A Rigorous Correctness Proof of a Tomasulo Scheduler Supporting Precise Interrupts

Daniel Kroening*, Silvia M. Mueller† and Wolfgang J. Paul

Dept. 14: Computer Science, University of Saarland,
Post Box 151150, D-66041 Saarbruecken, Germany
email: {kroening,smueller,wjp}@cs.uni-sb.de

ABSTRACT

The Tomasulo Algorithm is the classical scheduler supporting out-of-order execution; it is widely used in current high performance micro processors. In this paper, we combine the Tomasulo Scheduler with a reorder buffer which implements precise interrupts, and we give a mathematical correctness proof for this enhanced scheduling algorithm. We show that data consistency is maintained, and that the scheduling is deadlock free and fair.

Keywords: Out-of-Order Execution, Tomasulo Hardware Scheduler, Precise Interrupts, Reorder Buffer, Correctness, Fairness

1. INTRODUCTION

The performance of in-order, pipelined designs drops as soon as data dependencies occur or instructions with high latency such as floating point or memory instructions are used. Therefore, today's high-performance microprocessors such as Intel's Pentiums [1, 2], AMD's K7 or IBM's PowerPCs [3, 4] employ out-of-order execution to raise performance.

In contrast to in-order execution, out-of-order execution permits that instructions which block the execution stream can be overtaken by later instructions. This results in better utilization of the function units and in a higher performance compared to the pipelined in-order designs [5].

In processors with out-of-order execution, the resources and the instruction flow are governed dynamically by hardware schedulers. Most of these schedulers are based on the Tomasulo algorithm which was introduced in the IBM 360/91 [6].

It its original form, the Tomasulo scheduler does not support precise interrupts, which is another key concept of modern CPUs [7]. Precise interrupts are required by many techniques, like IEEE floating point arithmetic, virtual memory, or fast I/O [8]. There exist several mechanisms for implementing precise interrupts such as the reorder buffer, the future file[7], and the instruction window [9]. The reorder buffer is most commonly used. In this paper we therefore analyze a variant of the Tomasulo scheduler with a reorder buffer.

*supported by the DFG graduate program 'Effizienz und Komplexität von Algorithmen und Rechenanlagen'

†partially supported by the German Science Foundation DFG

Related Work

Recent papers show the correctness of the original Tomasulo scheduling algorithm in different models using automated model checkers. Henzinger et al. [10] use reactive models to specify an intuitive definition of the instruction set architecture and the scheduling hardware.

Damm and Pnueli [11] prove the original Tomasulo algorithm by refinement. In [12], they improve their approach by introducing the concept of predicted values. McMillan [13] partly automates the proof presented in [11] with the help of compositional model checking.

Skakkebæk, Jones, and Dill [14] provide a formal method called incremental flushing to verify a scheduler supporting out-of-order execution. Arvind and Shen [15] follow a new approach. They use term rewriting systems to prove the data consistency of a Tomasulo scheduler which was enhanced by hardware for speculative execution.

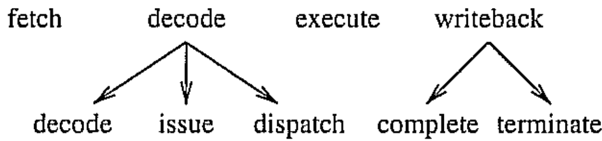
Except for [15] all these papers focus on hardware schedulers for out-of-order execution which do not support precise interrupts. In addition, all proofs concentrate on data consistency only and do not analyze whether the scheduler is deadlock-free. Nevertheless, the deadlock aspect cannot be ignored, since it was shown that the original Scoreboard, which is another major scheduler, can deadlock [16].

Contribution

This paper presents a variant of the Tomasulo scheduling algorithm which, in order to support precise interrupts, is enhanced by a reorder buffer. For this modified scheduling algorithm, we present a mathematical correctness proof. We show that data consistency is maintained, and that the scheduling algorithm is deadlock-free and fair. The fairness and the deadlock-free execution are shown by a worst case run-time analysis of an arbitrary program.

2. THE SCHEDULING ALGORITHM

In general, the execution of an instruction can be split into the following phases:



- Instruction fetch: The instruction is fetched from the instruction memory system into a special register.
- Instruction decode: The instruction is decoded, issued and dispatched. During issue, the instruction is assigned to a function unit, and together with its operands it is buffered in a queue. During dispatch, the data are passed to the function unit for the actual execution.
- Execution: The calculation and data transfer is performed.
- Writeback: Once an instruction leaves the function unit (complete), its result is buffered and forwarded to later instructions. In the final step (terminate), the result is written into the register file and it is checked for interrupts.

