International Conference on Computer Modeling and Simulation

Formal verification of Out-of-order Processor

Yanyan Gao Xi Li
Department of Computer Science, University of Science and Technology of China, China
yygao@mail.ustc.edu.cn *llxx@ustc.edu.cn*

Abstract

Out-of-order execution is a fundamental technique to achieve instruction-level parallelization in processor designs. The verification of out-of-order processor is a main challenge in processor design. This paper presents a formal method to model and check the correctness of out-of-order design at instruction level. This method is based on model checking, a widely used formal verification technique. The rules to generate properties an out-of-order design should satisfy are also provided. The abstracted model can be verified with NuSMV, a popular model checking tool. If the properties cannot be satisfied, a counterexample is created to help correct the design.

Keyword：Out-of-order，verification，processor

1. Introduction

Parallel execution is the key implementation technique used to enhance the throughput of processor and speed up the execution. A major effect of parallel execution is to change the relative timing of instructions by overlapping their execution. Out-of-order processor is a universal technique to implement the parallel execution with additional units such as Reorder Buffer (ROB), Reservation Station (RS), and so on. However, this overlapping introduces data hazards, which occur when the execution changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an in-order processor. And this also makes the verification of out-of-order processor a main challenge in processor design.

There are some previous approaches that model and verify an out-of-order processor. In [1], T. Arons and A. Pnueli present a refinement-based proof method for the verification of modern processor architectures. They introduce a *predicted value* field to directly compare system TOMASULO to system SEQ. The proof uses the PVS [2] theorem prover and the model is limited to non-branching programs in which no loads, stores or exceptions occur. In [3], Tomasulo's algorithm is verified using the SMV verifier [4] and achieved automation. The proof, however, is dependent on the configuration and arithmetic operators. And any modification requires that the new system be verified afresh. A more complex model of out-of-order execution is verified in [5] using the Stanford Validity Checker [6]. The proof is based on showing that the implementation machine is the refinement of an intermediate abstraction. Incremental flushing is used to show that the intermediate abstraction and the specification machine are functionally equivalent. Compared with these presented works, our verification method is able to verify the entire system without intermediate models. And the verification process is implemented with a model checker, NuSMV [7], which can automatically verify whether the design satisfies the properties.

The main challenge in model checking method is to efficiently abstract the model of target design and the properties the design should satisfy. Therefore, our method consists of two parts: 1) abstraction of the model for the verification of dynamic behavior; 2) generation of the property an out-of-order processor should obey from customer-built requirements. These works will be presented in section 2 and section 3 respectively.

2. Abstraction model from an out-of-order processor

2.1 Overview of Tomasulo algorithm

In [8], Tomasulo algorithm and extended are used to handle parallel execution. The basic structure of a Tomasulo-based MIPS processor consists of the floating-point unit, the load/store unit, and reservation station. Each reservation station holds an instruction that has been issued awaiting execution at a functional unit. Either the operand values for that instruction if they have already been computed, or else the names of the functional units that will provide the operand values, is recorded as operand of the corresponding instruction. In Tomasulo algorithm, reservation stations are used to solve data hazards and the instructions are committed out-of-orderly. In comparison, the extended Tomasulo algorithm consists of four steps - *Issue, Execute, Write result* and *Commit* to implement in-order commit by adding a reorder buffer (ROB).

The property of in-order commit makes the extended Tomasulo algorithm supports accurate exception. In extended Tomasulo algorithm, the instructions are in-order issued, but the execution sequence may be changed in step *Execute* to reduce the delay caused by data hazards. The instructions without data dependency on former instructions will be executed first. The changed sequence leads to out-of-order execution and commit. Multiple functional units executing simultaneously will also lead to out-of-order commit. Therefore, in step *Write result* and *Commit*, ROB is used to solve this problem. The results are written in the ROB and reservation stations through Common Data Bus (CDB). Only if the former instructions have been committed, the latter instructions can be committed. These adjustments make the in-order commit possible.

Figure 1 is a classical out-of-order processor based on extended Tomasulo algorithm, where *Load/Store* unit is used to access the memory unit *Mem*, the units *ADD* is used to implement algorithm instructions and unit *M1, M2, M3* and *M4* are combined to implement multiplication instructions. There are four stages in processor's execution: *issue, execute, write result,* and *commit*. The issued instructions are dispatched to reservation stations and reorder buffer to await execution and record computation result, respectively. The results of execution units are passed to reorder buffer by CDB. The results of the instructions which are recorded in reorder buffer are committed in-orderly.

