Making the Original Scoreboard Mechanism Deadlock Free

Silvia M. Müller and Wolfgang J. Paul
Computer Science Department
University of Saarland
PF. 151150, 66041 Saarbrücken, Germany
{smueller, wjp}@cs.uni-sb.de

Abstract

Very recently, it was shown that the well known scoreboard as introduced in the CDC 6600 and described in current textbooks runs into deadlocks. However, it is one of the two major scheduling mechanisms used in todays microprocessors.

This paper presents a corrected version of the scoreboard and formally proves that mechanism to be correct and deadlock free.

1. Introduction

In architectures with multiple function units, a better utilization of the hardware resources can be achieved when executing instructions out-of-order, but that requires a scheduling mechanism. The *scoreboard* is one of the two main hardware scheduling mechanism. It was first introduced in the CDC 6600 (CYBER 70/74) [9, 10], but variations of this mechanism are still used in more current processors, like the Motorola MC 88100, or the Intel i860 [3].

In [7], it is proven that the scoreboard mechanism as presented in [10, 4] can run into deadlocks and is therefore incorrect. The counter example also indicates that the flaw in the original scoreboard is due to an unusual definition of valid flags in combination with a generalization of the test and set conditions in the write back mechanism.

In this paper, it is shown that switching to true valid flags results in a deadlock free and correct scheduling mechanism. Section 3 describes the modified scoreboard, and section 4 presents the formal correctness proof.

Unfortunately, in the standard literature on computer architecture, there seems to exist no sufficient criteria (with proof) for the correctness of a hardware scheduling mechanism. Therefore, section 2 provides a sufficient set of conditions for correctness. Its proof is straightforward and can be found in [8].

2. The Notion of Correctness

In order to present sufficient conditions for the correctness, it is helpful to have some notation on the structure of the instructions, and on the timing of their read/write accesses. For the timing, we argue on a cycle by cycle basis and therefore number the cycles by $t = 1, 2, \ldots$. $X[t]$ denotes the value of X at the *beginning* of cycle t.

2.1. Structure of the Instructions

Definition 2.1 *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,*
- *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 usual 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 modus and on the value of the interrupt masks. However, the definitions and lemmas presented here can easily be generalized for larger n and m.

Let $\mathcal{R}$ denote all storage components of architecture $\mathcal{A}$; we call the elements of $\mathcal{R}$ *registers*. $\mathcal{R}$ also comprises a dummy register which is referenced when ever an instruction reads less than two operands.

The set $\mathcal{O}$ denotes all the operations executable by architecture $\mathcal{A}$. Thus, for a $(2, 1)$ architecture $\mathcal{A}$ every instruction is of the following form:

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

with sources $S1(i), S2(i) \in \mathcal{R}$, with a destination $D(i) \in \mathcal{R}$, and an operation $op(i) \in \mathcal{O}$.

0-8186-7610-8/96 $5.00 © 1996 IEEE
Proceedings of ISTCS '96

2.2. Timing of the Accesses

During execution, the instructions have to pass several phases, namely: the issuing, the reading of the operands, 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 $I_1, I_2, \ldots, I_p$ be the sequence of instructions of a program P executed in-order by an architecture $\mathcal{A}$. This corresponds to the order of a sequential execution of P.

Let us assume that any of these four actions is performed in a single cycle, and that the actual execution of the operation $op(i)$ alone requires $\varepsilon(i) \geq 1$ cycles. For every instruction I_i in the out-of-order execution of program P,

- $\iota(i)$ denotes the issue cycle of I_i
- $\rho(i)$ denotes the cycles during which I_i reads its operands $S1(i), S2(i)$
- $\omega(i)$ denotes the cycle during which I_i writes its result back in $D(i)$.
- $\tau(i)$ denotes the cycle during which I_i terminates. After cycle $\tau(i)$, 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, i.e.,

$$\iota(i) < t \leq \tau(i).$$

2.3. Data Hazards

In the standard literature on computer architecture [4, 1, 6] the correctness of a scheduling mechanism is usually motivated by the fact that the following three types of data hazards (dependencies) are properly resolved and do not cause read/write conflicts:

Definition 2.2 (Write After Write) *In a program P, there is a* WAW dependency *WAW(i,j) between two instructions I_i and I_j with $i < j$ iff both instructions have the same destination:*