The original Tomasulo scheduling algorithm [6] is limited to two-address-instructions, but it is easy to extend the algorithm to handle today's common instructions with three or more addresses [8]. Since the original Tomasulo algorithm does not support precise interrupts, we have added a *reorder buffer* (ROB). In the following, we describe this variant of the Tomasulo scheduling algorithm [17]. Figure 1 gives an overview of the basic data paths of a design employing this scheduler.

During issue, an instruction is written into a *reservation station* of an appropriate function unit. The instruction is even issued if some of the operands of the instruction are still missing. The reservation stations therefore basically form a queue for the issued instructions.

As soon as all operands of the instruction in a reservation station are available, the instruction is ready to be dispatched into the actual function unit. This is the point where instructions leave the program order. When leaving the function unit, the result is stored in a result buffer and the *common data bus* (CDB) is requested for writing. As soon as the CDB is assigned to the function unit, the result data and the tag of the instruction is put on the CDB. The reservation stations snoop this bus for the operands they are missing. They can identify the results by the tag.

3. KEY DATA STRUCTURES

The Tomasulo scheduling algorithm requires the following data structures: the register files are extended by a producer table, and a set of reservation stations is assigned to each function unit.

Producer Table

Each register, named $R_i.data$, is extended by a tag and a valid flag. This extension is called *producer table*. The additional fields have the following purposes:

- The valid flag $R_i.valid$ of a register is set iff the corresponding register $R_i.data$ contains the appropriate data.
- If the valid flag is not set, the tag data item $R_i.tag$ of the register contains a tag pointing to the instruction which produces the desired result.

Reservation Stations

Each function unit is extended by an instruction buffer which queues the issued instructions. These buffer entries are called *reservation stations*. Each reservation station RS_i holds exactly one instruction and its operands and consists of the following components:

- The $RS_i.full$ bit is set iff the entry is in use.
- The $RS_i.tag$ item contains the ROB tag of the instruction buffered in reservation station RS_i. This item replaces the destination address used in the original Tomasulo algorithm.
- The $RS_i.op_1$ and $RS_i.op_2$ items hold the source operands of the instruction. They are a copy of the appropriate register file and producer table entries with same semantics.

The Reorder Buffer

In order to realize precise interrupts, we add a *reorder buffer* (ROB) [7]. An interrupt between instruction I_{i-1} and I_i is precise iff instructions $I_1,...,I_{i-1}$ are completed before starting the interrupt service routine and later instructions I_i, I_{i+1}, ... did not change the state of the machine [7].

The results of the instructions leave the function units in an arbitrary order. On completion, the reorder buffer gathers the results and later-on writes them into the register file in issue order. However, before writing the result of instruction I_i, it is checked whether this instruction causes an interrupt or not. Thus, in case of an interrupt, the register file contains exactly all modifications made by instructions I_1, ..., I_{i-1}.

The reorder buffer is realized as FIFO queue with a head pointer *ROBhead* and a tail pointer *ROBtail*. New instructions are put into the ROB entry pointed to by the tail pointer, i.e., *ROB*[*ROBtail*]. The instruction is passed to the reservation station together with this pointer *ROBtail* as tag.

When an instruction completes, both the result and the exception flags are written into the reorder buffer entry pointed to by its reorder buffer tag.

