Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Imprecise Data-Memory Exceptions

Sudarshan K. Srinivasan
darshan@ece.gatech.edu

Miroslav N. Velev
mvelev@ece.gatech.edu

School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332, U.S.A.

Abstract

We present the formal verification of an Intel XScale processor model. The XScale is a superpipelined RISC processor with 7-stage integer, 8-stage memory, and variable-latency multiply-and-accumulate execution pipelines. The processor uses scoreboarding to track data dependencies, and implements both precise and imprecise exceptions. Such set of features has not been modeled, and formally verified previously. The formal verification was done with an automatic tool flow that consists of the term-level symbolic simulator TLSim, the decision procedure EVC, and an efficient SAT-checker.

1. Introduction

In striving to increase the performance of embedded processors like the Intel XScale [7][16], designers adapt mechanisms that were previously used only in out-of-order superscalar processors. The Intel XScale is a superpipelined RISC microprocessor compliant with ARM Version 5TE ISA [4]. It has applications from handheld internet devices to enterprise internet infrastructure products. Some examples of its applications are HP's iPAQ pocket PC H3900 series [3], handheld devices from Fujitsu, Toshiba and Casio Computer [28], Dell Axim X5 handheld [8], and Intergrity RTOS that uses the Intel XScale to deliver high reliability [17]. Other processors like the MIPS32 4Kp [10], and the Motorola M•CORE [24] are similar to the 5-stage DLX pipeline. However the XScale is superpipelined with 7-stage integer, 8-stage memory, and variable latency memory-and-accumulate (MAC) pipelines. Also, it has features such as register scoreboarding, out-of-order execution and completion, and imprecise exceptions, which make it more complex for formal verification.

We check the safety property of pipelined XScale variants with increasing complexity, using Burch and Dill's flushing technique [6] as a way to automatically compute the abstraction function. A term-level symbolic simulator TLSim [41], takes the specification and implementation processors, and a simulation-command file that indicates how to simulate the specification and implementation, and produces a formula in the logic of Equality with Uninterpreted Functions and Memories (EUFM) [6]. The decision procedure EVC (Equality Validity Checker) [41], converts the EUFM formula to a propositional formula, which is then evaluated by a Boolean satisfiability procedure [40][42] such as Chaff [26] or BerkMin [11]. The tool flow, consisting of TLSim, EVC, and an efficient SAT-checker has been used both in industry—Lahiri et al. [21] applied the tool flow at Motorola, detecting 3 bugs, and corner cases that were not implemented in a model of the M•CORE processor—and in academia to teach the principles of pipelined, speculative, and superscalar execution [43].

The control logic for detecting, and dealing with data hazards is one of the critical timing paths. With increase in issue width, pipeline depth, and a need to optimize the design for performance, the logic for detecting hazards becomes more complex, and difficult to fit within the stringent timing constraints. Therefore, in state-of -the-art processors such as the Itanium [30], alternate techniques to handle data hazards becomes necessary. The Itanium processor, which has a 6-wide front end, 9-wide execution engine, and 10 stage deep pipeline, uses a stall-based scoreboard control strategy to overcome the complexity of the control logic, and satisfy timing requirements. Since the Intel XScale is superpipelined, it faces similar considerations for detecting hazards, and therefore it uses a scoreboard to handle data hazards occurring in the main execution, parallel memory, and MAC pipelines.

2. Background

We prove the safety property of an implementation that can issue up to k instructions per clock cycle—if the implementation makes one step of regular operation, starting from an arbitrary initial state Q_{Impl}, then the new implementation state Q'_{Impl} corresponds to the state of the specification after 0, or 1, ... , or k steps when starting from state Q^0_{Spec} that corresponds to the initial imple-

Appears in the proceedings of *Formal Methods and Models for Codesign (MEMOCODE '03)*, June 2003, pp. 65–74.

mentation state Q_{Impl}. By flushing the implementation [5][6]—feeding it with bubbles (control signals that would not modify the architectural state) until all partially executed instructions are completed—we map an implementation state to an equivalent specification state. Before flushing, the implementation makes one step, which we term as the one cycle of regular operation.

The implementation and specification processors are described in the high-level hardware description language AbsHDL [41]. The language has constructs for basic logic gates such as `and`, `or`, `not`, `mux`, and equality comparators. It also has constructs for memories, latches, and for abstracting functional units. It differs from other HDLs such as Verilog and VHDL in that the bit-width of word-level values, and the implementation of memories and combinational functions are not specified.

The syntax of EUFM consists of terms and formulas. A term can be a variable, an Uninterpreted Function (UF) applied to a list of arguments, or an *ITE* operator that selects between two terms based on a formula, such that *ITE*(*formula*, *term1*, *term2*) evaluates to *term1* if *formula* is *true*, and to *term2* if *formula* is *false*. Terms are used to abstract data of arbitrary length, and they do not satisfy any particular property other than the *property of equality* (i.e., two terms can be compared for equality), which is required when defining the logic for forwarding or load interlock, etc. A formula can be a propositional variable, an Uninterpreted Predicate (UP) applied to a list of argument terms, or an *ITE* operator, which selects between two formulas based on a controlling formula. Formulas can be negated, and connected by Boolean connectives.

Combinational functional units are abstracted using Uninterpreted Functions (UFs), and Uninterpreted Predicates (UPs)—black boxes that only satisfy the property of *functional consistency*, i.e., if the inputs of two applications of an UF are equal, then their outputs are equal. The difference between an UF, and an UP is that the former is a term, and the latter a formula.

The syntax for terms is extended to model memories using interpreted functions *read* and *write*. Function *read* takes a term for the memory state, and address as input, and returns the data corresponding to the address. Function *write* takes the memory state, address, and data as input to return the new memory state. Functions *read*, and *write* satisfy the *forwarding property* of memory semantics—that the *read* operation returns the data most recently written to an equal address, or otherwise the initial state of the memory for that address. The efficiency of EVC is due to the property of Positive Equality [41], stating that the truth of an EUFM formula under a maximally diverse interpretation of the term variables that appear only in positive (not negated) equality comparisons implies the validity of the formula under any interpretation of those term variables.