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

Under out-of-order execution, instructions I_i and I_j cause a WAW conflict *iff there exists a WAW dependency and the writes occur in reversed order, i.e., iff*

$$WAW(i,j) \wedge (\omega(i) \geq \omega(j)).$$

Definition 2.3 (Read After Write) *In a program P, there is an* RAW dependency *RAW(i,j) between two instructions I_i and I_j with $i < j$ iff I_i reads the register written by I_j:*

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

Under out-of-order execution, instructions I_i and I_j cause an RAW conflict *iff there exists an RAW dependency and the reads of I_j overtake the write of I_i, i.e., iff*

$$RAW(i,j) \wedge (\omega(i) \geq \rho(j)).$$

Definition 2.4 (Write After Read) *In a program P, there is a* WAR dependency *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:*

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

Under out-of-order execution, instructions I_i and I_j cause a WAR conflict *iff there exists a WAR dependency and the reads of I_i are overtaken by the write of I_j, i.e., iff*

$$WAR(i,j) \wedge (\rho(i) \geq \omega(j)).$$

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

A Sufficient Condition Just avoiding data and structural hazards as suggested in standard literature on computer architecture is not sufficient. In addition to this condition, which is modeled after the well known Bernstein condition [2, 5, 11] from the theory of parallelizing compilers, it must be guaranteed that exactly the same instructions are executed and that the scheduling mechanism does not run into deadlocks. Thus,

Definition 2.5 *Let $\mathcal{A}$ be a (2,1) architecture. The scheduling mechanism of $\mathcal{A}$ is correct, if for any program P of $\mathcal{A}$*

1. *in the out-of-order execution of P, the same instructions are executed as in its in order execution,*
2. (data hazards) *WAW, WAR and RAW conflicts are properly resolved, and*
3. (structural hazards) *for any cycle t, no more read/write accesses and operations are scheduled than the architecture can perform during that cycle.*

Avoiding Deadlocks A common type of scheduling mechanisms 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 if the mechanism does not run into a deadlock. Thus, the following lemma [8] provides a sufficient condition for such a scheduling mechanism to be deadlock free and to satisfy part (1) of definition 2.5. That can easily be proven by induction on the index of the instructions.

Lemma 2.1 (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:*

$$T(i) \;=\; \begin{cases} 0 & ; i = 1 \\ \max\{\tau(j) \mid j = 1, \ldots, i-1\} & ; i > 1 \end{cases}$$

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

$$\tau(i) \;\leq\; T(i) + \alpha + \varepsilon(i)$$

then I_i terminates, and the scheduling mechanism is deadlock free.

3. The Fixed Scoreboard

This section describes a variant of the scoreboard which uses true valid flags as suggested in [7]. For the rest of the paper, we refer to this scheduling mechanism as the scoreboard.

3.1. Basics of the Architecture

Here, the scoreboard is applied to a general (2,1) architecture $\mathcal{A}$ with a limited number f of function units $\mathcal{F} = \{F_1, \ldots, F_f\}$, but the number of read and write ports per register (memory) of $\mathcal{R}$ is unbounded. This generalization is not crucial for the correctness, because in a given cycle, the scoreboard mechanism schedules at most one write access or some read accesses per storage component.

For any operation $op \in \mathcal{O}$ of architecture $\mathcal{A}$

$$\mathcal{F}_{op} = \{F \in \mathcal{F} \mid F \text{ can perform } op\}$$

denotes the set of function units capable of executing operation op. During issue, an instruction I_i is assigned to a free function unit from the set $\mathcal{F}_{op(i)}$. This unit is denoted as $F(i)$.

For a given program $P = I_1, \ldots I_p$, the in-order issuing implies that

$$\forall i = 1, \ldots, p-1 \quad : \quad \iota(i) < \iota(i+1).$$

3.2. Data Structures

The scoreboard (table 1) provides two data structures, one holds the status of the registers, the other holds the status of the function units $F \in \mathcal{F}$ and information on the instruction currently executed by F.

