Journal of Circuits, Systems, and Computers, Vol. 8, No. 2 (1998) 000–000

ON THE CORRECTNESS OF HARDWARE SCHEDULING MECHANISMS FOR OUT-OF-ORDER EXECUTION

SILVIA M. MUELLER* and WOLFGANG J. PAUL
Dept. 14: Computer Science, University of Saarland
66123 Saarbruecken, Germany
E-mail: {smueller,wjp}@cs.uni-sb.de

Received 2 July 1997
Accepted 6 March 1998

Hardware scheduling mechanisms are commonly used in current processors in order to make better use of instruction level parallelism. So far, such a mechanism is considered to be correct, if it avoids the standard structural and data hazards. However, based on two classical scheduling mechanisms, it will be shown that this condition is neither sufficient nor necessary for the correctness of such a mechanism, and that deadlocks are a serious matter in out-of-order execution as well. In addition, the paper provides sufficient conditions for the correctness of scheduling mechanisms.

1. Introduction

Current processors[3] comprise multiple function units which can work in parallel. Since the latency of the function units varies a lot, designs like PowerPC, Pentium-Pro, and MIPS R10000 allow instructions to overtake each other in order to achieve a better hardware utilization and a better performance. The instruction execution and the resources are then governed dynamically by scheduling mechanisms, most of which are based on the Scoreboard[10,3] and the Tomasulo algorithm.[11,3]

Such a mechanism should sequence the instructions as fast as possible and achieve concurrent instruction execution, but it must also ensure that the program performs the same task as in sequential (in-order) execution. Thus, checking the correctness of a scheduling mechanism becomes a hard but essential part of the design.

In the standard literature on computer architecture[1,3,5,6] the correctness of a scheduling mechanism is usually motivated by the fact that the structural and control hazards, as well as the data hazards (write after write, read after write, write after read) are properly resolved. However, this criterion was never proven to be sufficient.

On the other hand, out-of-order execution is a special type of shared memory computation. It achieves concurrent execution on instruction level. The instruc-

*Funded by the German Science Foundation DFG

tions, executed by parallel function units, all access the same set of registers and the same memory. Thus, the sufficient correctness criteria from the theory of parallel computing are also applicable to hardware scheduling mechanisms. These criteria are usually based on the data consistency and the deadlock free execution.

Outline: In Sec. 3, we adapt these correctness criteria to the hardware scheduling mechanisms. In Sec. 4, we prove that avoiding hazards implies data consistency, but not vice versa. There are well-known scheduling mechanisms which read and write consistent data but for which the hazard criterion is not applicable. In Sec. 5, we show that avoiding data hazards and ensuring data consistency is not enough, the scheduling mechanism must also ensure a deadlock free execution.

In order to present and prove sufficient conditions for the correctness, it is helpful to have some notation on the structure and timing of the instructions, and to formalize the data hazards.

2. Notation

For the timing, we argue on a cycle by cycle basis. Therefore, we number the cycles by $t = 1, 2, \ldots$.

2.1. *Structure of the instructions*

Definition 1: *For $n, m \geq 1$, a machine is of* type (n, m) *if and only if all its instructions read at most n operands and write results to at most m storage components, and if no instruction reads its arguments before it got issued, or writes a result before computing it.*

For simplicity's sake, the paper only addresses machines of type (2,1), although that is not the only interesting case. For example, a standard IEEE floating-point operation, among others, updates flags which signal overflow and underflow. These results depend on two floating-point numbers, on the rounding mode, and on the value of the interrupt masks. However, the definitions and theorems presented here can easily be generalized for larger n and m.

For a (2,1) architecture $\mathcal{A}$ every instruction of a program $P = I_1, \ldots, I_p$ is a three-address instruction

$$I_i : D(i) = S1(i)\ op(i)\ S2(i)$$

with sources $S1(i)$, $S2(i)$, with a destination $D(i)$, and an operation $op(i)$.

