COMPLEXITY AND CORRECTNESS OF COMPUTER ARCHITECTURES

S.M. MÜLLER
Computer Science Department, University of Saarland, PF. 151150
66041 Saarbruecken, Germany
email: smueller@cs.uni-sb.de

This paper studies the complexity of pipelining and precise nested interrupt handling, and formally quantifies their impact on the cost and cycle time of a RISC architecture. As for instance, it is shown that a standard 5-stage pipeline improves the performance of a fixed-point RISC core roughly by a factor of 2. another 10 to 26%. The paper also emphasizes the correctness of designs. Although formal correctness proofs are largely ignored in computer architecture, they can be instrumental in developing correct designs. Here it is proven that a major hardware scheduling mechanism named scoreboard can run into a deadlock. However, a small but subtile change is enough to make it deadlock free and correct.

1 Introduction

Studies in computer architecture derive solutions for standard problems, like the precise interrupt handling, and analyze their impact on the quality of a design. As far as complexity is concerned, i.e., the impact on the quality of a design, one would like to determine time/cost trade-offs, but it is a widely held opinion that those analyses are impracticable. They require the specification of a whole design, and usually involve chip simulation or prototyping.

Thus, most extensive, quantitative analyses are concerned with the cycle count, whereas cycle time and hardware cost are treated on a more qualitative level, i.e., as pitfalls and fallacies [1]. However, our formal architecture model [2] permits to evaluate the cost effectiveness of computer architectures and to quantify cost/time trade-offs with very reasonable effort. This paper covers some of the results gained with that model.

Besides quality aspects, studies in computer architecture should also cover the correctness of the designs. Formal correctness proofs increase the confidence in a design, they can be instrumental in simplifying the verification of a design, and sometimes they even achieve surprising results (section 5). Nevertheless, the correctness aspect is largely ignored in computer architecture. That needs to be changed, especially because up-to-date computer systems become increasingly complicated.

1.1 Outline of the Paper

After a brief discussion of the formal hardware model, sections 2 and 3 present our results concerning the complexity of forwarding and interrupt handling in pipelined designs. In this context, it is explained what is necessary to support precise nested interrupt processing.

The second part of the paper discusses the bug in a major hardware scheduling mechanisms named scoreboard. The scoreboard was first introduced in the CDC 6600 [3,1], but variations of it are still used in current processors. After a detailed description of the original scoreboard mechanisms (section 4), we provide a 5-line code which drives the said scoreboard in a deadlock (section 5).

1.2 The Formal Architecture Model

The major difference from other approaches is our hardware model which permits to interpret an architecture A as a whole design and to derive its cost C_A (in gate equivalents [g]) and its cycle time t_A(in gate delays [d]) with very reasonable effort. Then, architectures can also be *formally* evaluated and compared based upon their cost effectiveness, where the quality of an architecture A is defined as the quotient of its performance on a benchmark Be and of its hardware cost. The performance is the reciprocal of the benchmark's run time

$$T_A(Be) = t_A \cdot CC_A(Be) \tag{1}$$

where $CC_A(Be)$ denotes its cycle count.

Cost and Cycle Time The hardware is constructed from *basic components*, like gates, flipflops, RAMs, and 3-state drivers. Their cost and propagation delay can be extracted from any design system. Table 1 lists these technology parameters for the Motorola technology [4] used in this paper.

The cost C_A of the whole hardware is the cumulative cost of all basic components in the data paths and in the control. The cost of the off-chip main memory is usually ignored. The cycle time t_A of A is the maximal delay of all paths through the hardware.

Control Unit The control is known to be the hard part of designs. Therefore, it is usually specified by a finite state diagram (FSD) [1] which is implemented as a Mealy or Moore automaton [5]. Based on those automata, our framework [2] provides cost and delay formulae for several variants of hardwired and programmed control. These formulae depend in a very simple way on a

Table 1: Cost [g] (gate equivalents) and delay [d] of the basic components

	Not	Nand Nor	And Or	Xor Xnor	Flip-flop	Mux	3-state Driver
cost	1	2	2	4	8	3	5
delay	1	1	2	2	4	2	2

	small on-chip RAM with $A \leq 64$ entrees of width n
cost	$2 \cdot (A+3) \cdot (n + \log^2 n)$
delay	$\log n + A/4$

few parameters, which can easily be read off from the underlying FSD and the instruction set.

