terms of five predicates: a specification write ($\mathbf{Wr_s}$), a specification read ($\mathbf{Rd_s}$), an implementation write ($\mathbf{Wr_i}$), an implementation read ($\mathbf{Rd_i}$), and an address map ($\mathbf{AM}$) to relate the address/index of read and write operations in the implementation to their corresponding address/index in the specification.

3. MODELLING REGISTER RENAMING

In this section, we describe our generic model of register renaming, and then demonstrate how this model can be instantiated for retirement-register-file (RRF) register renaming and Dual-RAT register renaming. Our model is focused on data-dependency behavior. We abstract away details that are irrelevant to data-hazard correctness: for example, we do not model structural or control hazards and each instruction contains only one source operand.

3.1 Generic Processor for Register Renaming

Our processor model contains six pipeline units and a data-storage module (Figure 1). The pipeline units may themselves be pipelined and may produce instructions out of order. Read operations occur at the time of dispatch. The instantiation of implementation write is dependent upon the register renaming technique, and so is defined separately in Sections 3.2 and 3.3. A physical register becomes busy when it is allocated as a physical destination. The register becomes valid at writeback. After an instruction retires, its destination is marked as not busy. The register remains valid (i.e. represents the committed state) until the retirement of a subsequent instruction with the same architectural destination. After the register is neither valid nor busy, it is freed and can be reallocated.

We first define three predicates to capture concisely the behaviour on the ports of the pipeline units (Table 1). We use the port predicates to instantiate the PipeOk predicates $\mathbf{Rd_s}$, $\mathbf{Wr_s}$, $\mathbf{Rd_i}$, and $\mathbf{AM}$ for data hazards (Table 2). The instantiation of implementation write is dependent upon the register renaming technique, and so is defined separately in Sections 3.2 and 3.3. Read and write operations occur in the specification when the implementation issues the corresponding instruction. In the implementation, read operations occur at the time of dispatch.

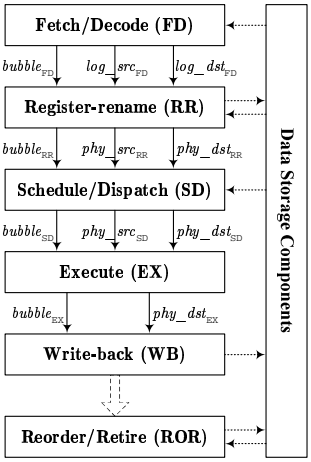


Figure 1: Processor for general register renaming

Port predicates
Bubble exiting unit u $\mathcal{B}_u(t : \texttt{Time}) \equiv bubble_u(t)$
Physical destination of instruction exiting unit u $\mathcal{D}_u(t : \texttt{Time}, h : \texttt{PhysIdx}) \equiv (h = phy\_dst_u(t)) \wedge \neg\mathcal{B}_u(t)$
Physical source of instruction exiting unit u $\mathcal{S}_u(t : \texttt{Time}, h : \texttt{PhysIdx}) \equiv (h = phy\_src_u(t)) \wedge \neg\mathcal{B}_u(t)$

Table 1: Port predicate shortcuts

Instantiations of data-hazard predicates
Specification read $\mathbf{Rd_s}(t : \texttt{Time}, m : \texttt{ArchIdx}) \equiv \mathcal{S}_{\text{FD}}(t, m)$
Specification write $\mathbf{Wr_s}(t : \texttt{Time}, m : \texttt{ArchIdx}) \equiv \mathcal{D}_{\text{FD}}(t, m)$
Implementation read $\mathbf{Rd_i}(t : \texttt{Time}, h : \texttt{PhysIdx}) \equiv \mathcal{S}_{\text{SD}}(t, h)$

Table 2: Generic register-renaming predicates

Each register renaming technique provides its own instantiation of the the pipeline units and data-storage module. In our generic model, the pipeline units are characterized by the interface signals in Figure 1, such as phy_src_{RR}, which is the physical source index output from the renaming unit. The data-storage module contains register-alias tables, the register file, the reorder buffer, etc. The data-storage module is characterized by three predicates for each physical register: $\mathbb{B}$: busy, $\mathbb{V}$: valid, and $\mathbb{F}$: speculative mapping between an architectural register and physical register. By "physical register", we mean any physical storage location: entries in reorder buffer, register-rename files, speculative register files, retirement register files, and pooled register files.