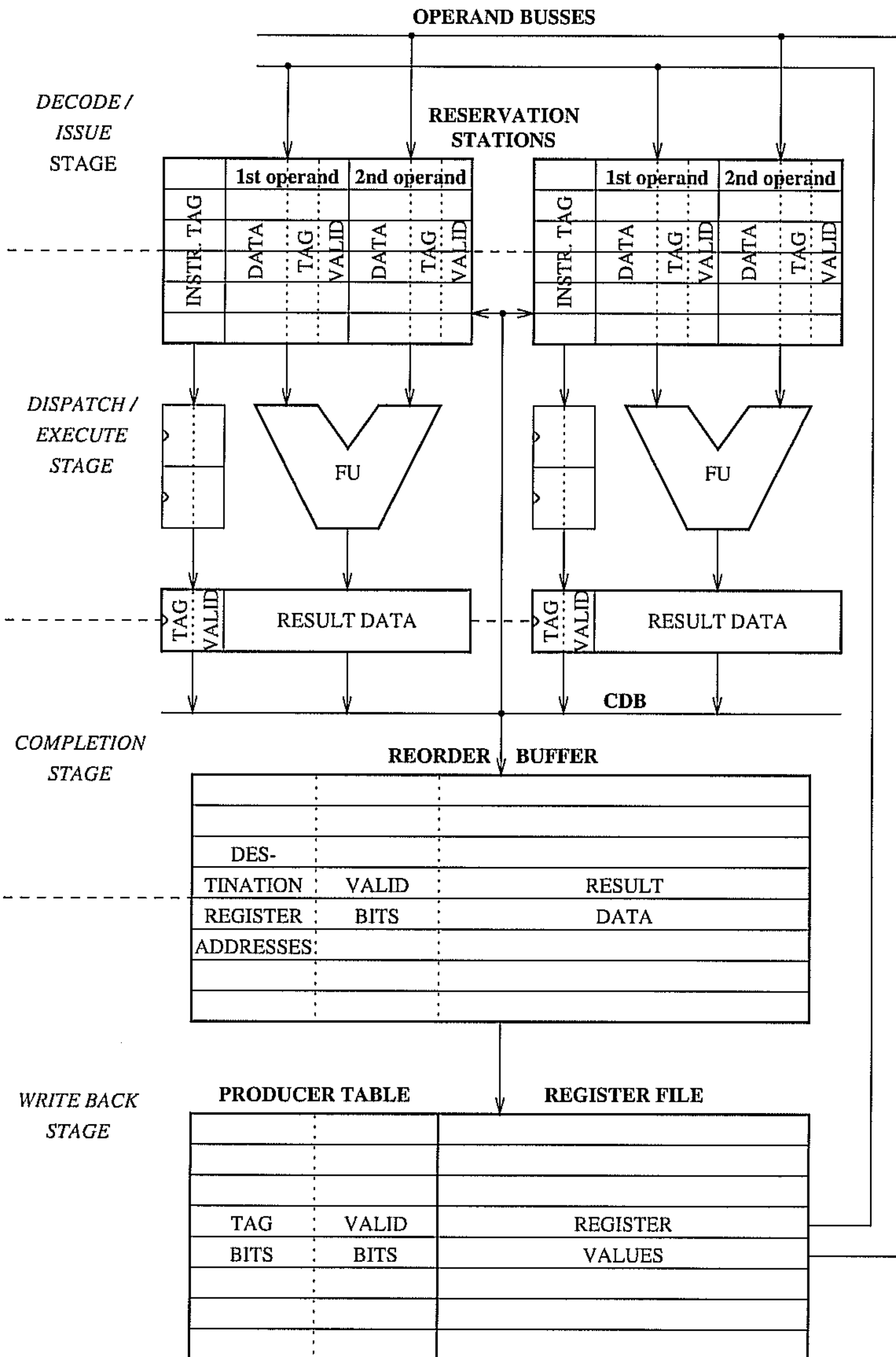


Figure 1: Structure of a CPU core with Tomasulo Scheduler

```
if there is a free RS and a free ROB entry
 {
   RS.full:=1;
   RS.tag:=ROBtail;
   For all operands x of I_i with address r
      if R_r.valid=1
         RS.opx:=R_r;
      else if CDB.tag=R_r.tag
         RS.opx:=CDB;
      else
         RS.opx:=ROB[R_r.tag];

   if (I_i has a destination register r)
      R_r.tag:=ROBtail;
      ROB[ROBtail].dest:=r;
   else
      ROB[ROBtail].dest:=none;

   ROBtail:=ROBtail+1;
 }
```

Figure 2: Issue protocol

In each cycle, the entry at the head of the reorder buffer is tested. If it is valid, i.e., the instruction has completed, a check for exceptions is performed and the data is written into the register file. Depending on the type of the interrupt, the result of I_i is written into the register file before executing the interrupt service routine.

The ROB address of the instruction is also used as a tag to identify the result. This is in contrast to the original Tomasulo design which uses tags associated with the reservation stations. To prevent that the same tag is used by different instructions simultaneously, the original Tomasulo requires a busy bit in each reservation station. This allows to keep track of the instructions which have been dispatched already. Since the new algorithm uses tags associated with a ROB entry, it can clear the reservation station right after dispatch and thus, the busy bits are no longer necessary.