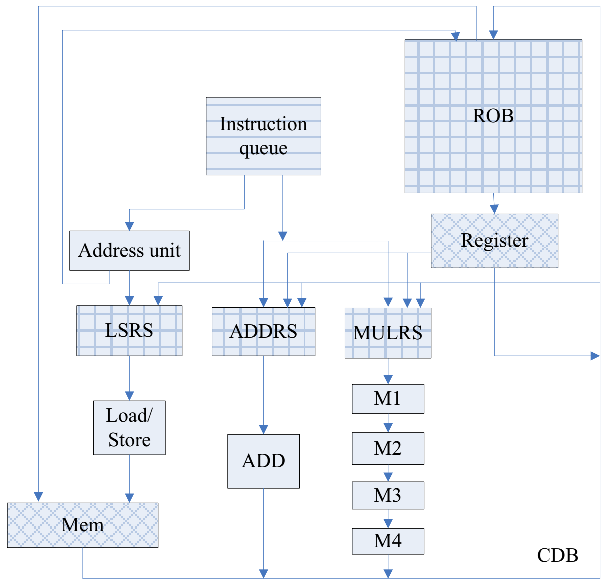

Figure 1. An example of an out-of-order architecture

If two designs are equal, the results of any programs input to them should be equal too. Therefore, if an out-of order execution design is equal with the in-order execution one, the results of them is equal.

Theorem 1. An out-of-order design is correct iff to any instruction sequence the result of the out-of-order execution is equal to the result of in-order execution of this sequence.

2.2 The data structure of the model

In in-order execution processors, the instructions are executed sequentially and the results are generated and committed to the storage units in sequence. In out-of-order execution processors, the result of the latter instruction maybe generated earlier than the former. These results will be committed to the storage immediately in Tomasulo algorithm. However, the extended Tomasulo algorithm will reorder the program sequence of the instructions in ROB to assure the results of instructions are committed in-orderly. According to theorem 1, to check the correctness of an out-of order design, we need to check the equivalency of the results between the out-of-order design and an in-order machine with the same input program. To efficiently compare these results, three parts are modified or added to the model while abstracting the model of the design for verification (see Figure 2).

To get the result of in-order execution and obviate the need of intermediate model, the field *ept_val* is added to several modules while abstracting the implementation model. This field is used to record the result of computation for in-order execution. The instructions are issued in-orderly. If an instruction is computed when it is issued and the result is sent to the corresponding storage unit at once, the execution is similar with the principle of in-order machine. In Figure 2 (b), each storage unit is added with an *ept_val* field to simulate the in-order execution. And in *Memory unit* module, a special field *tag* is added to record which instruction executes write operation to the *ept_val* field of the item.

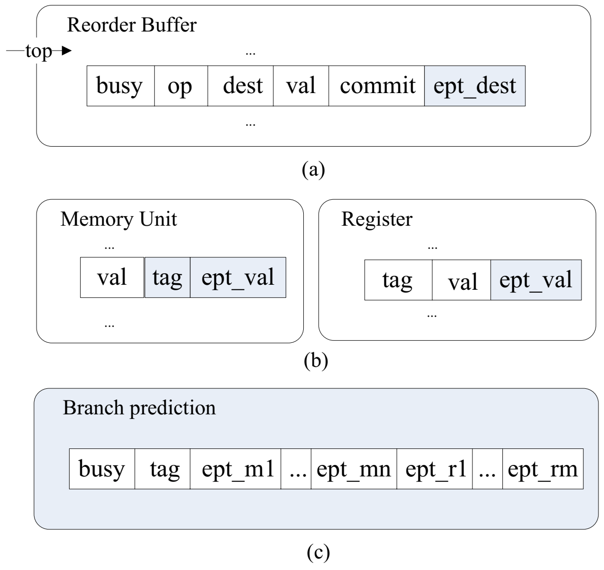

Figure 2. Data structure for out-of-order execution

To efficiently simulate the implementation of in-order execution, address computation is also an important part. Therefore, in Figure 2 (a) a field *ept_dest* is added to each item of the reorder buffer to record the address that the result should be stored. While checking the consistency of output of out-of order execution and in-order execution, the destination address in *dest* field and the expected destination address in *ept_dest* field are compared to check the correctness of the computation of address. The computation result (recorded in *val* field) and the expected value in the storage unit corresponding with the destination address are compared to verify the correctness of the computation result afterwards. If these contents are equal, the out-of-order execution design is equal with the corresponding in-order execution machine; otherwise there are some faults in the design.