3.2 Retirement-Register-File Renaming

The most common register-renaming technique in formal verification papers is the retirement-register-file (RRF) technique. Although often called "Tomasulo's algorithm", RRF renaming differs from Tomasulo's original algorithm as implemented on the IBM 360/91 [16]. RRF register renaming uses a register-alias table (RAT), reorder buffer (ROB), and retirement register file (RRF). The RRF maintains the committed state for the architected registers. Speculative results are stored in the ROB, pending retirement. The RAT represents the speculative state of the architected registers by mapping each architected register to either the register file (no in-flight instruction will write to the register) or to an entry in the ROB.

Table 3 gives the RRF instantiations of the register-renaming predicates. Implementation writes to the ROB are done in the write-back unit and writes to the RRF are done when the instruction retires (exits the ROB). A ROB entry is busy (corresponds to an in-flight instruction) if it is between the head and tail pointers. A register-file entry is busy if the ROB entry pointed to by the FRAT is busy. A ROB entry is valid if it is busy and its valid bit is true. The validity of a register file entry is determined directly by its valid bit. The FRAT mapping points to either the register file or the ROB, depending on whether the register file entry is valid.

$\mathbf{Wr_i}(t : \texttt{Time}, h : \texttt{RobIdx})$	$\equiv \mathcal{D}_{\texttt{EX}}(t, h)$
$\mathbf{Wr_i}(t : \texttt{Time}, h : \texttt{RFIdx})$	$\equiv \mathcal{D}_{\texttt{WB}}(t, h)$
$\mathbb{B}(t : \texttt{Time}, h : \texttt{RobIdx})$	$\equiv \texttt{Hd}(t) > h \geq \texttt{Tl}(t)$
$\mathbb{B}(t : \texttt{Time}, h : \texttt{RFIdx})$ $\texttt{Hd}(t) > \texttt{Rat}[h](t) \geq \texttt{Tl}(t)$	$\equiv$
$\mathbb{V}(t : \texttt{Time}, h : \texttt{RobIdx})$	$\equiv \mathbb{B}(t, h) \wedge \texttt{Rob}[h].v(t)$
$\mathbb{V}(t : \texttt{Time}, h : \texttt{RFIdx})$	$\equiv \texttt{RF}[h].\texttt{v}(t)$
$\mathbb{F}(t : \texttt{Time}, h : \texttt{RFIdx}, m : \texttt{ArchIdx})$ $\textbf{if } \texttt{RF}[m].\texttt{v}(t) \textbf{ then } h = m \textbf{ else } h = \texttt{Rat}[h](t)$	$\equiv$
$\mathbf{AM}(t : \texttt{Time}, h : \texttt{RFIdx}, m : \texttt{ArchIdx})$	$\equiv h = m$
$\mathbf{AM}(t : \texttt{Time}, h : \texttt{RobIdx}, m : \texttt{ArchIdx})$	$\equiv \mathbb{B}(t, h) \wedge \mathbb{F}(t, h, m)$

Table 3: Instantiations of register-renaming for RRF

3.3 Dual-RAT Renaming

As its name implies, the Dual-RAT register-renaming technique uses two register-alias tables: a front-end register-alias table (FRAT) and a rear register-alias table (RRAT). The FRAT serves the same purpose as the RAT in the RRF renaming technique: it maintains a mapping from architected registers to speculative state. Both speculative and committed state are stored in the same physical register file. The RRAT is updated when an instruction retires.

Each physical register has a valid bit, busy bit, and an index for the architected register that it represents. Storing the architected index in the register file, rather than in the ROB, eliminates one need for a CAM match on the ROB. The size of the ROB is equal to the difference between the number of physical and architected registers, because the ROB holds information only for in-flight instructions.

Table 4 lists the Dual-RAT instantiations of the register-renaming predicates. Implementation writes happen only in the writeback unit. The register file has busy and valid bits. The FRAT always maps architected registers to physical registers, as opposed to RRF renaming, where the RAT mapped architected registers to either the RRF or the ROB. For the address map, if the physical register is busy (instruction is in flight) then we use the register file to lookup the architected index. If the instruction has already retired, we need to be sure that it has not been superseded by a more recent instruction writing to the same architected register, so we check the RRAT.