Evaluation Since it is a messy and error prone task to evaluate all formulae for cost and delay by hand, we express them as C-routines. Then, a computer can keep track of cost and delay of the architectures. Our framework[2] provides a library of C-routines for the cost and delay of commonly used circuits, ranging from simple decoders and encoders over ALUs and control automata to whole processors.

2 The Impact of Pipelining

The pipelining concept is fairly well understood. Famous textbooks[6,1] describe in great length how to pipeline a RISC design and how to solve the major problems involved. The impact of pipelining on the cycle count of a design is also heavily studied, but quantitative results concerning cost/time trade-offs are still lacking. In his thesis[7], Knuth used our hardware model in order to close that gap. We now sketch that study, and present the major results.

2.1 Hardware Specification

As in the study of Hennessy and Patterson[1], the fixed-point core of the well known RISC architecture named *DLX* is mapped into the standard 5-stage pipeline comprising the stages – instruction fetch (IF) – instruction decode (ID) – execute (EX) – memory access (M) – write back (WB). Figure 1 depicts the pipelined data paths which resolve most data hazards by result forwarding.

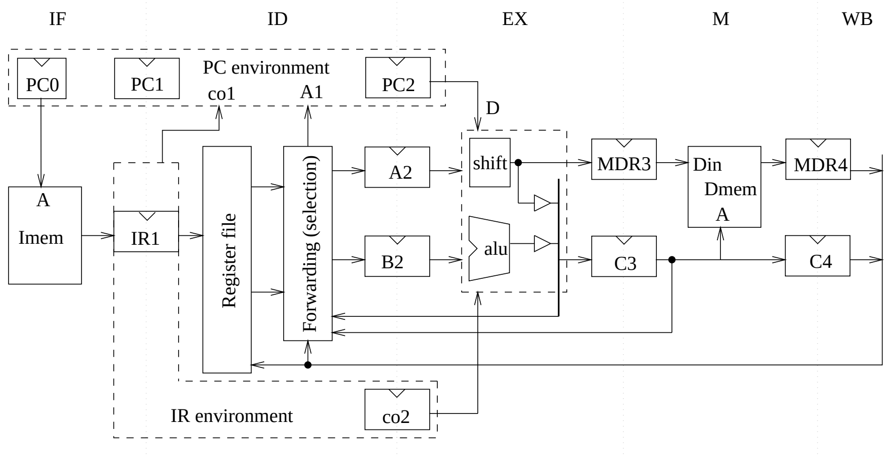

Figure 1: Block diagram of the DLX data paths with a separate instruction memory Imem and a data memory Dmem.

Result Forwarding Due to the interleaved execution, an instruction reads its operands before the preceeding instruction has written its results. Nevertheless, the pipelined design must also obey the data dependencies of the program; that causes severe problems. In the example of table 2, the decode of the second instruction must therefore be delayed until the first instruction has updated the register file. That causes three stall cycles.

Table 2: Scheduling of two dependent instructions

Instruction	Cycle					
R1 = R2 + R3	IF_1	ID_1	EX_1	M_1	WB_1	
R4 = R1 - 42		IF_2	$\cdots$	stall	$\cdots$	ID_2

However, inspection of the data paths (Figure 1) indicates that the result of the addition is already available at the end of stage EX. Thus, the three stall cycles can be avoided when forwarding the result directly to stage ID. The DLX design provides those bypasses for the stages EX, M and WB. That resolves the hazards caused by arithmetic operations. However, after a load instruction, upto two stall cycles can arise because the memory data can only be forwarded from the write back stage.

Details of the Data Paths In order to determine the cost and cycle time of the pipelined DLX design, all the environments need to be specified on gate level. However, it has turned out[7] that most circuits can directly be taken from the sequential DLX design, as specified in book[2]. Only the PC environment (Figure 2) encounters major changes. The pipelined version got an incrementer (Inc) and an adder (Add) on its own, so that the update of the PC and an ALU operation can be performed simultaneously.