Register Status The structure Res (result) specifies the status of all registers $R \in \mathcal{R}$, where $Res.R$ denotes the function unit for which register R is currently reserved. Thus, its intended meaning is:

Invariant 1 *In cycle t, the entree $Res.R$ equals $r \neq 0$ iff there exists an instruction I_j with destination R, which is executed on unit $F = F(j)$, and which is active during cycle t but has not performed its write back yet:*

$$\begin{aligned} Res.R[t] = r \neq 0 \quad &\Longleftrightarrow \quad \exists j \text{ with } F(j) = F_r \\ &\wedge \quad D(j) = R \\ &\wedge \quad \iota(j) < t \leq \omega(j). \end{aligned}$$

Phase Flags of the Function Units In order to issue instructions, the scoreboard must know whether a function unit is busy or not. It therefore provides a busy flag for each unit F:

Invariant 2 *Unit F is busy in cycle t, i.e., its flag $F.busy$ is active in cycle t, iff there exists an instruction I_i executed on unit $F(i) = F$, and I_i is active during cycle t:*

$$\begin{aligned} F.busy[t] = 1 \quad &\Longleftrightarrow \quad \exists i \text{ with } F(i) = F \\ &\wedge \quad \iota(i) < t \leq \tau(i). \end{aligned}$$

After instruction issue, unit $F(i)$ performs the actions required by I_i. Since this happens under the control of the scoreboard, it must know the phase of the active instructions respectively of the corresponding function units. The scoreboard therefore provides the flags rd, ex and wb per unit F indicating that it has passed the corresponding phase:

Invariant 3 *The flag $F.rd$ ($F.ex$, $F.wb$) is inactive in cycle t, iff there exists an instruction I_i executed on unit $F(i) = F$, and I_i is active during cycle t but has not left its read (execute, write back) phase yet:*

$$\begin{aligned} F.rd[t] = 0 \quad &\Longleftrightarrow \quad \exists i \text{ with } F(i) = F \\ &\wedge \quad \iota(i) < t \leq \rho(i) \\ F.ex[t] = 0 \quad &\Longleftrightarrow \quad \exists i \text{ with } F(i) = F \\ &\wedge \quad \iota(i) < t \leq \rho(i) + \varepsilon(i) \\ F.wb[t] = 0 \quad &\Longleftrightarrow \quad \exists i \text{ with } F(i) = F \\ &\wedge \quad \iota(i) < t \leq \omega(i). \end{aligned}$$

The Instruction Information The scoreboard also provides entrees for the operation (F.op), for the destination register (F.D) and for both source registers (F.S1, F.S2) of the current instruction I of unit F. The two valid flags F.Vx ($x = 1, 2$) are used to check for current data; during instruction execution, they are supposed to switch from 0 to 1 when the data become valid and then remain active till a new instruction is issued to unit F. Thus:

Invariant 4 *For $x = 1, 2$, the valid flag $F.Vx$ is inactive in cycle t iff there exist instructions I_i and I_j executed on units $F(i) = F$ respectively $F(j) = F_r$ such that the source register $Sx(i)$ was reserved by unit F_r during the issue cycle*

Unit	Phase				Instruction Information							
F	busy	rd	ex	wb	op	D	S1	P1	V1	S2	P2	V2
1	1	0	0	0	+	2	1	f	0	3	0	1
:												
f	1	1	0	0	*	1	..	..	..	..	..	..

Registers	R_1	R_2	R_3	...	$R_{\|\mathcal{R}\|}$
Res	f	1	0	...	0

Table 1. Data structure of a Scoreboard. Possible entrees just after instruction I: $R_2 = R_1 + R_3$ got issued which has to wait for the result of unit F_f.

of I_i, that both instructions are issued before cycle t, and that instruction I_j is still active:

$$\begin{aligned} F.Vx[t] = 0 \iff & \exists\, j < i\ \exists\, r \neq 0 \text{ with} \\ & F(i) = F \ \wedge\ F(j) = F_r \\ \wedge\ & Res.Sx(i)[\iota(i)] = r \\ \wedge\ & \iota(j) < \iota(i) < t \leq \tau(j). \end{aligned}$$

Thus, there exists an RAW(j,i) dependency between instructions I_i and I_j.

In case unit F is waiting for its source $F.Sx$ to become valid, the entree F.Px $\in \{0, \ldots, f\}$ specifies the function unit which is going to *produce* the required operand F.Sx. The value 0 indicates that the operand is already valid. Thus:

Invariant 5 *For $x = 1, 2$, the entree $F.Px$ equals $r \neq 0$ in cycle t iff there exists an instruction I_i executed on unit $F(i) = F$ such that its source register $Sx(i)$ was reserved by unit F_r during the issue cycle of I_i, and that I_i was issued before cycle t but has not left its read phase yet:*

$$\begin{aligned} F.Px[t] = r \neq 0 \iff & \exists\, i \text{ with } (F(i) = F) \\ \wedge\ & (Res.Sx(i)[\iota(i)] = r) \\ \wedge\ & \iota(i) < t \leq \tau(i). \end{aligned}$$

3.3. The Bookkeeping

Initialization The scoreboard requires some initialization. On power up, the valid flags F.Vx and the phase flags F.rd, F.ex and F.wb of all function units F are set to one. All the other entrees of the scoreboard are set to zero.