Value of the Data Referenced: For an architecture $\mathcal{A}$ the values of the data referenced by the instructions of a program P basically depend on two things, namely on the initialization of the storage components (registers) and on the timing of the accesses. Thus, let us assume a fixed initialization of the architecture $\mathcal{A}$. For any instruction I_i of the program P running on $\mathcal{A}$

- $\sigma_1(i)$, $\sigma_2(i)$ denote the values of the two source operands of I_i and $\delta(i)$ denotes the value of its result under in-order execution;
- $\sigma'_1(i)$, $\sigma'_2(i)$ and $\delta'(i)$ denote the values of its two arguments and of its result under out-of-order execution.

Timing of the Accesses: During execution, the instructions have to pass several phases, namely: the issuing, the reading of the operands, the execution of the operation, the writing of the result, and the termination. The scheduling mechanism must specify the timing of these phases for each instruction of the program to be executed.

Let us assume that, except for the actual execution of the operation, any of these actions is performed in a single cycle. For every instruction I_i in the out-of-order execution of program P,

- $issue(i)$ denotes the issue cycle of I_i. During this cycle, I_i is assigned to a function unit capable of executing operation $op(i)$;
- $read(i)$ denotes the cycles during which I_i reads its operands $S1(i)$ and $S2(i)$;
- $write(i)$ denotes the cycle during which I_i writes its result back in $D(i)$;
- $term(i)$ denotes the cycle during which I_i terminates. After this cycle, instruction I_i has given free all the resources involved in its execution.

During a cycle t, an instruction I_i is called *active* iff it got issued before t and did not terminate yet:

$$issue(i) < t \leq term(i)\,.$$

2.2. *Data and structural hazards*

With respect to data hazards, we distinguish between data dependences and data conflicts. The dependences are inherent in the program. A conflict denotes the wrong order of read/write accesses in the execution of a program and is caused by the scheduling mechanism.

Definition 2 (Write After Write): *In a program P, there is a* WAW dependence *$WAW(i,j)$ between two instructions I_i and I_j with $i < j$ iff both instructions have the same destination*:

$$WAW(i,j) \iff D(i) = D(j)\,.$$

The instructions I_i and I_j cause a WAW conflict *iff there exists a WAW dependence and the writes occur in reversed order, i.e., iff $WAW(i,j) \wedge write(i) \geq write(j)$.*

Definition 3 (Read After Write): *In a program P, there is a* RAW dependence *$RAW(i,j)$ between two instructions I_i and I_j with $i < j$ iff I_i reads the register written by I_j :*

$$RAW(i,j) \iff D(i) \in \{S1(j), S2(j)\}\,.$$

The instructions I_i and I_j cause a RAW conflict *iff there exists a RAW dependence and the reads of I_j overtake the write of I_i, i.e., iff $RAW(i,j) \wedge write(i) \geq read(j)$.*

Definition 4 (Write After Read): *In a program P, there is a* WAR dependence *$WAR(i,j)$ between two instructions I_i and I_j with $i < j$ iff I_j writes to one of the source registers of I_i :*

$$WAR(i,j) \iff D(j) \in \{S1(i), S2(i)\}\,.$$

The instructions I_i and I_j cause a WAR conflict *iff there exists a WAR dependence and the reads of I_i are overtaken by the write of I_j, i.e., iff $WAR(i,j) \wedge read(i) \geq write(j)$.*

Definition 5 (Structural Hazard): *The scheduling mechanism of an architecture $\mathcal{A}$ produces a structural hazard if there exists a cycle t in the execution of a program P for which the mechanism scheduled more read/write accesses or more operations than $\mathcal{A}$ can perform in that cycle.*

Those dependences occur in almost any program, but they only become problematic in case the scheduling mechanism does not take the proper precautions.

Control hazards are special data hazards which involve the program counter.

3. Correctness Criteria

In the theory of parallelizing compilers,[1,4,12] the correctness of the concurrent execution of a program is modeled as follows:

Criterion 6: *The concurrent execution of an instruction sequence P is correct,*

6.1. *if exactly the same instructions are executed as in the sequential (in-order) execution, and*

6.2. *if the data are consistent, i.e., each instruction reads and writes the same data as in sequential execution, and writes to the same variable do not overtake each other.*

This sufficient criterion is also applicable to hardware scheduling mechanisms. However, they work on the level of machine instructions and manipulate registers and memory words instead of variables. At any given time, the schedulers only see a small fraction of the whole program sequence. Thus, it is desirable to simplify Criterion 6, such that one only has to consider pairs of (dependent) instructions.