To a design based on Tomasulo algorithm, this method is also valid. The difference is that the field *ept_dest* is added to the reservation station instead of reorder buffer.

Our model also supports branch prediction. If prediction is not taken, both the out-of-order processor and in-order execution should be reverted to the state before the branch is executed. To solve this problem, a *Branch prediction* module (see Figure 2 (c)), in which an item *FPB* is a backup storage of the model to record the last state before the execution of a branch instruction, is added. When an issued branch is falsely predicated and the item is empty, the tag of the instruction and the backup information of the storage units are added to the item. When the branch instruction in the item is to be committed, the backup information in it will be sent to the *Memory unit* module and *Register* module to renew the values in the field *ept_val* to the state before the branch happened. The information in the item will be flushed, too. Because the instructions following the falsely predicated branch instruction will be flushed while the branch instruction is committed, one item in the module is enough.

2.3 Abstraction of model for verification

To efficiently verify the design, there are two kinds of models to abstract. The first kind is the model of the global design, in which many details are hided to reduce the scale of the model. This kind of model is used to check the validity of the interaction between the units. The second kind is local models, in which concrete designs of each individual unit are described to verify detailed specifications. In this situation, the other parts of the design are treated as outer environment, and the interface parameters of the model are set as stochastic values in the possible range. The abstraction of local models helps to make the verification focuses on local designs.

To the design in Figure 2, firstly the global design is abstracted as described in section 2.2. Since NuSMV is based on FSM, which easily leads to state space explosion problem, in this model each unit is simplified to reduce the scale of the NuSMV program. Then the rule 1, 2 and 3 (described in section 3) are implemented to check whether the execution result of the implementation model is consistent with sequential execution.

Afterwards, local models are abstracted from individual units. Each model is more refined compared with the corresponding description in the global model. At this step, the properties concern concrete details are presented to help to locate the mistakes and verify the contents which maybe ignored in the global model. The verification is carried out according to the description in rule 4 (see section 3).

3. Property for verification

The main challenge in generating properties for an out-of-order processor is that the properties should cover all the situations in parallel execution. Depending on the models constructed in section 2, there are two kinds of properties should be verified. To the global model of the processor, it requires that the result of the out-of-order execution is equal to the result coming from the in-order execution for any input instruction sequence. Rules to construct properties cover the global model are introduced as follows.

Rule 1. To a global model which is abstracted following section 2.2 and 2.3, the design is correct iff when an instruction is to be committed,

(1) The destination address in *dest* field is equal to the expected address in *ept_dest* field;

$$\begin{aligned}&(ROBhead commit = 1) \rightarrow \\ &\quad (ROBhead dest = ROBhead ept\_dest)\end{aligned} \tag{1}$$

(2) The destination result in *val* field is equal to the expected result in *ept_val* field in the storage unit which is located by the address in *ept_dest* field;

$$\begin{aligned}&(ROBhead.commit = 1)\,\& \\ &\quad (ROBhead.op \in MemWrClass) \rightarrow \\ &\quad (Mem[ROBhead.ept\_dest].tag != ROBhead.tag)\,| \\ &\quad (Mem[ROBhead.ept\_dest].ept\_val = ROBhead.val)\end{aligned} \tag{2}$$

or

$$\begin{aligned}&(ROBhead.commit = 1)\ \&\\ &(ROBhead.op \in RegWrClass) \rightarrow \qquad (3)\\ &(Reg[ROBhead.ept\_dest].tag != ROBhead.tag)\ |\\ &(Reg[ROBhead.ept\_dest].ept\_val = ROBhead.val)\end{aligned}$$

Proof:

For out-of-order configuration is used to solve data hazards, if the result of computation is correct the addresses and the results should equal to the addresses and results coming from the in-order execution respectively. Otherwise, when the result is committed, the corresponding storage unit will be set falsely.

Due to the *ept_val* field in the storage units are modified by the latest instructions that are issued to the reservation and ROB. Only when there is no instruction which is later in the program sequence than the committing instruction has written the same unit, the value in the field *ept_val* of the unit presents the expected value of the committing instruction. ■

To extended Tomasulo algorithm, since each issued instruction is stored into one of the reservation station and reorder buffer synchronously, rule 2 is needed to check the consistency of these two units.