Initializing the flags in this way simplifies the invariants and the correctness proof, because after power up, any unit F is idle but looks like it terminated an instruction in a previous cycle. Thus, the first cycle after power up is not a special case for the scoreboard.

The scoreboard issues in-order the instructions I to an adequate function unit, but only one at a time. For each function unit F, the scoreboard also performs the bookkeeping and governs the resources according to rules which can be summarized as follows:

- The next instruction I is issued as soon as its destination and a function unit capable of executing I become available.
- After issuing the instruction, the reading of the source registers is postponed till both registers hold the current value, i.e., till both sources are valid.
- After the function unit ran to completion, the write back is postponed till no function unit with source register $R = F.D$ requires the old value any longer. Register R is released.
- After writing the result, all function units with a source depending on a result of unit F are notified. Their corresponding valid flag is set. The unit F is released.

We now describe the bookkeeping in more detail. For actions A and B, `$A \mid B$' denotes that they are executed in the same cycle; `$A\,;\,B$' denotes that B is executed a cycle after A.

Instruction Issue Let I_i be the instruction to be issued next. It is issued as soon as the destination register $D(i)$ and a function unit F capable of executing operation $op(i)$ become available.

The scoreboard then reserves the destination register and such a free unit, which we denote as $F(i)$. Furthermore, it initializes the phase flags and the instruction information of unit $F(i)$. For the valid flag of source F(i).Sx it checks whether the corresponding register is currently reserved as destination or not (table 2).

while (I_i not issued yet) $\wedge$ (I_{i-1} issued)

if (Res.D(i) = 0) $\wedge$ ($\exists$ Fu $\in F_{op(i)}$: Fu.busy=0)
then issue:
{ Let F(i) $\in F_{op(i)}$ with F(i).busy=0
$\forall x = 1,2$ do
{ F(i).Sx := Sx(i) | F(i).Px := Res.Sx(i) |
F(i).Vx := $\begin{cases} 1 & \text{if Res.Sx(i)} = 0 \\ 0 & \text{otherwise} \end{cases}$ |
}
F(i).rd := F(i).ex := F(i).wb := 0 |
F(i).busy := 1 | F(i).op := op(i) |
F(i).D := D(i) | Res.D(i) := F(i);
}

Table 2. Bookkeeping of the issue phase

Read Operands One cycle after issuing an instruction to unit F, that unit tries to read its arguments, as long as its flag $F.rd$ is inactive. It checks whether the registers to be read are up-to-date, i.e., whether its valid flags $F.Vx$ are set. If so, the registers are read and the flags $F.Px$ are cleared. After fetching its operands, F no longer waits for the result of the units $F.Px$ anyway (table 3).

while (F.rd = 0)
if (F.V1 $\wedge$ F.V2) then : { read operands | F.P1 := F.P2 := 0 | F.rd := 1; }

Table 3. Bookkeeping of the read phase

Execute Phase After reading the operands, unit F remains in the execute phase till the computation run to completion and then switches to the write back phase (table 4).

while (F.rd $\wedge$ / F.ex)
F.ex := ready flag of F;

Table 4. Bookkeeping of the execute phase

Write Back Phase In this phase, unit F tries to write its result. In order to avoid WAR conflicts, it must postpone the write back, as long as there is another function unit which has register $R = F.D$ as a source and still has to read the old (valid) value of R. While the unit F writes its result into register R, the scoreboard gives this register free (table 5).

while (F.ex $\wedge$ / F.wb)
if $\nexists$ Fu $\neq$ F, $\nexists x = 1,2$: ((Fu.Sx = F.D) $\wedge$ (Fu.Vx=1) $\wedge$ (Fu.rd=0)) then : { write result to F.D | F.wb := 1 | Res.F.D :=0; }

Table 5. Bookkeeping of the write back phase

Termination Phase In the last cycle of the execution of an instruction, 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 (table 6).

while (F.wb $\wedge$ F.busy)
$\forall$ Fu $\neq$ F, $\forall x = 1,2$ do : { if (Fu.Px = F) then Fu.Vx := 1 | } | F.busy := 0;

Table 6. Bookkeeping of the notify phase

4. Correctness of the Scoreboard

Here, it is proven that the scoreboard described above is a correct scheduling mechanism. According to definition 2.5 it suffices to show that the scoreboard avoids structural and data hazards, and that it is deadlock free. For this proof, we heavily build on the invariants (1) to (5). Inspection of the initialization and of the bookkeeping shows that the scoreboard satisfies those invariants.