After addressing the data consistency, we focus on the deadlock aspects which cover condition (6.1), and we then derive three additional sufficient criteria for the correctness of a scheduling mechanism.

3.1. *Data consistency*

The Bernstein condition[1,2,4,5,12] checks the consistency of a (parallel) program on the level of single instructions:

Lemma 1 (Bernstein Condition): *Two blocks of code* $A = I_{i_1}, \ldots, I_{i_a}$ *and* $B = I_{j_1}, \ldots, I_{j_b}$ *can be executed in any order (especially in parallel) if the codes* A *and* B *are data independent, i.e., there exists no data dependence between any two instructions from* A *and* B:

$$\begin{aligned} &\nexists\, I_i \in A, I_j \in B \text{ with} \\ &\quad RAW(i,j) \vee WAR(i,j) \vee RAW(i,j) \\ &\vee RAW(j,i) \vee WAR(j,i) \vee RAW(j,i)\,. \end{aligned}$$

For a scheduling mechanism, the code blocks comprise a single instruction but their execution is broken down in several phases, and these phases are the source of the parallelism. In this case, the Bernstein condition can be applied to the phases of the instruction execution, and it would then not be allowed to overlap the execution of two instructions with a data dependence. Generally, this condition is far too restrictive; floating point instructions could not be overlapped because they update the same flags. As far as scheduling mechanisms are concerned, it is therefore standard to consider data *conflicts* rather than data *dependences*.[3,1,6]

Definition 7 (Hazard Condition): *A scheduling mechanism of an architecture* $\mathcal{A}$ *satisfies the hazard condition if*

1. *it avoids all structural hazards, and if*
2. *it resolves all data dependences, i.e., if it avoids WAW, RAW and WAR conflicts.*

In Sec. 4, it is proven that the hazard condition implies data consistency, but that the hazard condition is not applicable in combination with the result forwarding.

3.2. *Avoiding deadlocks*

After rearranging or parallelizing the code, it is fairly easy for a compiler to verify that still the same instructions are used. In order to ensure condition (6.1), it must only be ensured that the program runs to completion, i.e., that it does not run into a deadlock.

In general, it is more complicated to ensure condition (6.1) for a hardware scheduling mechanism. However, a common type of scheduling mechanisms (e.g., the Scoreboard) receives the instructions in program order from the fetch/branch unit and issues them in order. Such a scheduling mechanism executes exactly the instructions of the current program (maybe in a different order) if the mechanism does not run into a deadlock, i.e., if every instruction which is issued also terminates. Thus, the following lemma provides a sufficient condition for such a scheduling mechanism to be deadlock free and to satisfy condition (6.1). The lemma can easily be proven by induction on the index of the instructions.

Lemma 2 (Deadlock free): *Let $T(i)$ denote the cycle in the scheduled execution of P after which all instructions $I_1, \ldots, I_{i-1}$ have terminated:*

$$T(i) = \begin{cases} 0 & ; \quad i = 1 \\ \max\{term(j) | j < i\}\,; & i > 1\,. \end{cases}$$

If there exists a constant α such that for any instruction I_i

$$term(i) \leq T(i) + \alpha$$

then I_i terminates in cycle

$$\tau(i) \leq T(i+1) \leq i \cdot \alpha$$

at the latest, and the execution is deadlock free. Thus, an instruction I_j is only allowed to delay a later instruction I_k $(k > j)$.

Of course, this lemma only holds in the absence of interrupts and wrong speculation, because otherwise instructions which are already partially executed might be rolled back and evicted. However, these aspects are beyond the scope of this paper and must be solved when designing interrupt or speculation hardware.

For the basic scheduling mechanism with in-order issue, lemma simplifies condition (6.1). However, for hardware scheduling mechanisms, the deadlock aspect is usually ignored. Thus, is there something truly special about instruction level parallelism that deadlocks cannot occur? The answer is *no*, because in Ref. 7 it is shown that, in spite of data consistency, the Scoreboard mechanism of Refs. 10 and 3 can run into a deadlock; but the mechanism can easily be fixed. Consequently, *deadlocks are a serious matter in out-of-order execution as well.* In Sec. 5, we will sketch this study.