3. XScale Architecture

The Intel XScale core has superpipelined RISC architecture with 7-stage integer, 8-stage memory, and variable latency multiply-and-accumulate (MAC) pipelines. Figure 1 shows the pipeline organization of the XScale.

To allow for aggressive clocking of the processor, the instruction fetch is divided into two stages—F1 and F2. One instruction is delivered each cycle to the instruction decode stage ID, if the two fetch stages are not stalled. The Branch Target Buffer (BTB) is in F1, and predicts the direction and target of branch instructions. Once the branch reaches the ALU in EX1, the target address of the branch is computed. If that target address is different from the predicted target address, the pipeline is flushed, and execution continues at the actual target address. The ID stage is responsible for general instruction decoding such as extracting the opcode, operand addresses, source and destination identifiers, and control bits.

The RF (Register File/Shifter) stage is used to read, and write to the register file unit (RFU). It provides source data to the execute stage (EX) for ALU operations, to MAC for multiply operations, and to the memory pipeline for load/store instructions. The register identifiers from ID specify, which registers are accessed in the register file unit. If there is a pending update to a register that is accessed, and the result of the update is not yet available, the RFU stalls the pipeline. The RFU keeps track of pending updates to the register file using a scoreboard. The ARM architecture specifies one of the operands for an ALU instruction as the shift operand. A 32-bit shift can be performed on that operand by the shifter in RF, before it is used as input to the ALU.

ALU operations and address calculations for load/store instructions are performed in EX1. Conditional instruction execution is determined in EX1. Every instruction has an associated condition that is compared with the architecture flags. The flags are part of the program status register (PSR), and are updated by the previous instruction. An instruction, whose qualifying predicate is false, is cancelled, and is not allowed to update the architectural state. The direction and target of a branch operation are also determined in EX1. In case of a misprediction, instructions in all previous stages are squashed, and the PC is updated with the correct target address. A branch misprediction results in a 4-cycle penalty. The EX2 stage contains the PSR, which is updated by instructions in the main execution pipeline.

The memory pipeline consists of two pipestages MEM1 and MEM2, and handles load/store instructions. Memory operation (load and store) begins in MEM1 after the effective address for a load/store has been calculated in EX1. The MAC unit executes multiply and multiply-and-accumulate instructions. The latency of instructions in the MAC unit depends on the size of the input operands, and the instruction can take between 2 and 5 cycles.

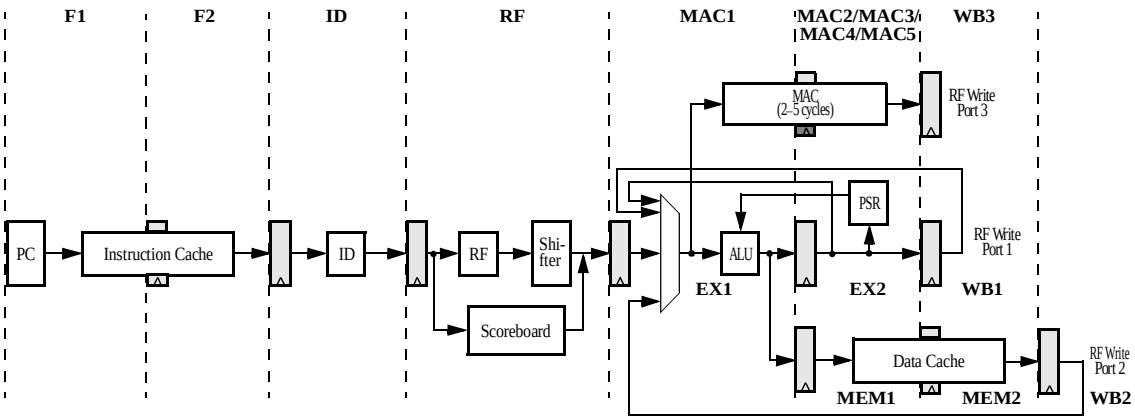

Figure 1. XScale pipeline organization.

Instructions are issued in-order, but as the 3 specialized execution pipelines have different execution latencies, instructions may complete out-of-order, provided that no data dependencies exist. Register scoreboarding is used to track data dependencies, and stall the pipeline in the RF pipestage.

4. Base Processor Model

4.1 Main Execution Pipeline

We started by modeling the Main Execution Pipeline (MEP), capable of executing only ALU instructions, and then extended this model with other abstract instruction types, and architecture features as presented in Sections 4 and 5. Conditional instruction execution, pipeline hazards, branch prediction, and instruction memory and ALU exceptions are key features of the MEP that are discussed in this Section.

Conditional instruction execution is implemented as in [39], using the term *Cond*, obtained from the instruction decoder, and the UP, *Cond_Evaluate* that evaluates the instruction condition. The flags, used to determine conditions, are abstracted in the program status register (PSR). The PSR is modeled as a latch, and is part of the architectural state. Its contents is read in EX1, and updated in EX2. The PSR is modeled as a read-after-write latch so as to allow the instruction in EX2 to update latch before its contents is read in EX1.

Handling read-after-write (RAW) hazards in the MEP requires both data forwarding to EX1, and stalling in RF. Two levels of forwarding from EX2 and WB1, to EX1 are used, and priority is given to results from EX2. The result of an instruction in EX1 is not available until it has reached EX2. Therefore, an instruction in RF that requires a shift operation, and depends on the instruction in EX1 has to be stalled in RF. The main execution pipeline does not exhibit write-after-write (WAW) or write-after-read (WAR) hazards, as all writes to the register file are in-order, and happen in WB1.