4.1. Avoiding Structural Hazards

Lemma 4.1 *The scoreboard avoids structural hazards.*

Proof: Under the assumption that the architecture has sufficient read/write ports per storage component, structural hazards can only arise in the function units. According to invariant (2), unit $F(i)$ is reserved for instruction I_i as long as I_i is active. Since a new instruction I_j can only be issued if an adequate function unit is available, the allocation scheme of the scoreboard avoids structural hazards □

4.2. Resolving Data Hazards

Lemma 4.2 *The scoreboard avoids WAW conflicts.*

Proof: Let there be a dependency WAW(j,i) between I_j and I_i, and let I_j be executed on unit $F(j) = F_r$. Due to in order issuing, I_j with $j < i$ is issued before I_i: $\iota(j) < \iota(i)$.

According to invariant (1) of the Res data structure, register $D(j)$ is reserved for unit F_r from one cycle after I_j got issued till I_j finished its write back, i.e.:

$$\iota(j) < t \leq \omega(j) \quad \Rightarrow \quad Res.D(j)[t] = r.$$

The bookkeeping of the issue phase requires that register $D(i)$ is free when I_i gets issued, i.e.,

$$Res.D(i)[\iota(i)] = 0.$$

Due to the WAW(j,i) dependency, both instructions have the same destination $D(j) = D(i)$. Thus, the issuing of I_i is postponed till I_j finished the write back. Since the write back occurs after issuing it follows that

$$\text{(A)} \qquad \omega(i) > \iota(i) \geq \omega(j) + 1 = \tau(j),$$

and that the scoreboard avoids WAW conflicts. □

Lemma 4.3 *The scoreboard avoids RAW conflicts.*

Proof: Let there be a dependency RAW(j,i) between I_j and I_i. Thus, $j < i$ and $D(j) = Sx(i)$ for an $x \in \{1, 2\}$.

1) In case I_i is issued after the write back of I_j, i.e., $\omega(j) < \iota(i)$, the RAW(j,i) hazard is obviously resolved because an instruction only reads its arguments after it got issued: $\rho(i) > \iota(i) > \omega(j)$.

2) Now, $\omega(j) \geq \iota(i) > \iota(j)$.
Let I_j be executed on unit $F(j) = F_r$. According to invariant (1), register $Sx(i)$ is reserved for unit F_r during the issuing of I_i, i.e.,

$$Res.Sx(i)[\iota(i)] = r.$$

and then, the invariant (4) of the valid flags indicates that the valid flag $F(i).Vx$ becomes inactive after the issuing of I_i and remains inactive till I_j terminated:

$$\text{(B)} \quad \iota(j) < \iota(i) < t \leq \tau(j) \quad \Rightarrow \quad F(i).Vx[t] = 0.$$

Since the operands are read as soon as both valid flags are active, it follows that

$$\rho(i) > \tau(i) = \omega(j) + 1.$$

Thus, the scoreboard resolves the RAW(j,i) hazard properly. □

Lemma 4.4 *The scoreboard avoids WAR conflicts.*

Proof: Let there be a dependency WAR(i,k) between I_i and I_k. Then, $i < k$ and $D(k) = Sx(i)$ for an $x \in \{1, 2\}$, and it is to show that I_i reads its arguments before I_k writes its result: $\omega(k) > \rho(i)$.

1) In case instruction I_i terminates before the write back of I_k, i.e., $\omega(k) > \tau(i) > \rho(i)$, the WAR(i,k) conflict obviously resolved.