3. THE SCHEDULING PROTOCOL

This section presents the scheduling protocol in detail [17, 18]. The execution of an instruction I_i is split into six phases: fetch, issue, dispatch, execution, completion and writeback.

Issue

Let I_i be the instruction to be issued. For issue, it is essential that an appropriate reservation station and a ROB entry are available (Fig. 2). If so, the instruction and its operands are issued into this reservation station entry. For each operand of the instruction three sources have to be checked: The operand can be in the register file, on the CDB, or in the reorder buffer. If the operand is the destination of a preceding, uncompleted instruction I_j, the operand is not valid yet, and then, instead of the result, the tag of instruction I_j is stored in the reservation station. The forwarding of operands directly from the CDB is essential for the correctness, as mentioned by [13].

Simultaneously, the ROB entry *ROB[ROBtail]* is allocated and initialized for the instruction. If the instruction has a destination register, the address of this register is stored in the ROB entry and the tail pointer *ROBtail* is stored as tag in the producer table entry of the destination register. Afterwards, the tail pointer is incremented.

```
if there is a RS with
   RS.opx.valid=1 for all operands x
   and the function unit is not stalled
 {
   Pass instruction, operands,
   and tag to FU

   RS.full:=0;
 }
```

Figure 3: Dispatch protocol

```
if FU has result and
   got CDB-acknowledge
 {
   CDB.valid:=1;
   CDB.data:=result from FU;
   CDB.tag:=tag from FU;

   ROB[CDB.tag].valid:=1;
   ROB[CDB.tag].data:=CDB.data;
 }
```

Figure 4: Completion protocol

Dispatch

During instruction dispatch (Fig. 3), a valid instruction moves from a reservation station into the actual function unit. An instruction is valid iff all its operands are valid. Furthermore, dispatch can only be performed if the function unit is not stalled, i.e., it must be ready to accept a new instruction. If several instructions for a certain function unit are valid, the scheduler has to choose one for dispatch. Our correctness proof relies on choosing the oldest among the valid instructions. The dispatch hardware must ensure this. After dispatch, the reservation station is freed.

In real hardware, the operands can also be forwarded via CDB. In contrast to the forwarding during issue, this forwarding is just an optimization and not necessary for correctness. Thus, this protocol element is omitted here.

Completion

Before completion (Fig. 4), the reservation station requests the CDB. As soon as the reservation station gets an acknowledge, the result and the ROB tag are put on the CDB. The according reorder buffer entry is filled with the result and its valid bit is set.

```
For all operands x:
   if RS.full=1 and RS.opx.valid=0
      and RS.opx.tag=CDB.tag
     {
      RS.opx:=CDB;
     }
```

Figure 5: CDB snooping protocol

```
if ROB not empty
   and ROB[ROBhead].valid=1
 {
   if instruction in the ROB[ROBhead]
      requires writeback
     {
      x:=ROB[ROBhead].dest;
      Rx.data:=ROB[ROBhead].data;
      if ROBhead=Rx.tag
         Rx.valid=1;
     }

   ROBhead:=ROBhead+1;
 }
```

Figure 6: Retirement / writeback protocol

Snooping on the CDB

On completion, the result of an operation is put on the CDB. Instructions in the reservation stations, which depend on this result, read the operand data from the CDB (Fig. 5). In order to do so, reservation stations with missing operands check wither the tag of the operand matches the tag on the CDB. If so, the result on the CDB is copied into the reservation station.

Retirement, Writeback, and Interrupts

During retirement (Fig. 6), a result of an instruction in the ROB is written into the register file. Note that the valid bit is only set if the tag in the register file matches the tag of the instruction. If the tags do not match, a later instruction with the same destination register was issued. Thus, a set valid bit truly indicates that the register holds its appropriate value.

In contrast to the original Tomasulo scheduling algorithm, the write back of the result value always takes place, even if the tags do not match. This is essential for the interrupt mechanism. Since the writeback of the results is made in program order, the register file after the writeback of instruction I_i matches the register file of after the sequential execution of I_1, ..., I_i.

Prior to writeback, a check for interrupts is performed. In case of interrupts such as page faults the writeback is skipped. On interrupts such as external interrupts used for I/O, the writeback takes place as usual. Then all interrupts are processed as follows: all valid bits of the registers are set, the complete instruction pipeline including the reservation stations, function units, and the reorder buffer is cleared, i.e., all remaining instructions in the queues are discarded.