The branch predictor is modeled with a Finite State Machine (FSM), as done in [38]. On each clock cycle, this FSM produces an arbitrary term for the predicted target, and an arbitrary formula for the predicted direction, thus abstracting any implementation of a branch predictor. The XScale uses a Branch Target Buffer (BTB), which is abstracted as described above with a latch that holds an arbitrary state; UP, *PredictTakenBranch*, whose output gives the prediction for the branch direction; and UF, *PredictedTarget*, whose output is the target of the branch. Both *PredictTakenBranch*, and *PredictedTarget* take the present state of the FSM latch as input. Since the state of the latch is arbitrary, so are the predictions. What is verified is that, if the prediction is not correct then the implementation has a mechanism to correct the misprediction. The specification does not include a branch predictor, as it is used only to enhance the performance of the implementation processor.

Precise exceptions are modeled in the style of [38]. For every functional unit that can generate an exception, an UF is introduced with the same inputs as the functional unit, and output that is a formula indicating if an exception is raised. An exception status register that is part of the user visible state, is used for every type of exception. Also, an ExceptionPC latch is used to hold the PC of the excepting instruction. Since instructions in the different specialized pipelines have different execution latencies, it is possible that instructions complete out of order. A valid instruction in MAC2 or EX2 could have updated the architectural state before an older load instruction raises a data memory exception in WB2. As instructions may complete out of order before data memory exceptions are detected, data memory exceptions may be imprecise. ALU and instruction memory exceptions are raised before any younger instruction in program

order updates architectural state. Therefore, ALU and instruction memory exceptions are precise.

4.2 Memory Pipeline

Loads and stores are routed to the parallel memory pipeline after the data memory address is calculated in EX1. Data memory access consists of 3 stages: MEM1, MEM2, and WB2 (write back). In our model, all memory access operations are completed within two cycles in MEM1 and MEM2. Note that the latency of a load instruction that completes in WB2 is 8 cycles, and the latency of a ALU instruction that completes in WB1 is 7 cycles. An ALU instruction that follows a load in program order could complete in the same cycle as the load. The structural hazard due to simultaneous writes from both WB1, and WB2 is overcome by using two write ports. Instructions in WB1 are more recent in program order, and are therefore given priority over instructions in WB2, when writing to the register file. Results from loads are available only in WB2, from where they are forwarded to EX1.

An instruction that performs both branch, and load operations is possible, as there is no restriction on the specification that prevents more than one instruction type from occurring together. But this would lead to an invalid instruction exception in the actual processor. We term this as instruction overlapping. Such properties of the implementation can be exploited to prune its state space using invariant constraints. Non-overlapping instruction constraints are used to eliminate from consideration the processor behavior due to overlapping instructions.

A simple example of a non-overlapping constraint that excludes an instruction from being both a branch, and a store is shown. Let the control bits `is_Branch` and `MemWrite` indicate a branch and a store, respectively. Then, the constraint that an instruction cannot be both a branch and a store can be defined in the syntax of AbsHDL as:

```
constraint_Branch_or_Store =
  (or (not is_Branch) (not MemWrite))
```

The signal `constraint_Branch_or_Store` is used in the simulation-command file to impose the constraint, but is not used as part of the processor control logic. Constraints to exclude ALU, load, store, and branch instructions from occurring together are imposed on all stages of the pipeline in the first, and only cycle of regular operation of the implementation. These constraints are checked for invariance, i.e., the implementation processor will satisfy them after 1 clock cycle of regular operation. Constraints imposed in F1 on the outputs of the instruction memory are not checked for invariance, once those constraints are assumed to be properties of the program.

Instructions are routed to the integer or memory pipeline from EX1. Since we have imposed constraints on the implementation to eliminate overlapping instruction types from occurring, it is not possible for EX2 and MEM1 to both have a valid instructions in the same cycle. Similarly, both WB1 and MEM2 cannot have valid instructions in the same cycle. This property of the pipeline is exploited using exclusivity constraints, which impose the restriction that the a pair of parallel stages cannot both have a valid instruction in the same cycle.

In case of data dependencies, the dependent instruction is stalled in the RF stage. Depending on whether the pipeline is stalled or not, there is variability in how instructions flow through the pipeline. Controlled flushing [5] is used to schedule the pipeline so that this variability is removed only when flushing the pipeline in order to compute an abstraction function, resulting in a simpler EUFM correctness formula.

To implement controlled flushing, a new control input *force_stall* is introduced. During the one cycle of regular operation of the implementation, *force_stall* is set to *false*. An instruction in RF could be dependent on instructions in EX1, MEM1, and MEM2. The worst case stalling of 3 cycles occurs when an instruction I1 in RF is dependent on instruction I2 in EX1, in that case I1 will not be issued to EX1 until the result of I2 is available from WB2. During flushing, instructions in F1, F2, ID, and RF are stalled for 3 cycles by setting *force_stall* to *true,* and allowed to advance in the fourth cycle by setting *force_stall* to *false*. The same pattern of stalling is used for the new instruction in RF. This pattern is repeated until pipeline stages F1, F2, ID, and RF do not have valid instructions. The implementation is simulated for another 4 cycles to allow the pipeline to be flushed completely. To check that the pipeline is fully flushed, i.e., there are no pending updates to the architecture state after flushing, a signal *is_flushed* is defined as the AND of the negations of all control bits of all pipeline latches. If *is_flushed* is valid at the end of the flushing sequence, then the pipeline is completely flushed.

4.3 PC Logic

The XScale ISA specifies that ALU and load instructions are allowed to update the PC as it is part of the register file. We modeled this in two alternate ways. The register address of the PC is abstracted with a term variable, and the destination address of a ALU or a load instruction, and the PC address are checked for equality. Alternatively, an uninterpreted predicate with the destination address of the instruction as the only input is used to indicate if the instruction writes to the PC. In both of the above cases, the instruction is not allowed to update the register file.