2) Thus, let now be $\tau(i) \geq \omega(k) > \iota(k)$.
According to the invariant (3), it suffices to show that the flag $F(i).rd$ is active during the write back of I_k:

$$F(i).rd[\omega(k)] = 1,$$

respectively, that during that cycle the destination of unit $F(k)$ equals the source Sx of unit $F(i)$, and that the valid flag F(i).Vx is active:

$$\begin{array}{lrcl} (a) & F(i).Sx[\omega(k)] & = & F(k).D[\omega(k)] \\ (b) & F(i).Vx[\omega(k)] & = & 1. \end{array}$$

Since in cycle $\omega(k)$ both instructions are active, and since $D(k) = Sx(i)$, condition (a) is satisfied.

For condition (b), we distinguish two cases, namely that the valid flag Vx of unit F(i) is active immediately after the issue of I_i or not.

- $F(i).Vx[\iota(i) + 1] = 1$. From the invariant (4) of the valid flag it follows that the flag is active while instruction I_i is active, i.e., during cycles t with $\iota(i) < t \leq \tau(i)$, and especially during cycle $t = \omega(k)$.

- $F(i).Vx[\iota(i) + 1] = 0$. According to invariant (4) there exists I_j which is executed on unit $F(j) = F_r$, which is active during cycle $\iota(i)$ and causes register Sx(i) to be reserved for unit F_r during that cycle:

  $$(c) \quad Res.Sx(i)[\iota(i)] = r.$$

  According to invariant (1) of the data structure Res, the destination of I_j equals the source of I_i: $D(j) = Sx(i) = D(k)$, and there exists a WAW(j,k) dependency. Due to condition A (proof 4.2), instruction I_j is not active after the issuing of I_k, and therefore

  $$\tau(j) \leq \iota(k) < \omega(k) \leq \tau(i).$$

  Due to equation (c) and invariant (4), the valid flag Vx of unit F(i) becomes active after the termination of I_j and remains active till the termination of I_i

  $$\tau(j) < t \leq \tau(i) \Rightarrow F(i).Vx[t] = 1$$

  and especially during cycle $t = \omega(k)$. Thus, condition (b) is satisfied as well. □

4.3. Avoiding Deadlocks

In order to prove the (modified) scoreboard to be deadlock free, we use the sufficient condition of lemma 2.1. $T(i)$ still denotes the cycle after which all instructions $I_1, \ldots, I_{i-1}$ have terminated.

Lemma 4.5 *When delivering the instructions I_i to the scoreboard in program order, then I_i terminates in cycle $\tau(i) \leq T(i) + \varepsilon(i) + 4$.*

Proof: In order to prove this lemma, it is first shown that the timing of I_i satisfies the following conditions:

$$\begin{aligned} \iota(i) &\leq T(i) + 1 && (1)\\ \rho(i) &\leq T(i) + 2 && (2)\\ \omega(i) &\leq T(i) + \varepsilon(i) + 3. && (3) \end{aligned}$$

(1) Issue Time. Since the instructions are issued in-order and only one at a time, I_i can be issued one cycle after I_{i-1}, at the earliest:

$$\iota(i) \geq \iota(i-1).$$

The bookkeeping requires that at issue time unit F(i) and the destination register D(i) are available:

$$F(i).busy[\iota(i)] = 0 \ \wedge \ Res.D(i)[\iota(i)] = 0.$$

According to invariants (2) and (1), I_i must wait till all preceding instructions executed by the same unit have terminated

$$\iota(i) \geq 1 + \max\{\tau(k) \mid k < i \ \wedge \ F(k) = F(i)\}$$

and till all instructions I_j causing a WAW(j,i) dependency have finished the write back

$$\iota(i) \geq 1 + \max\{\omega(j) \mid j < i \ \wedge \ WAW(j,i)\}.$$

Instruction I_i is issued as soon as all three conditions are fulfilled, i.e.,

$$\begin{aligned} \iota(i) &= 1 + \max\{\iota(i-1), \omega(j), \tau(k) \mid j,k < i \\ &\qquad \wedge WAW(j,i) \ \wedge \ F(k) = F(i)\} \\ &\leq T(i) + 1. \end{aligned}$$

(2) Read Phase. Before its read access, function unit F(i) only checks its own valid flags F(i).Vx. These flags have been initialized during instruction issue and will be updated by another function unit during its notify step, if necessary. Since invariant (4) implies that this modification can only be performed by a preceding instruction I_j with a RAW(j,i), a direct impact of a later instruction ($j > i$) on the outcome of the read test is impossible.