4. DATA CONSISTENCY

In this section we show that data consistency is maintained. The following shorthands are used: $fetch(i)$, $issue(i)$, $disp(i)$, $compl(i)$ and $term(i)$ denote the cycles in with instruction I_i is fetched, issued, dispatched, completed and terminated, respectively.

Let I_1, ..., I_n be an instruction sequence in the scheduled execution. Data consistency intuitively means that the results generated by any instruction I_i conform to the semantics of a sequential execution. These semantics are defined as transition rules on an abstract configuration set $C(i)$ of the machine. The configuration includes all registers, and the rest of the program to be executed. The start configuration $C(0)$ is the configuration after the reset.

This sequential machine processes one instruction with each transition. Let the source registers of instruction I_i be $S_1(i)$ to $S_{s(i)}(i)$ with values $\sigma_1(i)$ to $\sigma_{s(i)}(i)$ and the destination register $D(i)$ with result value $\delta(i)$. We use $s(i)$ source registers to handle an arbitrary number of source registers. We limit the proof to one destination register for sake of simplicity, but it is easy to extend it to handle multiple destination registers per instruction.

Let op_i be the operation that instruction I_i performs. Instruction I_i can then be specified as

$$I_i\text{: } D(i) = \text{op}_i(S_1(i),..,S_{s(i)}(i))$$

with values

$$I_i\text{: } \delta(i) = \text{op}_i(\sigma_1(i),..,\sigma_{s(i)}(i))$$

As mentioned above, the abstract machine processes instruction I_i in transition i, which results in configuration $C(i)$. Registers not written by instruction I_i are passed unmodified from configuration $C(i-1)$.

Let $last(r,i)$ be index of the last instruction prior I_i which modified register r:

$$last(r,i) = \max\{\, j < i \mid D(j) = r \,\}$$

This allows for a simple criterion of data consistency for the scheduled execution.

Criterion For any instruction I_i, for any given source register x, the value of register x read by I_i must be the result of the last instruction writing it:

$$\sigma_x(i) = \delta(last(S_x(i), i)) \qquad \forall x \in \{1,..,s(i)\}$$

If there is no such instruction, $last(S_x(i), i)$ is undefined. It is therefore assumed that all registers, which are used as source register, are initialized by the program prior their first use.

In the rest of the section, we prove that this condition holds for a machine implementing the protocols of section 3. This is done with the help of five invariants.

Invariant 1 and 2 summarize the semantics of the register file items *valid* and *tag*. Depending on the value of the valid bit, these invariants state where the appropriate value of the register can be found. Invariant 3 serves the same purpose for the operand data in the reservation stations. Invariant 4 and 5 specify the source of the data on the CDB and in the ROB, respectively.

For any cycle t, let R_r^t denote the content of the register R_r during cycle t.

Invariant 1 Let $t = issue(i)$ be the cycle in which instruction I_i is issued. Any given register R_r in the register file which is marked valid in this cycle holds the result of the last instruction prior to I_i writing R_r.

$$R_r^t.valid = 1 \implies R_r^t.data = \delta(last(r, i))$$

Invariant 2 Let $t = issue(i)$ be the cycle in which instruction I_i is issued. Any given register R_r in the register file which is marked invalid in this cycle is the destination register of an previous instruction I_j ($j < i$) which has not retired yet. In this case, the tag item of the register holds $I_j.tag$, i.e., the tag of the instruction I_j. This instruction is the last instruction writing R_r, i.e., $j{=}last(r, i)$:

$$R_r^t.valid = 0 \implies R_r^t.tag = I_{last(r,i)}.tag$$

Invariant 3 In any cycle t, let instruction I_i be an instruction in reservation station RS_l. If there is an operand x in this reservation station which is not valid yet ($RS_l^t.opx.valid$=0), there must be a previous instruction I_j, which produces the value for the operand, and I_j has not completed yet. Thus, I_j it is the last instruction prior to I_i writing the source register of I_i. The tag of instruction I_j is the tag in $RS_l^t.opx.tag$.

$$\begin{aligned} &RS_l^t.opx.valid = 0 \quad \text{and} \quad I_i \text{ in } RS_l \\ &\implies RS_l^t.opx.tag = I_{last(S_x(i),i)}.tag \end{aligned}$$