Rule 2. The instructions in the reservation stations are consistent with the instructions in the reorder buffer if

(1) The number of instructions in the reservation stations is equal to the number of instructions in the reorder buffer;

$$ROB.num = \sum_{i=1}^{sn} RS_i.num \qquad (4)$$

where *sn* is the number of the reservation stations.

(2) Each instruction in the reservation stations can be found in the reorder buffer;

$$\forall i (i \in \bigcup_{j=1}^{sn} RS_j) \rightarrow \exists k (ROB[k] = i) \qquad (5)$$

where *sn* is the number of the reservation stations.

(3) Each instruction in the reorder buffer is stored in a unit of the reservation stations.

$$\forall i (i \in ROB) \rightarrow \exists j (i \in RS_j), \quad j \in [1, sn] \qquad (6)$$

where *sn* is the number of the reservation stations.

Rule 3. To an out-of-order execution design based on extended Tomasulo algorithm, each branch instruction in the *ROB* should satisfy:

(1) If the *val* field of the instruction is *false*, i.e. the prediction of the branch is false, the instruction should also be stored in the *branch prediction* module; otherwise, the instruction can not be found in the module.

$$\begin{aligned}&\forall x (ROB[x].op \in BranchInst)\ \&\\ &\quad (ROB[x].val = false) \rightarrow \qquad (7)\\ &\qquad \exists y (Br[y].tag = x)\end{aligned}$$

(2) Once a falsely predicated branch instruction is committed, the *ept_val* field of the storage units will be updated by the corresponding backup information recorded in the *Branch prediction* module; and the item *FPB* in the *Branch prediction* module must be flushed.

$$\begin{aligned}&(ROBhead.op \in BranchInst)\\ &\ \&\ (ROBhead.commit = 1)\\ &\ \&\ (ROBhead.val = false) \qquad (8)\\ &\ \&\ (FPB[d_1] = v_1)\ \&\ ...\ \&\ (FPB[d_{sm}] = v_{sm}) \rightarrow\\ &\quad X((SU[d_1] = v_1)\ \&\ ...\ \&\ (SU[d_{sm}] = v_{sm}))\end{aligned}$$

and

$$\begin{aligned}&(ROBhead.op \in BranchInst)\\ &\ \&\ (ROBhead.commit = 1) \qquad (9)\\ &\ \&\ (ROBhead.val = false)\\ &\qquad \rightarrow X(FPB[busy] = 0)\end{aligned}$$

where *sm* is the number of the items in the storage units, *SU* presents the storage units, d_i presents the *i*th item in the storage units, v_i presents the value in the *i*th item and *X* presents the next state.

To implement an all-sided verification of the design, so as to locate possible mistakes, concrete properties used to check the correctness of local designs are introduced as follows.

Rule 4. Each unit in the model such as issue unit, reservation stations, etc. satisfies the corresponding specifications, i.e., the customized requirements.

Rule 1 to rule 3 are used to check the correctness of the design of the global system. Rule 4 is used to verify the detail configuration of each unit.

4. The Correctness of the model

The additional information in the model is used to simulate the execution of the in-order machine. In the model, the instruction is calculated before it is issued, and the result and the destination address are recorded in the field *ept_dest* and *ept_val*. The *ept_dest* is either passed to the *ROB* with the instruction, if the design is based on extended Tomasulo algorithm, or passed to the corresponding reservation station, if the design is based on Tomasulo algorithm. The *ept_val* is stored in the field *ept_val* of the corresponding storage unit as the results of the in-order execution after the instruction is issued. The source operands are also coming from the field *ept_val* in the storage unit. Therefore, the execution of the instructions strictly depends on the program sequence and there is no data hazard in the execution, i.e., if the program sequence is i_1, i_2, …, i_{n-1}, i_n, …, the instruction i_n is computed only after the instruction i_{n-1} have been completed. To simulate the in-order machine, additional computing and control logic are added. To assure these additional designs are correct, the following theorem should be obeyed.