Writes to the PC are detected in EX1. Since an operation that modifies the PC behaves like a branch, instructions in stages F1, F2, ID, and RF are squashed. ALU

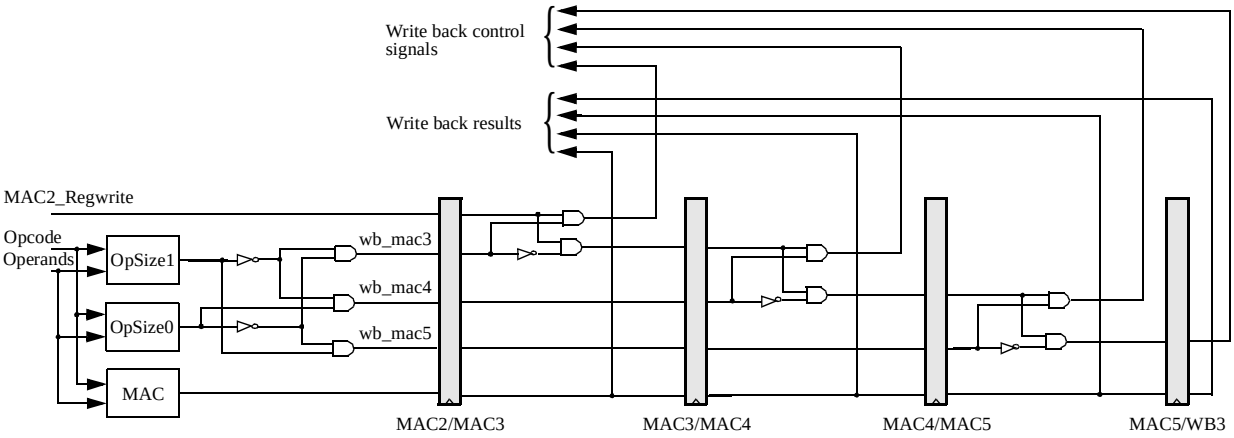

Figure 2. Implementation of the variable latency (2 to 5 cycles) MAC pipeline.

instructions that update the PC are treated as branch instructions, and complete in EX1. The result of a load is not available until the WB2 stage. Loads that write to the PC are detected in the EX1 stage, and all instructions in previous pipeline latches are squashed. Every new instruction that is fetched is also squashed until the load advances to WB2 when it can update the PC.

5. Modeling the Multiply-and-Accumulate Unit, Scoreboarding, and Imprecise Exceptions

5.1 Multiply and Accumulate Unit

The variable instruction execution latency of the multiply-and-accumulate unit (MAC) makes it complex to formally verify. Control flushing and invariant constraints are used to simplify the correctness formula, and reduce the time taken by the tool flow.

Multiply instructions are routed to the multiply-and-accumulate (MAC) pipeline from RF—consisting of six pipeline stages namely MAC1, MAC2, MAC3, MAC4, MAC5, and WB3. Figure 2 shows the implementation of the MAC pipeline with variable latency. The UF, MAC takes the opcode and operand as input, and produces a result. The variability in the number of execution cycles required by the MAC is abstracted using two UPs, *OpSize0* and *OpSize1* that take the opcode and operands as input. For example, if the both UPs are false, then it could mean a 3 cycles execution latency. The two bits indicating the size of the operands are used to generate 4 signals, *wb_mac3*, *wb_mac4*, *wb_mac5*, and *wb_wb3* that determine whether the multiply instruction will writeback in MAC3, MAC4, MAC5 or WB3, respectively. Note that for an instruction, only one of the 4 signals is *true*, and all the other signals are *false*. These signals are used to set or reset *RegWrite* (the control signal that indicates a write to the register file) in MAC3, MAC4, MAC5, or WB3.

Controlled flushing is modified for the implementation with the MAC pipeline. The worst case stalling occurs when an instruction I1 in RF depends on instruction I2 in MAC1, in that case I1 is stalled in RF for a maximum of 5 cycles for the result of I2 to become available from WB3. During controlled flushing, the instruction in RF is force-stalled for 5 cycles instead of the previous 3 cycles before it is allowed to advance. This pattern is repeated until there are no valid instructions in F1, F2, ID, and RF. The implementation is simulated for another 6 cycles to complete the flushing, as a valid instruction in MAC1 could take 6 cycles to complete.

The constraints for non-overlapping instruction types are extended to include multiply instructions. Exclusivity constraints are imposed on the groups of parallel stages including EX1-MAC1, EX2-MEM1-MAC2, WB1-MEM2-MAC3, and WB2-MAC4, i.e., only one of the stages in a parallel group is allowed to have a valid instruction in any given cycle.

5.2 Register Scoreboarding

The Intel XScale uses a register scoreboard to detect register dependencies between instructions, and simplify the control logic for handling pipeline hazards. In order to hide the scoreboard from the specification, we had to initialize the scoreboard to reflect the current state of the implementation.

The scoreboard stalls the pipeline in RF for both read-after-write (RAW), and write-after-write (WAW) hazards. Write-after-write hazards are present as instructions may complete out of order in the MAC pipeline with respect to instructions in the parallel memory pipeline.