Invariant 4 In any cycle t, let I_i be an instruction in issue or dispatch stage, i.e., $issue(i) \leq t \leq dispatch(i)$. Let I_i have a source operand $r = S_x(i)$ with tag $S_x(i).tag$. If the CDB is valid and if the CDB tag is equal to the tag of the operand, the result of the last instruction prior to I_i writing the operand is on the CDB.

$$CDB^t.tag = S_x(i).tag \implies \delta(last(r, i)) = CDB^t.data$$

Invariant 5 In any cycle t, let I_i be an instruction in issue or dispatch stage. Let I_i have a source operand $r = S_x(i)$ with tag $S_x(i).tag$. If the ROB entry pointed to by the tag of the operand is valid, the result of the last instruction writing the operand is in this ROB entry.

$$\begin{aligned} &ROB^t[S_x(i).tag].valid = 1 \\ &\implies \delta(last(r, i)) = ROB^t[S_x(i).tag].data \end{aligned}$$

With the help of these invariants, the proof of the data consistency is done by induction over the number n of instructions. For $n = 0$, the claim is obvious, since no instructions have been issued. The induction for $n > 0$ requires a distinction of two cases. Let instruction I_i read source operand $S_x(i)$. The protocol permits this in two different phases.

- Let instruction I_i read register $r{=}S_x(i)$ **during issue.** In dependence of the value of $R_r.valid$, either invariant 1 or invariant 2 applies. If $R_r.valid$ is set, the operand is copied from the register file and the claim is an implication of invariant 1.

  If $R_r.valid$ is not set, invariant 2 states that R_r contains the tag of the instruction I_j which produces the result. As the operand of the instruction is already available during issue, it is copied from either the CDB or the ROB, and then either invariant 4 or 5 applies. Thus, the result in the ROB or on the CDB is $\delta(last(r, i))$. The instruction I_j had correct operands because of $j < i$.

- Let instruction I_i read register $r{=}S_x(i)$ **while in reservation station** RS_l, i.e., the operand is read by the CDB snooping protocol. This only happens if $RS_l.opx.valid$ is not set. In this case, invariant 3 states that the tag in $RS_l.opx.tag$ is the tag of the instruction I_j which produces $\delta(last(r, i))$. The protocol requires this tag to be equal to the tag on the CDB, thus invariant 4 applies. I_i therefore reads the result of I_j. This is the correct result because of $j < i$.

5. TERMINATION AND FAIRNESS

The termination proof will show that an instruction sequence I_1, ..., I_n is processed in a finite amount of clock cycles.

Hardware Constraints

Our proof requires several hardware properties:

- As mentioned above, the dispatch hardware must choose the oldest among the valid instructions.
- The CDB must be allocated round robin to the requesting producers.
- Our proof assumes fully pipelined function units. However, it is easy to extend it to handle iterative function units such as floating point dividers.

Note that all instructions are issued in program order and terminate in program order, i.e., issue(i) $<$ issue(j) and term(i) $<$ term(j) for any $i < j$. In-order termination is an implication of the fact that an instruction terminates when leaving the ROB. The ROB is a FIFO queue and is filled in program order.

Consequently, to prove termination, it is sufficient to prove that a finite α exists with

$$\text{term}(i) + \alpha \geq \text{term}(i+1) \quad (1)$$

for any i, i.e., to find an upper bound for the number of cycles of an instruction. A weak bound of α to satisfy Eq. (1) can be determined by summing up the maximum time which an instruction spends in each stage after all previous instructions terminated. Let α_0 denote the maximum time spent for the instruction fetch and so on.

$$\alpha \leq \alpha_0 + ... + \alpha_4 \quad (2)$$

The bound (1) is proved by induction over the number n of instructions.

Fetch The instruction fetch can be performed if the issue stage is not stalled. After all previous instructions terminated, the issue stage becomes available at the latest and instruction I_i can be fetched. Let $lmem$ be the maximum latency of the memory, then $\alpha_0 = lmem$, i.e.:

$$\text{fetch}(i) \leq term(i-1) + lmem$$

Issue Instruction I_i can be issued if there is a free, appropriate reservation station and if the ROB is not full. Both conditions hold one cycle after all previous instructions terminated at the latest. Thus, $\alpha_1 = 1$.