3.3. *Sufficient criteria*

In the previous sections, we have derived conditions which under certain hypotheses imply data consistency (6.2) or imply that the same instructions are executed as in the in-order execution (condition 6.1). When combining these conditions, one obtains the following three criteria which are sufficient for the correctness of a scheduling mechanism. Note that these criteria, especially the third one, are less general than Criterion 6.

Criterion 8: *A scheduling mechanism of an architecture $\mathcal{A}$ is correct if for every program P of $\mathcal{A}$, it satisfies at least one of the following three sets of conditions*:

1. *The scheduling mechanism satisfies the hazard condition* (*Definition* 7), *and the same instructions are executed as under in-order execution* (6.1).
2. *The scheduling mechanism issues the instructions in-order*, *satisfies the deadlock condition of Lemma* 2, *and reads and writes consistent data* (6.2).
3. *The scheduling mechanism issues the instructions in-order and satisfies the deadlock condition of Lemma* 2 *and the hazard condition* (*Definition* 7).

4. The Hazard Condition

It is tempting to believe that the hazard condition and the consistency condition are equivalent, but that is not the case. Avoiding hazards implies data consistency (Sec. 4.1), but not vice versa (Sec. 4.2). There exist standard scheduling mechanisms which read and write consistent data, but for which the hazard condition is not applicable.

4.1. *Hazard condition implies data consistency*

In this section, we rigorously prove the folklore theorem, that the hazard condition implies data consistency. The proof is easy but we have not found it in the open literature. As a first step, we show that concurrent writes are precluded, and a read and a write operation accessing the same register cannot occur in the same cycle.

Lemma 3: *In case a scheduling mechanism of a* $(2,1)$ *architecture* $\mathcal{A}$ *properly resolves WAW*, *RAW and WAR dependences*, *then in any cycle t it either schedules at most one write access per register* (*storage component*) R *or some read accesses for R. This holds for any program P of $\mathcal{A}$ and for any of its registers.*

Proof: (1) Let us assume that I_i and I_j are two instructions with $i < j$ which write to the same register $R = D(i) = D(j)$ during cycle t. Then, there exists a dependence WAW(i,j), and due to $t = write(i) = write(j)$ these two instructions also cause a WAW conflict. However, that is a contradiction to the hypotheses of the lemma. Thus, in any cycle t there occurs at most one write access per register.

(2) Let us assume that instruction I_i writes register $R = D(i)$ which instruction I_j reads in the same cycle, i.e., $D(i) \in \{S1(j), S2(j)\}$ and $t = write(i) = read(j)$.

- If $i < j$ or $i > j$, then exists a dependence RAW(i,j) or WAR(j,i) and the two instructions cause a RAW or a WAR conflict. Either would be a contradiction to the hypotheses of the lemma.
- If $i = j$, then instruction I_i writes before it had time to read its operands and to compute the result, but that is impossible on a (2,1) architecture.

Consequently, a register is never read and written at the same time.

□

This lemma implies that for any cycle, reads and writes can be considered separately, because they do not interfere with each other.

Lemma 4: *Let $\mathcal{A}$ be a $(2,1)$ architecture. Let the scheduling mechanism produce no structural hazards, and for every program P, let the architecture execute the same instructions as in the in-order execution of P. The proper resolving of the WAW, WAR and RAW dependences then implies the data consistency, i.e., in the out-of-order execution, any instruction I_i reads and writes the same data as in the sequential execution of P:*

$$\sigma_1(i) = \sigma_1'(i)\,, \quad \sigma_2(i) = \sigma_2'(i)\,, \quad \delta(i) = \delta'(i)\,.$$

Proof: Let the execution start in cycle $t = 1$. It will be shown that instructions reading or writing during cycle t read or write values consistent with those of the sequential execution, i.e., $\forall\ t$

$$write(i) = t \Rightarrow \delta'(i) = \delta(i)$$

$$read(j) = t \Rightarrow \sigma_1'(j) = \sigma_1(j) \wedge \sigma_2'(j) = \sigma_2(j)\,.$$

$t = 0$: So far, the execution of P has not started yet, and has neither read nor written inconsistent data.