The register scoreboard is implemented as a memory element. All registers in the register file have a corresponding one bit entry in the Scoreboard. If the bit-entry

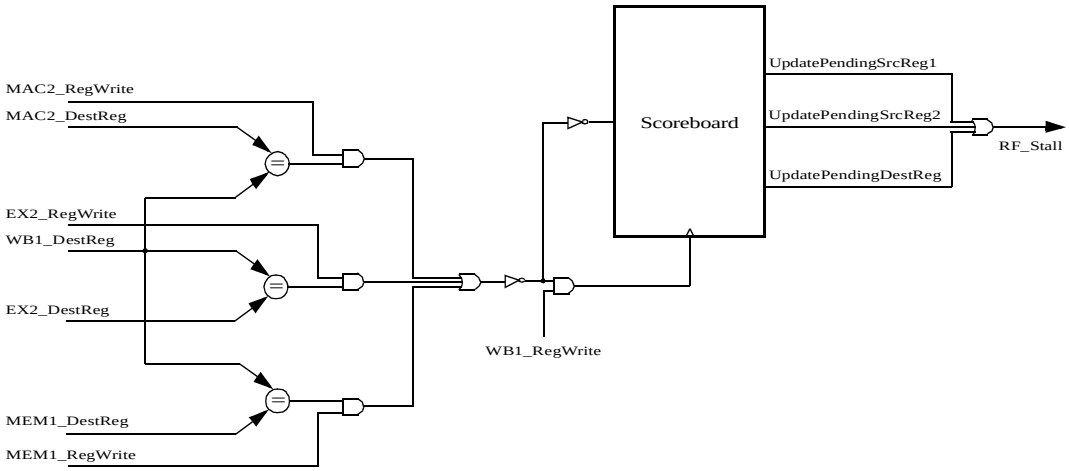

Figure 3. Control logic for scoreboard update from WB1.

is *true*, then an update is pending to the register corresponding to that entry. Figure 3 shows the control logic for updating the scoreboard from WB1. If an instruction completes in WB1, then it resets the entry in the scoreboard corresponding to the destination address of that instruction, provided no previous stage has a pending update to the same destination register. If an update is pending to any of the source registers of the instruction in RF, the pending update causes a read-after-write hazard. Similarly, a pending update to the destination register of the instruction in RF if an update is pending to the destination register, it indicates a write- after-write hazard. In any of the above cases, the pipeline is stalled up to, and including the RF pipestage.

Since the Scoreboard is used to simplify the control-logic for detecting data hazards, it is only part of the implementation, and not used in the specification processor. The scoreboard has to be initialized with pending updates from all stages in the pipeline after the RF stage. A new primary input *Initialize* is introduced in the implementation, and is used as the enable for all write ports to the scoreboard used for initialization. *Initialize* is triggered before the first implementation cycle. During regular operation, the scoreboard entries are set from EX1 and MAC1, and released from MAC3, MAC4, MAC5,WB2, and WB3 if write back occurs from that stage. The scoreboard has to be initialized from all stages after RF.

5.3 Imprecise Exceptions

Some data memory exceptions in the XScale are imprecise due to out of order execution. To formally verify the processor with imprecise exceptions, we modify the safety property by identifying the conditions that lead to imprecise exceptions, and use these conditions as a don't care set of the EUFM correctness formula.

We modeled three types of exceptions—instruction memory exceptions, ALU exceptions, and data memory exceptions. ALU and instruction memory exceptions are precise as they are detected before any younger instruction (that are more recent in program order), update the architectural state. Data memory exceptions can be both precise and imprecise. An exception is precise when there have been no out-of-order updates to the architectural state before the exception is detected; otherwise, the exception is considered imprecise. Younger instructions would not have been executed if a load raises an exception in a non-pipelined specification. However, in the XScale, those younger instructions may complete before the older load, if the younger instructions are completed in the main execution pipeline, or the first 2 stages of the MAC pipeline. Since these updates would not have occurred in the specification, where instructions are executed and completed sequentially, the architectural state that was updated out-of-order, now contains corrupted data. It is possible that the corrupted data has been used before the data memory exception handler is invoked. Therefore, all imprecise data memory exceptions are treated to be unrecoverable [16]. When a data memory exception is imprecise, a flag that is part of the fault status register (FSR) is set.

In order to formally verify imprecise exceptions, we propose a modified correctness criterion. The original correctness criterion checks the *safety property.* Since the implementation processor state is corrupted in case of an imprecise exception, the condition that an imprecise exception has occurred is used as a don't-care set, indicating when we do not care about the safety property.

When an imprecise exception occurs, the PC is updated to the address corresponding to the imprecise exception handler (IPH), and the FSR is cleared by the exception handler routine. We model the FSR as a latch,

Processor	Boolean Variables			CNF Variables	CNF Clauses	Formal Verification Time [sec]			
	e_{ij}	Control	Total			TLSim	EVC	BerkMin	Total
XS-7	450	102	552	4,347	39,459	0.11	1	3	4
XS-7-8	566	151	717	7,216	73,179	0.15	3	9	12
XS-7-8-SB	565	155	720	7,253	73,306	0.22	3	11	14
XS-7-8-SB-PC1	1,666	210	1,876	16,386	214,163	0.18	7	21	28
XS-7-8-SB-PC2	1,667	216	1,883	19,215	244,916	0.26	8	32	40
XS-7-8-SB-PC1-MC	2,082	280	2,362	23,369	396,711	0.34	17	92	109
XS-7-8-SB-PC2-MC	2,078	291	2,369	26,651	435,140	0.38	17	127	144
XS-7-8-SB-PC1-MC-IMP	2,386	272	2,658	36,266	573,252	0.46	23	162	185
XS-7-8-SB-PC2-MC-IMP	2,384	281	2,665	40,369	616,368	0.47	22	244	266

Table 1: Statistics for the Boolean correctness formulas, and the formal verification time. The SAT checker BerkMin62 was used for the experiments.