Rule 5. The in-order part of the model is correct iff:

(1) If an instruction is in the *issue* stage and there is no *flush* operation to the stage, in the next clock cycle the instruction will be stalled or issued to the tail of ROB;

$$\forall i(Issue\_inst = i)\,\&\,(!\,flushed) \rightarrow X(Issue.state = stall \mid ROB.tail = i) \quad (10)$$

(2) If an instruction is to be issued, the *ept_dest* and *ept_val* is computed correctly;

(3) If an instruction is issued, the value in the field *ept_val* must be stored in the expected destination address *ept_dest*.

$$\forall i((Issue\_inst = i)\,\&\,(ept\_dest \in Mem)\,\&\,(ept\_val = x)) \rightarrow X(Mem[ept\_dest] = x) \quad (11)$$

or

$$\forall i((Issue\_inst = i)\,\&\,(ept\_dest \in \mathrm{Reg})\,\&\,(ept\_val = x)) \rightarrow X(\mathrm{Reg}[ept\_dest] = x) \quad (12)$$

In out-of-order execution processor which supports branch prediction. If the prediction is not taken, the machine must rollback to the state just before the branch instruction is executed and the instructions after the branch instruction in the program sequence must be flushed at the same time. This situation is also supported by the in-order machine.

Rule 6. If an issued branch instruction is falsely predicated and the item *FPB* in the *Branch prediction* module is empty, it must be sent to the *FPB* in the next state.

$$\forall i((Issue\_inst = i)\,\&\,(i \in BranchInst)\,\&\,(ept\_val = false)\,\&\,(FPB.busy = 0)\,\&\,(SU[d_1] = v_1)\,\&\ldots\&\,(SU[d_{sm}] = v_{sm})) \rightarrow X((FPB[tag] = i\_tag)\,\&\,(FPB[d_1] = v_1)\,\&\ldots\&\,(FPB[d_{sm}] = v_{sm})) \quad (13)$$

Rule 7. If a branch instruction to be committed is falsely predicated:

(1) It must be in the *FPB* of the *Branch Unit*;

$$\forall i((Commit[head] = i)\,\&\,(i \in BranchInst)\,\&\,(val = false)) \rightarrow (FPB[tag] = i\_tag) \quad (14)$$

(2) After the instruction is committed the *FPB* must be flushed.

$$\forall i((Commit[head] = i)\,\&\,(i \in BranchInst)\,\&\,(val = false)) \rightarrow X(FPB[busy] = 0) \quad (15)$$

Rule 5, 6 and 7 are used to assure the correctness of the description of in-order execution. Rule 5 describes the conditions an in-order description should obey. Rule 6 and 7 are conditions that an out-of-order design, which supports branch instructions, should obey.

5. Conclusions

We have introduced a method to model and verify out-of-order processor design. The method includes two steps: the verification for the global system and the verification for the local designs. In the first step, a global model is abstracted to check the consistency of the out-of-order design with the sequential execution. Some additional information is added to simulate the execution of in-order machine. The generation of properties that an out-of-order design should satisfy is also presented. This step helps to check the correct of out-of-order design automatically. To implement an all-sided verification, local models are detailedly abstracted and verified in step 2. This method proves correctness for arbitrary configurations. In addition, an out-of-order processor has been studied to illustrate the applicability of our approach to practical systems.

The future work is to apply this method to verify more complex architecture, such as multi-core processor.

Acknowledgments

This work was supported in part by Anhui Natural Science Foundation (No. 070412030) and The Research Fund for the Doctoral Program of Higher Education (No. 20050358040).

Reference

[1] T. Arons and A. Pnueli. *Verifying Tomasulo's algorithm by refinement*. in *VLSI Design, 1999. Proceedings. Twelfth International Conference*. 1999.

[2] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. *A tutorial on using PVS for hardware verification*. in *Proceedings of the Second Conference on Theorem Provers in Circuit Design*. 1994: FZI Publication, Universit¨at Karlsruhe.

[3] K.L. Mcmillan. *Verification of an implementation of Tomasulo's algorithm by compositional model checking*. in *CAV'98*. 1998.

[4] K.L. Mcmillan. Symbolic Model Checking. Kluwer. 1993.

[5] J.U. Skakkebaek, R.B. Jones, and D.L. Dill. *Formal verification of out-of-order execution using incremental flushing*. in *CAV'98*. 1998.

[6] C. Barrett, D. L. Dill, and J. Levitt. *Validity checking for combinations of theories with equality*. in *FMCAD '96*. 1996. Stanford, CA, USA: Springer-Verlag.

[7] *NuSMV: a new symbolic model checker*. [cited 2007 July]; Available from: http://nusmv.irst.itc.it/

[8] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*. Fourth ed. 2007: Elsevier.