Instantations of register-renaming for Dual-RAT	
$\mathbf{Wr_i}(t : \texttt{Time}, h : \texttt{PhysIdx})$	$\equiv \mathcal{D}_{\texttt{EX}}(t, h)$
$\mathbb{B}(t : \texttt{Time}, h : \texttt{PhysIdx})$	$\equiv \texttt{RF}[h].\texttt{b}(t)$
$\mathbb{V}(t : \texttt{Time}, h : \texttt{PhysIdx})$	$\equiv \texttt{RF}[h].\texttt{v}(t)$
$\mathbb{F}(t : \texttt{Time}, h : \texttt{PhysIdx}, m : \texttt{ArchIdx})$	$\equiv \texttt{FRat}[m](t) = h$
$\mathbf{AM}(t : \texttt{Time}, h : \texttt{PhysIdx}, m : \texttt{ArchIdx})$ $\textbf{if } \mathbb{B}(t, h) \textbf{ then } m = \texttt{RF}[h].\texttt{a}(t) \textbf{ else } h = \texttt{RRat}[m](t)$	$\equiv$

Table 4: Predicates for Dual-RAT

4. VERIFICATION

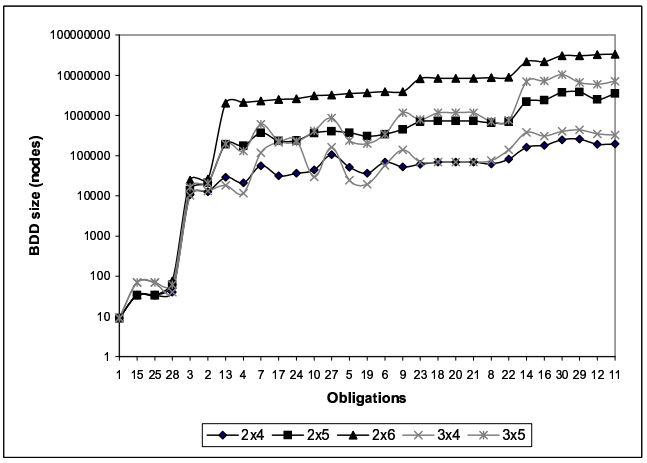
For register renaming techniques, write-after-write and write-after-read ordering correctness properties are easy to prove by showing that a register will be allocated only if there are no in-flight instructions with it as a destination. The challenge lies in the remaining four properties: read-after-write correctness and three properties for a one-to-one mapping between read and write operations in the specification and implementation.

Each of the data-hazard properties uses nested "eventually" and "until" temporal operators and refers to the behaviour of the entire system. To make it feasible to use model checking to verify the behaviour of a register-renaming algorithm, we decomposed the data-hazard correctness properties into thirty model checking obligations. Each obligation refers only to a few units in the processor and all except one obligation are either invariants or single-step properties [14]. This one property can be decomposed into invariants and single step properties by exposing implementation-specific details about the ROB. Because our goal was to create a decomposition that would hold for any renaming technique, we did not decompose this obligation any further.

To validate the effectiveness of our decomposition strategy, we verified a high-level model of the Dual-RAT algorithm. This algorithm poses new verification challenges in specification and model-checking complexity. In RRF renaming, the physical destination is identical to the ROB index and the relative age of two instructions is determined by the relative positions of their physical destinations and the head and tail pointers in the ROB. In Dual-RAT renaming, the physical destination is independent of the ROB entry. Comparing the ages of two instructions requires a CAM match on the ROB, which is expensive in model checking. Another complication arises from the lack of an architected-register file. Projecting the externally visible state of the implementation requires a data-dependent projection function. For each architected register, we must use the RRAT to look up the corresponding physical register. The address map function in PipeOk, which was originally designed for bypass paths, works equally well for the dynamic mapping between committed physical registers and architected registers.

Our model implements the units (register-rename, write-back, and reorder/retire) that interact with the storage module. For the other three units (fetch/decode, schedule/dispatch, and execute), we have defined characteristic properties, such as liveness and a one-to-one mapping between entering and exiting instructions, that serve as sufficient environmental assumptions for our verification.

Figure 2 shows the BDD sizes of each obligation (sorted by increasing size) for different numbers of architected and physical registers. The most space-consuming configuration that we could run reasonably was 2 architected registers and 6 physical registers (2x6). For the most complex of these obligations, the BDDs peaked at around 30M nodes and runtimes peaked at about 10 hours on an 1.8GHz Intel Pentium® III processor. Ignoring the four simplest obligations, there is a factor of about 20 between the largest and smallest obligations for a given configuration. Given the breadth of obligations verified, this seems reasonable. A smaller spread would be desirable, but additional, specialized, decomposition is always possible if an obligation exceeds available memory or time. In conducting the model checking, the only assumption that was needed was that the fetch/decode unit outputs an instruction only if there is a free register. The other assumptions were used in the proof. The average six largest obligations (14, 16, 30, 29, 12, and 11) are distinguished by referring to a specification operation and the FRAT. This indicates that a fruitful place to look for further decomposition might be in the interaction between fetch/decode and register-rename.