IsImpreciseException, that is initialized to *false*, and set when an imprecise exception occurs. The input to the latch is the OR of the previous state of the latch, and the condition that an imprecise exception is raised in the current cycle. The latch is updated only during the implementation side of the correctness criterion. All instructions are squashed after an imprecise exception is raised. The earliest that an imprecise exception can occur is in the one cycle of regular operation of the implementation. Even in that case, the new instruction, if fetched, will be squashed. Effectively, if an imprecise exception occurs, it is the last executed instruction during flushing, and the PC is updated to the address of the imprecise exception handler (IPH). We define the safety property for the implementation with imprecise exceptions in the framework for microprocessor correctness provided by Aagaard et al. [1][2]:

$$\forall q_i, q_i' \in Q_i \,.\, \forall q_s \in Q_s \,.\, \exists\, q_s' \in Q_s \,.$$
$$[N_i(q_i, q_i') \wedge \Re(q_i, q_s) \wedge \neg isImprecise(q_i') \Rightarrow \exists\, j \le w \,.\, N^j{}_s(q_s, q_s') \wedge \Re(q_i', q_s')] \wedge$$
$$[N_i(q_i, q_i') \wedge \Re(q_i, q_s) \wedge isImprecise(q_i') \Rightarrow equal\_PC\_IPH(q_i')]$$

where, Q_s and Q_i are the set of possible specification and implementation states, respectively. N_s and N_i are next state relations for the specification and implementation, respectively. $N^j{}_s(q_s, q_s')$ is a predicate that returns *true* if q_s' is reachable from q_s in exactly j steps of N_s, and returns *false* otherwise. $\Re(q_i, q_s)$ is a relation that returns *true* if the specification state, q_s, and the state obtained after flushing the implementation state q_i are equal. w is the issue width of the implementation. *isImprecise*() and *equal_PC_IPH*() are user-provided predicates that take an implementation state as argument; the former is *true* if its argument implementation state is imprecise, and returns *false* if the argument implementation state is precise; the latter is *true* if the PC of the argument implementation state is equal to the IPH, and is *false* otherwise.

6. Results

We started with the basic abstract processor model of the 7-stage main execution pipeline: XS-7, which can execute 5 basic abstract instruction types—register-register, register-immediate, conditional and unconditional branch, and return-from-exception. The model included features such as branch prediction, predicated execution, ALU, and instruction memory exceptions, as well as shifter, and ALU in different stages. The above model was extended to implement: 1) parallel memory pipeline with abstract load and store instructions, designated with "-8", where 8 represents the total latency of memory instructions; 2) register scoreboarding, marked by "-SB"; 3) logic to model the PC as part of the register file—marked with "-PC1" when the PC was abstracted with a term variable, and with "-PC2" when an UP was used to decide whether an instruction writes to the PC; 4) multiply-and-accumulate pipeline with abstract multiply-and-

accumulate instructions, which can take between 2 to 5 cycles, designated by "-MC"; 5) imprecise exceptions "-IMP", using the modified correctness criterion.

Controlled flushing [5]—a technique used to schedule the pipeline, so that the variability of instruction flow in the pipeline due to stalls is eliminated leading to a simpler correctness formula—was employed for all designs. Table 1 presents the results. "Control" Boolean variables are those that represent the initial state of bit-level signals in the pipeline latches, or were introduced when eliminating uninterpreted predicates. The e_{ij} Boolean variables encode g-equations (syntactically distinct g-term variables), with some of those variables added in order to enforce the property of transitivity of equality. We used both SAT-checkers Chaff and BerkMin, and found that BerkMin was faster by a factor of 2 for all benchmarks. As seen in table the time taken by BerkMin ranges from 3 seconds for XS-7 up to 244 seconds for the most complex model with all features, including imprecise exceptions, XS-7-8-SB-PC2-MC-IMP. The memory requirement varies between 29 MB and 141 MB. TLSim requires less than 1 MB of memory, and less than a second for all steps. EVC takes between 1 second and 22 seconds of CPU time, and uses between 29 MB and 148 MB of memory. The total CPU time required by the tool flow is between 4 seconds and 344 seconds.

Constraints imposed on the initial state of the pipeline latches need to be checked for invariance, i.e., they should be satisfied after the one cycle of regular operation of the implementation. We imposed constraints on all benchmarks except XS-7. To check for invariance, Chaff takes less than 0.03 seconds, the time taken by EVC ranges between 0.13 seconds and 0.19 seconds, and TLSim takes less than 0.1 seconds for all benchmarks. The memory requirement for TLSim, EVC, and Chaff is less than 1 MB. A maximum of 540 CNF variables, 2025 CNF clauses, 77 e_{ij} variables, and 177 total Boolean variables were required.

6.1 Bug Report

We find the following three bugs to be the most interesting from those we made:

Bug 1. Adding the parallel memory pipeline required taking into consideration corner cases for overlapping instructions. For example, an instruction might perform both a store and a load instruction. The bug was cleared by defining non-overlapping instruction (checks that an instruction does not perform more than one type of function), and exclusivity constraints (ensures that pipeline stages that are parallel do not both contain valid instructions). For a more detailed explanation of constraints refer to Section 5.2.

Bug 2. The scoreboard is not part of the specification, and therefore its initial state is abstract, and does not reflect the current state of the processor. This was cleared by initializing the scoreboard to reflect the instruction flow in the pipeline with updates from all pipeline stages after RF. Other entries in the scoreboard that have not been updated are set to *false*.

Bug 3. A data memory exception should be suppressed if the instruction causing the data memory exception was previously squashed. When an instruction becomes invalid due to a false value of the qualifying predicate (determined in EX1) was not used to squash the data memory exception. The value of the predicate condition of the instruction was propagated to WB2, where it was used to suppress the data memory exception in case the predicate condition was *false*. A similar bug is described in [39], where the Current Frame Marker register (CFM) is updated speculatively in the Register-Files-Access stage by the instruction packet in that stage. However, if the instruction packet is squashed in a later pipeline stage, the original CFM value should be restored.

7. Related Work

Previous work on verifying processors has been done primarily on pipelines similar to the five stage pipelined DLX. Some of these efforts include features such as precise exceptions, and external interrupts, but no work has been done previously in processors with scoreboarding or imprecise exceptions.

Patankar et al. [27] verified a processor that is a hybrid between ARM7, and StrongARM, with features such as predicated execution, and multi-cycle instructions. They used Symbolic Trajectory Evaluation (STE) to verify that the implementation circuit fulfills the ISA that is defined as a set of abstract assertions. However their method required a lot of manual intervention to formally verify only one instruction.