Since RAW dependencies are properly resolved, and since the notification is performed in the termination cycle, it holds $\rho(i) > \tau(j) > \omega(j)$. The function unit reads its arguments after instruction issue as soon as the test condition is true. Thus:

$$\begin{aligned} \rho(i) &= \max\{\iota(i), \tau(j) \mid j < i \ \wedge \ RAW(j,i)\} + 1 \\ &\leq T(i) + 2. \end{aligned}$$

(3) Write Phase. For the write back test, we distinguish two types of function units Fu, namely those with an active respectively inactive flag Fu.busy. In case this flag is inactive, unit Fu is either idle or just tries to issue an instruction. Then, its flag Fu.rd is still set — either by initialization or by a previous instruction — and Fu flunks the test

$$\text{(C)} \qquad (\text{Fu.Sx} = \text{F(i).D}) \wedge \text{Fu.Vx} \wedge (\text{Fu.rd} = 0)$$

and does not delay the write back of I_i.

Let the flag Fu.busy be active in cycle t during which instruction I_i performs its write back test, and let Fu just process instruction I_j. For the write back of I_i, we are now only interested in cycles during which I_j is active:

$$\iota(j) < t \leq \tau(j).$$

There either exists a WAR(j,i) or a RAW(i,j) dependency or the sources of I_j are different from the destination of I_i.

- In the latter case, $Sx(j) \neq D(i)$, Fu flunks test (C), and therefore does not delay the write back of instruction I_i.
- $RAW(i,j)$. Due to this dependency, I_j is issued after I_i, but before the termination of I_i, because it is active during the write test of I_i:

  $$\iota(i) < \iota(j) < t \leq \tau(i).$$

  According to condition B (proof 4.3), the valid flag Vx of unit F(j) is active for all those cycles t:

  $$\iota(i) < \iota(j) < t \leq \tau(i) \quad \Rightarrow \quad F(j).Vx[t] = 0$$

  and Fu does not delay the write back of I_i.
- $WAR(j,i)$. Due to this dependency, I_j is issued after I_i and:

  $$\iota(j) < \iota(i) < t \leq \tau(j).$$

  During this time frame, Fu delays the write back of I_i at most till its flag $Fu.rd$ becomes active. According to the invariants (2) and (3), unit F(j) still processes instruction I_j one cycle after its read access, and its flag $F(j).rd$ is active then:

  $$F(j).rd[\rho(j) + 1] = 1.$$

Altogether, it holds

$$\begin{aligned}\omega(i) &\leq 1 + \max\{\rho(i) + \varepsilon(i), \rho(j) \mid j < i \\ &\qquad \wedge\, WAR(j,i)\} \\ &\leq T(i) + 3 + \varepsilon(i).\end{aligned}$$

Termination is performed one cycle after write back, thus

$$\tau(i) = \omega(i) + 1 \leq T(i) + \varepsilon(i) + 4,$$

and the scoreboard is deadlock free. □

5. Conclusion and Prospects

A deadlock free version of a scoreboard has been derived and its correctness has been proven formally. Now, the next step must be to provide a hardware realization of this scheduling mechanism and a description which tells how to adapt it to a given architecture. This also brings up the question, how this mechanism impacts the hardware cost, the cycle time and the performance of a system with multiple functional units.

References

[1] G. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjamin Cummings Publishing Company, Inc., 1994.

[2] A. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. Electronic Computers*, 15:757–762, 1966.

[3] A. Bode, editor. *RISC-Architekturen*. BI-Wissenschaftsverlag, 1988.

[4] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 1990.

[5] R. Karp and R. Miller. Parallel program schemata. *J. Computer & System Sciences*, 3 (2):147–195, 1969.

[6] P. Kogge. *The Architecture of Pipelined Computers*. McGraw–Hill, New York, 1981.

[7] S. Müller. Complexity and correctness of computer architectures. In *Proc. 4th Workshop on Parallel Systems and Algoriths (PASA)*, 1996.

[8] S. Müller and W. Paul. On the correctness of hardware scheduling mechanisms for out-of-order execution. Technical report, CS Department, University of Saarland, Germany, http://www-wjp.cs.uni-sb.de/~smueller, 1996.

[9] J. Thornton. Parallel operation in the Control Data 6600. In *Proc. 26th AFIPS Conference*, volume 2, pages 33–40, 1964.

[10] J. Thornton. *Design of a Computer: The Control Data 6600*. Scott Foresman, Glenview, Ill, 1970.

[11] H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM-Press, 1990.