$t-1 \rightarrow t$: According to Lemma 3, concurrent writes are precluded, and a read and a write operation accessing the same register cannot occur in the same cycle. Thus, the reads and writes of a cycle can be analyzed separately.

Write Accesses: Let I_i be an instruction writing during cycle t, i.e., $write(i) = t$. The corresponding read accesses of the operands occurred in a cycle $0 < t_0 < t-\varepsilon(i)$, where $\varepsilon(i)$ denotes the latency of operation op(i). According to the induction hypothesis, I_i read consistent data

$$\sigma_1'(i) = \sigma_1(i) \quad \text{and} \quad \sigma_2'(i) = \sigma_2(i)\,.$$

Since the function unit had enough time to produce the result $\delta'(i)$ it holds:

$$\delta'(i) = \sigma_1'(i)\, op(i)\, \sigma_2'(i) = \sigma_1(i)\, op(i)\, \sigma_2(i) = \delta(i)$$

Consequently, only consistent data can be written in cycle t.

Read Accesses: Let I_j be an instruction which during cycle t reads its two arguments $S1(j)$ and $S2(j)$, i.e., $read(j) = t$. Since RAW and WAR conflicts are precluded, all write accesses to registers $S1(j)$ and $S2(j)$ of preceeding instructions I_i did already occur before cycle t, and the write accesses to these registers of succeeding instructions I_k will occur after cycle t, i.e., $\forall\ i < j < k$ with RAW(i,j) and WAR(j,k):

$$write(i) < read(j) = t < write(k)\,.$$

- If the instructions $I_1, \ldots, I_{j-1}$ only update registers different from $S1(j)$ and $S2(j)$, the instruction I_j reads the initial value of its source registers. Due to a fixed initialization, $\sigma'_1(j) = \sigma_1(j)$ and $\sigma'_2(j) = \sigma_2(j)$.
- Let $Sx(j)$ denote one of the source registers of instruction I_j. If register $Sx(j)$ is written by an instruction I_i with $i < j$, then let $(i_l)_{l=0}^k$ be the sequence of indices $i_l < j$ such that $D(i_l) = Sx(j)$ in rising order. The corresponding instructions cause WAW(i_l, i_{l+1}) and RAW(i_k, j) dependences. Since WAW and RAW conflicts are avoided, it follows that

$$write(i_0) < \cdots < write(i_k) < read(j) = t$$

and in both types of executions, instruction I_{i_k} performs the last write access to register $Sx(j)$ before the read access of I_j. Thus, I_j reads the consistent value of its operand $Sx(j)$:

$$\sigma'_x(j) = \delta'(i_k) = \delta(i_k) = \sigma_x(j)\,.$$

Consequently, in cycle t, as in prior cycles, only consistent data are read and written. Thus, avoiding hazards implies data consistency.

□

4.2. *Limits of the hazard condition*

The pipelining concept is a simple scheduling mechanism which is fairly well understood. Textbooks[3,9] describe in great length how to pipeline a RISC processor and how to solve data dependences by forwarding. It is well known, that such a pipelined design reads and writes consistent data. Nevertheless, we show now that the hazard condition is not applicable to such a pipelined design.

Implications: Although, pipelining is a very simple scheduling mechanism, the example will show that the problems are solely due to the forwarding. Many of the current scheduling mechanisms are based on the Tomasulo algorithm[11,3] for which forwarding is one of the key features. Thus, for all these scheduling mechanisms, the hazard condition is not applicable, and one has to rely on the consistency condition in order to prove their correctness.

Example: Consider a standard 5-stage pipeline with the stages – fetch (F) – decode and operand fetch (D) – execute (E) – memory access (M) – write back (W). On this design, we run the program of Fig. 1. There is a WAW dependence between the instructions I_1 and I_2, and a RAW dependence between I_2 and I_3. Instruction I_3 reads its arguments in cycle $t = 4$; this is a few cycles before I_2 updates register R1. However, the result of I_2 is already valid in cycle $t = 3$. Due to forwarding, I_2 gets this value on time. Thus, the pipeline reads and writes consistent data.