This paper is a case study that is based on the work by Velev and Bryant [37], who developed EVC [41], and used it to formally verify 7 different configurations of dual-issue superscalar DLX processors. They extended this work further in [38] to model features such as functional units, and memories with arbitrary multicycle latencies, branch prediction, and precise exceptions in the context of single-issue and dual-issue DLX models. Lahiri et al. [21] used the same tool flow, and abstraction techniques at Motorola to check the correctness of the M•CORE processor with instruction prefetching, multicycle functional units of fixed latency, precise exceptions, and branch prediction.

Mishra and Dutt [25] used SVC [31] to check the correct flow of instructions in a pipelined DLX with multicycle functional units and exceptions. Their method does not guarantee completeness of the correctness proof, and is only applicable to processors with in-order execution, and in-order completion.

Sawada and Hunt [29] used the ACL2 theorem prover [19] to verify an out-of-order superscalar microprocessor with precise exceptions, external interrupts, and speculative execution by comparing the state transitions of pipelined, and non-pipelined machines in the presence of external interrupts. Tahar and Kumar [35] proved the correctness of a pipelined DLX with exceptions using the theorem prover HOL [12]. They wrote scripts to generate the verification conditions for the absence of hazards, but had to manually define conditions to verify the data forwarding and exceptions.

Huggins and Van Campenhout [15] used Abstract State Machines to prove the correctness of refinement steps that transform a non-pipelined ARM processor into a pipelined implementation. However, they had to manually define a large number of lemmas to prove the absence of hazards after each refinement step. Hosabettu et al. [13][15] spent 1 month of manual work to formally verify a plain DLX by using manually defined completion functions to compute the abstraction function.

Previous work in handling out-of-order execution with respect to loads has been done by Kroft [20], where he uses a Miss Status Handler Register (MSHR) to keep track of the outstanding memory requests. If an instruction attempts to use the destination register of a load before the load is completed, that instruction is blocked in the decode stage.

We use conditions to modify the correctness criterion to verify the implementation processor with imprecise exceptions. Such a modification is necessary as imprecise exceptions are not a property of the specification. Srivas and Miller [32][33] have defined similar preconditions to prove the correctness of the AAMP5 processor with the PVS theorem prover. These preconditions were required to handle the consequences of hiding the stack cache from the macro level.

Su et al. [34] use syntactic manipulation of formulas in first-order logic to automatically generate invariants. Isles et al. [18] extended their earlier work on the Integer Combinational Sequential concurrency model in order to automatically compute invariants for the reachable states of a pipelined implementation. Tiwari et al. in [36] have described a process for inductive invariant generation, and strengthening based on computing under-approximations, and over-approximations of the reachable state set. These methods derive a large number of invariant constraints that are too restrictive and detailed. In contrast we define constraints that are sufficient for the formal verification to go though, with each constraint representing conditions that are either local to one stage (non-overlapping constraints) or involve control bits from parallel stages (exclusivity constraints).

Marinescu and Rinard [22][23] present an algorithm for automatic pipelining of sequential circuits, and allow speculative update of the architectural state. The algorithm ensures that the old values of speculatively updated state elements are carried along the pipeline, and are restored in case of wrong speculation. If correctly implemented, this method could have prevented bug 3 (see Section 6.1), and a similar bug in [39].

8. Conclusion

We formally verified a model of the Intel XScale superpipelined RISC processor where the main execution, enhanced memory, and MAC pipelines, have different latencies. Features such as the Register Scoreboarding (that was formally verified for the first time), precise and imprecise exceptions, branch prediction, and predicated instruction execution were implemented. The most complex model was formally verified in 344 seconds. We proposed a method to formally verify imprecise exceptions.

Acknowledgments

We thank M. Morrow for answering questions about the Intel Xscale architecture.

References

[1] M.D. Aagaard, B. Cook, N.A. Day, R.B. Jones, “A Framework for Superscalar Microprocessor Correctness Statements,” Software Tools for Technology Transfer, Vol. 4, No. 1 (2002).

[2] M.D. Aagaard, N.A. Day, M. Lou. “Relating Multi-step and Single-step Microprocessor Correctness Statements,” *Formal Methods in Computer-Aided Design (FMCAD’02)*, M. Aagaard and J.W. O’Leary, *eds*., LNCS 2517, Springer-Verlag, November 2002,.

[3] *allnetDevices*, http://www.allnetdevices.com/wireless/news/2002/06/24/hps_ipaq.html.

[4] *ARM Technical Reference Manuals & Data Sheets*, http://www.arm.com.

[5] J.R. Burch, “Techniques for Verifying Superscalar Microprocessors,” *33rd Design Automation Conference (DAC’96)*, June 1996, pp. 552–557.

[6] J.R. Burch, and D.L. Dill, “Automated Verification of Pipelined Microprocessor Control,” *Computer-Aided Verification (CAV‘94)*, D.L. Dill, *ed*., LNCS 818, Springer-Verlag, June 1994, pp. 68–80.

[7] L.T. Clark, E.J. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M.Morrow, K.E. Velarde, and M.A. Yarch, “An Embedded 32-bit Microprocessor Core for Low-Power and High-Performance Applications,” IEEE Journal of Solid-State Circuits, Vol. 36, No. 11 (November 2001), pp. 1599–1608.

[8] *Dell Axim X5*, http://www.dell.com/us/en/gen/topics/segtopic_axim.htm.

[9] D. Dobberpuhl, “The design of a high performance low power microprocessor,” *International Symposium on Low Power Electronics and Design*, 1996,.

[10] *EMBEDDED MIPS PROCESSOR CORE*, http://www.mips.com/products/s2p3.html.

[11] E. Goldberg, and Y. Novikov, “BerkMin: A Fast and Robust SAT-Solver,” *Design Automation and Test in Europe (DATE’02)*, 2002.

[12] M.J.C. Gordon, and T.F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher*

OrderLogic, Cambridge University Press, 1993.