"nxk" means n architected registers and k physical registers.

Figure 2: BDD sizes of model-checking obligations

5. CONCLUSION

We have developed a general decomposition strategy for verifying data-hazard correctness of register-renaming techniques. We demonstrated the generality and effectiveness of our strategy by modeling the retirement register file and both modeling and verifying Dual-RAT register-renaming.

High-level models of the original Tomasulo algorithm have been verified by McMillan [11], Berezin [3], and Arons and Pnueli [2]. High-level models of register renaming with a reorder buffer and retirement register file (RRF) have been verified by Skakkebæk, Jones, and Dill [7], Pnueli and Arons [12], Hosabettu, Srivas, and Gopalakrishnan [6], Sawada and Hunt [13], Velev [17], and Lahiri and Bryant [9]. For both Tomasulo's algorithm and RRF renaming, when the processor is flushed, the projection from the implementation state to the specification state is just a direct projection of the register file and other state variables, such as the program counter. For Dual-RAT register renaming, even when the machine is flushed, the mapping from implementation state to architectural state is data-dependent: the RRAT must be used to determine the mapping between physical and architected registers. The only previous use of a data-dependent address map in processor verification was by Arons and Pnueli [12]. Their map chose between the reset value and the actual value of the state, depending on whether the machine is flushed. A second challenge in verifying Dual-RAT register renaming is that determining the program order (i.e., age) of two instructions requires a CAM match on the reorder buffer, rather than just comparing the physical destinations of the instructions to the head and tail pointers in the ROB. CAM matches are expensive, both in hardware and in model checking.

All of our model-checking obligations are safety properties; all but one are either invariants or single-step properties. We were able to verify the obligations without resorting to structural decomposition or assume-guarantee reasoning. The environmental assumptions about the unimplemented units (fetch/decode, schedule/dispatch, and execute) were used in the proof that the model-checking obligations guarantee data-hazard correctness. For our model, only one environmental assumption was needed in conducting the model checking. These features helped reduce the complexity of model checking, and should be beneficial in verifying other implementations of register renaming.

Our model checking obligations are defined in terms of behavior at the interface between units. This makes the obligations relatively insensitive to low-level design optimizations or changes. We hope that the obligations are understandable and might be helpful in creating test plans and evaluating the correctness of new register renaming algorithms.

6. REFERENCES

[1] M. D. Aagaard. A hazards-based correctness statement for pipelined circuits. In *CHARME*, pp 66–80. 2003.

[2] T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. In *Int'l Conf. on VLSI Design*, pp 92–99, 1999.

[3] S. Berezin, *et al*. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *FMCAD*, pp 369–386. 1998.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, pp 68–70. 1994.

[5] G. Hinton, D. Sager, U. Mike, and D. Boggs. The microarchitecture of the Pentium® 4 processor. *Intel Tech. Jour.*, Q1, Feb. 2001.

[6] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *CAV*, pp 47–59. 1999.

[7] R. Jones, J. Skakkebæk, and D. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *FMCAD*, pp 2–17. 1998.

[8] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, Mar/Apr 1999.

[9] S. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *CAV*, pp 341–354. 2003.

[10] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[11] K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV*, pp 110–121. 1998.

[12] A. Pnueli and T. Arons. Verification of data-insensitive circuits: An in-order-retirement case study. In *FMCAD*, pp 351–368. 1998.

[13] J. Sawada and W. A. Hunt. Verifying the FM9801 microarchitecture. *IEEE Micro*, 19(3):47–55, May/June 1999.

[14] H. I. Shehata and M. D. Aagaard. A verification strategy for register renaming (extended version). Technical report 2004-12, E&CE, Univ. of Waterloo, Mar. 2004.

[15] J. E. Thornton. Parallel operation in the Control Data 6600. In *Proc. of the AFIPS, vol II*, volume 26, pp 33–40, 1964.

[16] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Jour. of Res. and Dev.*, 11:25–33, Jan. 1967.

[17] M. N. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *DATE*, pp 28–35, Mar. 2002.

[18] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.