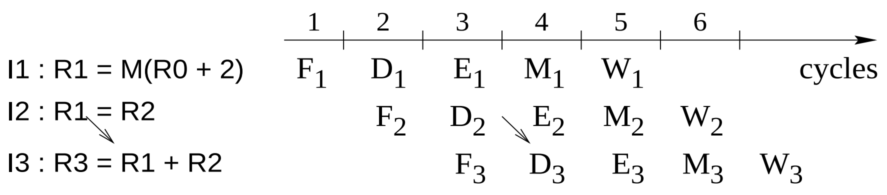

Fig. 1. Execution of the test program on a standard 5-stage pipeline with forwarding.

Due to the WAW(1,2) and RAW(2,3) dependences, the hazard condition requires the following order of reads and writes:

$$\mathit{write}(1) < \mathit{write}(2) < \mathit{read}(3) = 4\,.$$

In a pipelined design, two definitions are feasible for the write cycle $\mathit{write}(i)$ of instruction I_i: either $\mathit{write}(i)$ denotes the write back cycle of I_i, or it denotes the first cycle during which the result of I_i becomes available. In the first case, the test program encounters an RAW conflict

$$4 = \mathit{read}(3) < \mathit{write}(2) = 6\,,$$

and in the second case, the program encounters a WAW conflict, because the result of I_1 becomes first available in cycle $t = 4$,

$$3 = \mathit{write}(2) < \mathit{write}(1) = 4\,.$$

Thus, the moment (cycle) of a "write access" is no longer well defined, and the hazard criterion is therefore not applicable.

If RAW dependences are resolved by stalling instead of forwarding, and if $\mathit{write}(i)$ is defined as the write back cycle of I_i, then the pipelined design also satisfies the hazard condition. Thus, the problems with the hazard condition are due to the forwarding.

5. The Scoreboard Mechanism

The Scoreboard is one of the two main hardware scheduling mechanisms. It was first introduced in the CDC 6600, but variations of this mechanism are still in use. Since it is almost impossible to extract the underlying mechanism from a hardware implementation, one has to rely on the literature, e.g., in Refs. 10 and 3.

In Ref. 3 it is shown in an informal way that this scheduling mechanism resolves structural and data hazards, but in this section it is shown how to drive this mechanism into a deadlock. However, a small but subtile modification is enough to make the Scoreboard deadlock free.

Note, we do not claim that any of the implemented Scoreboard variants have this flaw, just the one described in the literature.

5.1. *The scoreboard mechanism*

The Scoreboard issues the instructions in-order, one at a time, but enables an out-of-order execution. In order to keep track of the instructions, the Scoreboard provides two data structures, one for the registers and one for the function units. For simplicity's sake, we assume that the architecture is of type (2,1) and comprises f function units $F_1, \ldots, F_f$.

Registers. For each register R, the Scoreboard specifies whether R is currently reserved as destination register, and if so, which function unit is going to produce the result.

Function Units. Every instruction I_i of a program P goes through several phases, i.e., issuing, reading operands, execution of the operation, write back of the result. During the instruction issue phase, I_i is assigned to a free function unit $F(i)$ which in the following cycles performs the actions required by I_i. Since all this happens under the control of the Scoreboard, it stores the phase of each function unit F. The Scoreboard also provides entries for their current operation ($F.op$), and for the addresses of their source and destination registers ($F.S1, F.S2, F.D$). The flags $F.Vx$ ($x = 1, 2$) are used to check for valid data, but they are not true valid flags. $F.Vx = 0$ denotes that the data of operand $F.Sx$ is not valid yet, or unit F has already read it. The entries $F.Px$ specify the unit which is going to *produce* operand $F.Sx$. The value 0 indicates that the operand was already valid during instruction issue.

The Bookkeeping: The Scoreboard governs the resources and every function unit F according to the following rules:

Instruction Issue. The next instruction I_i is issued as soon as the destination register $D(i)$ and a function unit $F(i)$ capable of executing operation $op(i)$ become available. The Scoreboard then reserves register $D(i)$ and unit $F(i)$, and it initializes the table entries of unit $F(i)$.

Read Operands. After issuing an instruction to function unit F, that unit tries to read its arguments. Unit F checks whether the registers to be read are up-to-date, i.e., whether the valid flags of both operands are set ($F.V1 \wedge F.V2$). If so, the registers are read, and the flags $F.Vx$ are cleared.