Dispatch Instruction I_i can be dispatched iff all its operands are valid and if the function unit is able to accept data. Obviously, the operands are valid after all previous instructions terminated. However, the function unit might be blocked by later instructions, i.e, instructions I_j with $j > i$. Assuming fully pipelined function units, the function unit can accept a new instruction as soon as one result leaves the function unit. However, before an instruction can leave the function unit, it is necessary to wait for the CDB. Since the CDB is allocated round robin, this takes at most f cycles, with f being the number of function units. Thus, it takes $\alpha_2 = f$ cycles at most until the function unit accepts a new instruction. As I_i is the oldest valid instruction in the reservation stations (all previous instructions already terminated), I_i is dispatched afterwards.

Completion Let l be the number of pipeline stages of the function unit (including the result buffer pipeline stage). For each instruction in the pipeline, the CDB has to be requested which takes at most f cycles per instruction. Thus, instruction I_i leaves the function unit (completes) after at most $\alpha_3 = l \cdot f$ cycles.

Termination A valid (completed) instruction in the ROB terminates one cycle after all previous instructions terminated. Thus, $\alpha_4 = 1$.

6. CONCLUSION

Although already introduced in 1967, the Tomasulo scheduler can be enhanced to meet today's requirements like precise interrupt handling. Such a Tomasulo scheduler then outperforms other competing schedulers at a reasonable cost (gate count) increase, as shown in [5]. Thus, the Tomasulo scheduling algorithm remains a good choice for current microprocessor designs.

Despite of the complexity of the enhanced Tomasulo scheduler, the paper gives a fairly simply correctness proof which also covers the deadlock aspect. However, our proof omits the scheduling problem within complex function unit such as iterative floating point dividers. The combination of Tomasulo scheduling with variable latency function units is for example covered by [19]. Furthermore, our proof lacks automatization. The concept of model checking should be applied to modern variants of the well known scheduling algorithms.

References

[1] Robert P. Colwell and Randy L. Steck. A 0.6um bicmos processor employing dynamic execution. International Solid State Circuits Conference (ISSCC), 1995.

[2] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995.

[3] L. Gwennap. PPC 604 powers past Pentium. *Microprocessor Report*, 8(5):1, 6–9, 1994.

[4] PowerPC 750 RISC Microprocessor Technical Summary, 1997.

[5] Silvia M. Müller, Holger Leister, Peter Dell, Nikolaus Gerteis, and Daniel Kröning. The impact of hardware scheduling mechanisms on the performance and cost of processor designs. In *In Proc. 15th GI/ITG conference 'Architektur von Rechensystemen' ARCS'99.*, 1999.

[6] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11 (1), pages 25–33. IBM, 1967.

[7] J.E. Smith and A.R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[8] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[9] H.C. Torng and Martin Day. Interrupt handling for out-of-order execution processors. *IEEE Transactions on Computers*, 42(1):122–127, 1993.

[10] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. 10th International Conference on Computer-aided Verification (CAV)*, 1998.

[11] W. Damm and Pnueli A. Verifying out-of-order executions. In H.F. Li and D.K. Probst, editors, *Advances in Hardware Design and Verification: IFIP WG 10.5 Internatinal Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 23–47. Chapmann & Hall., 1997.

[12] T. Arons and A. Pnueli. Verifying Tomasulo's algoithm by refinement. In *Proc. 12th International Conference on VLSI Design*, 1999.

[13] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by composition model checking. In *Proc. 10th International Conference on Computer Aided Verification*, pages 110–121, 1998.

[14] Jens U. Skakkebæk, Robert B. Jones, and David L. Dill. Formal verification of out-of-order execution using incremental flushing. In *10th International Conference on Computer Aided Verification*, 1998.

[15] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro Special Issue on Modeling and Validation of Microprocessors, May/June*, 1999.

[16] S.M. Müller and W.J. Paul. Making the original scoreboard mechanism deadlock free. In *Proc. 4th Israel Symposium on Theory of Computing and Systems (ISTCS)*, pages 92–99. IEEE Computer Society, 1996.

[17] N. Gerteis. The performance impact of precise interrupt handling on a RISC processor (German). Master's thesis, University of Saarland, Computer Science Department, Germany, 1998.

[18] Daniel Kröning. Design and evaluation of a RISC processor with a Tomasulo scheduler. Master's thesis, University of Saarland, Computer Science Department, Germany, 1999.

[19] Silvia M. Müller. On the scheduling of variable latency functional units. In *Proc. for the 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA'99).*, 1999.