[13] R.M. Hosabettu, "Systematic Verification of Pipelined Microprocessors," Ph.D. Thesis, Department of Computer Science, University of Utah, August 2000.

[14] R.M. Hosabettu, M. Srivas, and G. Gopalakrishnan, "Decomposing the Proof of Correctness of Pipelined Microprocessors," *Tenth International Conference on Computer Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds*., LNCS 1427, Springer-Verlag, June 1998.

[15] J.K. Huggins, and D.V. Campenhout, "Specification and Verification of Pipelining in the ARM2 RISC Microprocessor," ACM Transactions on Design Automation of Electronic Systems, Vol. 3, No. 4 (October 1998), pp. 563–580.

[16] *Intel XScale™ Technology*, http://www.intel.com/design/intelxscale/.

[17] *INTEGRITY RTOS*, http://www.ghs.com/download/whitepapers/integrity-rtos-xscale.pdf.

[18] A.J. Isles, R. Hojati, and R.K. Brayton, "Computing Reachable Control States of Systems Modeled with Uninterpreted Functions and Infinite Memory," *Computer-Aided Verification (CAV'98)*, A.J. Hu, and M.Y. Vardi, *eds.*, LNCS 1427, Springer-Verlag, June 1998.

[19] M. Kaufmann, and J.S. Moore, "ACL2: An Industrial Strength version of Nqthm," *Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS'96)*, June 1996, pp. 23–34.

[20] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization," *Proceedings of the International Symposium on Computer Architecture*, May 1981.

[21] S. Lahiri, C. Pixley, and K. Albin, "Experience with Term Level Modeling and Verification of the M•CORE" Microprocessor Core," *6th Annual IEEE International Workshop on High Level Design, Validation and Test (HLDVT'01)*, November 2001.

[22] M. Marinescu, and M. Rinard, "High-level Automatic Pipelining for Sequential Circuits," *International Symposium on System Synthesis (ISSS'01)*, September-October 2001, pp. 215–220.

[23] M. Marinescu, "Synthesis of Synchronous Pipelined Circuits from High-Level Modular Specifications," Ph.D. Thesis, University of California at Santa Barbara, April 2002.

[24] *M•CORE*: *microRISC Engine Programmer's Manual*, http://www.motorola.com/SPS/MCORE.

[25] P. Mishra, and N. Dutt, "Modeling and Verification of Pipelined Embedded Processors in the Presence of Hazards and Exceptions," *IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems (DIPES'02)*, August 2002.

[26] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Design Automation Conference (DAC'01)*, 2001, pp. 530–535.

[27] V.A. Patankar, A. Jain, and R.E. Bryant, "Formal verification of an ARM processor," *Twelfth International Conference On VLSI Design*, 1999, pp. 282–287.

[28] *PC WORLD*, http://www.pcworld.com/news/article0,aid,83923,00.asp.

[29] J. Sawada, and W.A. Hunt, "Processor Verification with Precise Exceptions and Speculative Execution," *Computer-Aided Verification (CAV'98)*, A.J. Hu and M.Y. Vardi, *eds*., LNCS 1427, Springer-Verlag, June 1998.

[30] H. Sharangpani, and K. Arora, "Itanium Processor Microarchitecture," *IEEE Micro*, September-October 2000, pp. 24–43.

[31] *Stanford Validity Checker (SVC)*, http://sprout.Stanford.EDU/SVC.

[32] M.K. Srivas, and S.P. Miller, "Formal Verification of an Avionics Microprocessor," Technical Report CSL-95-04, SRI International Computer Science Laboratory, June 1995.

[33] M.K. Srivas, and S.P. Miller, "Applying Formal verification to a Commercial Microprocessor," *International Conference on Computer Hardware Description Languages (IFIP'95)*, August 1995.

[34] J.X. Su, D.L. Dill, and C. Barrett, "Automatic Generation of Invariants in Processor Verification," *Formal Methods in Computer-Aided Design (FMCAD'96)*, M.K. Srivas and A.J. Camilleri, *eds.*, LNCS 1166, Springer-Verlag, November 1996.

[35] S. Tahar and R. Kumar, "A Practical Methodology for the Formal Verification of RISC Processors," Formal-Methods in Systems Design, Vol. 13, No. 2 (September 1998), Kluwer Academic Publishers, pp. 159–225.

[36] A. Tiwari, H. Rueb, H. Saidi, and N. Shankar, "A Technique for Invariant Generation," *Tools and Algorithms for Construction and Analysis of Systems (TACAS'01)*, April 2001, vol. 2031, pp. 113–127.

[37] M.N. Velev, and R.E. Bryant, "Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic," *Correct Hardware Design and Verification Methods (CHARME'99)*, L. Pierre and T. Kropf, *eds*., LNCS 1703, Springer-Verlag, September 1999.

[38] M.N. Velev, and R.E. Bryant, "Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction," *37th Design Automation Conference (DAC'00)*, June 2000.

[39] M.N. Velev, "Formal Verification of VLIW Microprocessors with Speculative Execution," *Computer-Aided Verification (CAV'00)*, E.A. Emerson, and A.P. Sistla, *eds*., LNCS 1855, Springer-Verlag, July 2000.

[40] M.N. Velev, and R.E. Bryant, " Effective Use of Bool ean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," *38th Design Automation Conference (DAC'01)*, June 2001.

[41] M.N. Velev, and R.E. Bryant, "EVC: A Validity Checker for the Logic of Equality with Uninterpreted Functions and Memories, Exploiting Positive Equality and Conservative Transformations," *Computer-Aided Verification (CAV'01)*, G. Berry, H. Comon, and A. Finkel, *eds*., LNCS 2102, Springer-Verlag, July 2001.

[42] M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors," Journal of Symbolic Computation (JSC), Vol. 35, No. 2, (February 2003), pp. 73–106.

[43] M.N. Velev, "Integrating Formal Verification into an Advanced Computer Architecture Course," *ASEE Annual Conference & Exposition*, June 2003.