Write Back. After executing the actual operation, unit F tries to write back its result. However, the Scoreboard must avoid the writing, as long as there is another function unit $Fu \neq F$ which has register $F.D$ as a source and still has to read its old value: $Fu.Sx = F.D \wedge Fu.Vx$. While the unit F writes its result into register $F.D$, the Scoreboard gives this register free.

Notification. One cycle after the write back, the Scoreboard gives unit F free and notifies *all* function units with a source depending on a result of unit F. Corresponding valid flags are activated: if $Fu \neq F \wedge Fu.Px = F$ then $Fu.Vx := 1$. Note also that units with instructions depending on an earlier result of unit F are notified.

5.2. *A counter example*

The notification mechanism is counter intuitive, and in combination with the unusual valid flags it becomes problematic. Consider a slow function unit F waiting for its source S1 to be produced by a fast unit. In case the fast unit completes another operation after F read its argument but before it terminated, then the generalized set condition in the notify mechanism forces the valid flag $F.V1$ back to 1. This flag cannot be cleared till a new instruction is issued to unit F. Then $F.V1 = 1$, even so unit F has already read its operands.

Along these lines, we now construct a program which forces the Scoreboard into a deadlock. Let the model architecture comprise the three function units of Table 1 and use the above Scoreboard to sequence the code segment of Table 1.

Table 1. Latency (in cycles) of the function units.

	F1	F2	F3
Type	Add	Div	Mul
Delay	1	5	4

Table 2. Code of the test program.

(I_1)	R3 = R1 + R2
(I_2)	R4 = R4 / R3
(I_3)	R3 = R1 * 42
(I_4)	R5 = R5 + R5
(I_5)	R3 = ...

Program Execution: Figure 2 traces the Scoreboard signals which are relevant for the deadlock. The execution starts in cycle $t = 1$. Everything works fine until the second addition (I_4) performs its notification in cycle 10. Since the division is still active, and since its second source depended on an earlier result of the adder, the corresponding valid flag F2.V2 is activated again. However, there is *no* data dependence between these two instructions, and this action is just a result of the generalized set condition in the notify mechanism.

In cycle 11, the multiplier tries to write back its result, but it has to pay attention to data dependences, especially to those with the division. Since the second source register F2.S2 of the divider equals the destination of the multiplier and since the valid flag F2.V2 is active, the write back test of the multiplier fails. Thus, the multiplier has to postpone its write back till the flag F2.V2 is cleared or till the divider requires different source registers.

The division instruction I_2 cannot modify these entries anymore, because it already read its arguments. Thus, the multiplication I_3 can only write (and finish) after a new division I_i $(i > 4)$ is issued to unit F2: $issue(i) < write(3)$.

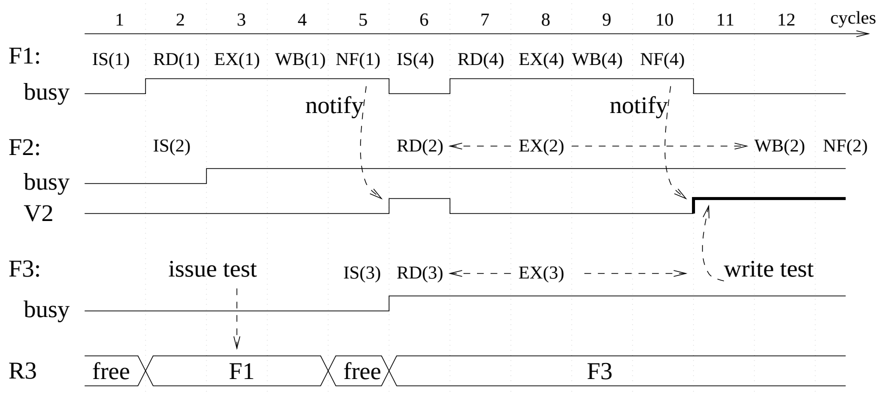

Fig. 2. Trace of the Scoreboard signals which are relevant to the deadlock. IS(i), RD(i), EX(i), WB(i) and NF(i) denote the issue, operand read, execute, write back and notification cycles of instruction I_i. The notify of cycle 10 raises flag $F2.V2$, although the divider has already read the corresponding operand. Thus, signal $F2.V2$ signals a false data dependence.

On the other hand, there exists a data dependence between the multiplication I_3 and the instruction I_5 to be issued next. Both have the same destination register. That leads to the contradiction

$$issue(5) < issue(i) < write(3) < issue(5)\,,$$

i.e., I_5 cannot be issued before the multiplication I_3 is finished (wrote its result) and vice versa. Thus, the Scoreboard ran into a deadlock and is therefore not correct.

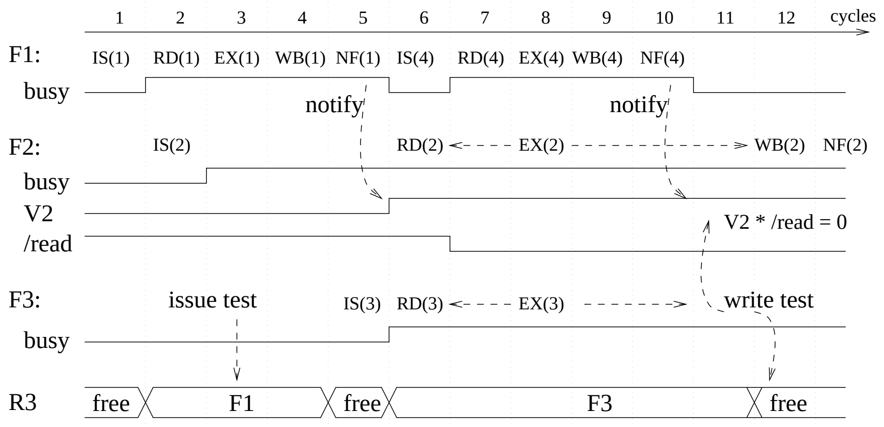

Fig. 3. Trace of the Scoreboard signals over the run time of the test program, in case that true valid flags are used. The notify of cycle 10 has no impact on the valid fag $F2.V2$. In cycle 11, the write back test holds, and the multiplication terminates in the next cycle.

Correction of the Scoreboard: The deadlock is due to the combination of the unusual valid flag and the generalized notification mechanism. Thus, it is an obvious modification to make the flags Vx true valid flags and to test for operands "valid and unread" ($F.Vx \wedge /F.read$) instead of $F.Vx$. In the test program, this modification avoids the deadlock (Fig. 3). The correctness of the modified Scoreboard mechanism is proven in Ref. 8.

References

1. G. S. Almasi and A. Gottlieb, *Highly Parallel Computing* (The Benjamin Cummings Publishing Company, Inc., 1994).
2. A. J. Bernstein, "Analysis of programs for parallel processing", *IEEE Trans. Electron. Comput.* **15** (1966) 757–762.
3. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach* 2nd edition (Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996).
4. R. M. Karp and R. E. Miller, "Parallel program schemata", *J. Computer & System Sciences* **3**, 2 (1969) 147–195.
5. R. M. Keller, "Look-ahead processors", *Computing Surveys* **7**, 4 (1975) 177–195.
6. P. M. Kogge, *The Architecture of Pipelined Computers* (McGraw-Hill, New York, 1981).
7. S. M. Mueller, "Complexity and correctness of computer architectures", in *Proc. 4th Workshop on Parallel Systems and Algorithms* (*PASA'96*), World Scientific Publishing, 1997, 125–146.
8. S. M. Mueller and W. J. Paul, "Making the original scoreboard mechanism deadlock free", in *Proc. 4th Israel Symposium on Theory of Computing and Systems* (*ISTCS*), IEEE Computer Society, 1996, 92–99.
9. D. A. Patterson and J. L. Hennessy, *The Hardware/Software Interface* (Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1994).
10. J. E. Thornton, *Design of a Computer: The Control Data 6600* (Scott Foresman, Glenview, Ill, 1970).
11. R. M. Tomasulo, "An efficient algorithm for exploring multiple arithmetic units", in *IBM J. Research and Development* **11**, 1 (1967) 25–33.
12. H. Zima, *Supercompilers for Parallel and Vector Computers* (ACM-Press, 1990).