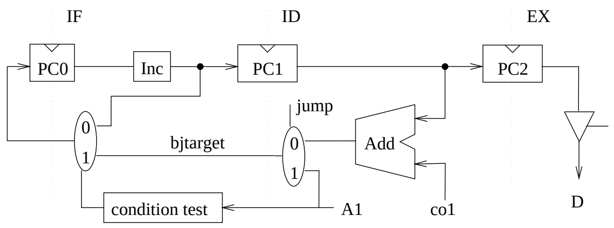

Figure 2: PC environment of the pipelined DLX design

Note, that the target address of a branch becomes valid just at the end of the decode stage. That is to late for next instruction fetch, causing a stall cycle after the branch.

2.2 Complexity Analysis

Cycle Count The literature [1] quantifies the impact of pipelining on the performance of the DLX design, but only based on the cycle count. Table 3 lists the CPI ration (cycles per instruction) of the two benchmarks GCC and TeX from the SPEC-89 suite, executed on a sequential DLX respectively on the pipelined DLX with result forwarding. It is assumed, that both designs have an average memory access time of 1.2 cycles per access.

Table 3: Average CPI ratio and speedup factor for the DLX designs

	absolute CPI ratio		relative
	sequential	pipelined	CPI
GCC	4.69	1.67	2.8
TeX	4.89	1.64	3.0

At best, the pipelined DLX can start one instruction every 1.2 cycles, but due to branch/jump and load stall cycles, it only achieves a CPI ratio of 1.66.

This ratio is about 3 times better than the one of the sequential DLX. It would be tempting to conclude that pipelining increases the performance of the DLX architecture by a factor of 3, but that is not the case.

Cost and Performance Table 4 lists the changes in the cost and cycle time of the DLX design caused by pipelining and result forwarding.

The pipelining itself has virtually no impact on the cycle time, but the result forwarding causes a 30% increase. Thus, despite a three times better CPI ratio (Table 4), the 5-stage pipeline with result forwarding only improves the performance of the DLX architecture by a factor of 2.3. The 30% increase in the cycle time hurts. In section 2.3, it is therefore analyzed, how that increase can be compensated for.

Table 4: Changes in cost and cycle time due to pipelining and result forwarding

	Δ cost [%]			Δt_{DLX}
	DP	CON	DLX	[%]
no forwarding	+30	−45	+20	−1
full forwarding	+32	−28	+26	+31

The pipelining requires an extended PC environment and some additional buffers (registers) to separate the pipeline stages. That increases the cost of the DLX data paths by roughly 30%, but the cost of the whole design is only increased by 20 - 26%, because the control becomes cheaper by one third.

The cost reduction in the control is somewhat surprising at the first sight. However, the pipelined control generates all the control signals which are required for the execution of an instruction at once, and let them trickle down the pipeline. That saves the sequencing logic of the control, and accounts for the 30% cost reduction.

2.3 Partial Forwarding

In the sequential DLX design, the critical path goes from the source registers A2, B2 and co2 through the ALU into the destination register C3. Inspection of the pipelined data paths (Figures 1 and 2) indicates that now the result of the ALU is also forwarded to the PC environment, where it is fed into the condition test.

In order to reduce the cycle time, Knuth suggested the following re-timing optimization which he calls *partial forwarding*. Results from the execute stage

(ALU) can only be forwarded to the operand fetch circuit but not to the PC environment. That splits the critical paths and brings its delay down to 1.03 times the cycle time of the sequential design. That is an improvement of roughly 26%, but partial forwarding also impacts the CPI ratio.

The condition test of a branch is performed in the decode stage. Thus, a branch which tests the result of the preceeding instruction must be delayed by one cycle, because the forwarding from the execute stage is cut off.

```
Rx = R1 + R2;
Branch (Rx = 0);
```

In order to keep the performance loss as small as possible, the compiler tries to insert an useful instruction between the computation of the condition and the branch, but that is not always possible. Especially on a workload with many branches like the GCC benchmark (22%), partial forwarding therefore increases the CPI ratio. Nevertheless, it achieves a speedup of 2.4 to 2.7 over the sequential DLX design. Thus, the re-timing improved the performance by another 10 to 26%.

3 Precise Interrupt Handling

Up-to-date computer systems provide fast I/O, virtual memory and support the full IEEE floating point standard. All that calls for a powerful interrupt mechanism, which is acknowledged to be a particularly hard part of machine design [1] (p. 214). Sofar, the literature only sketched hardware for handling *single* interrupts, but real systems require a more complex mechanism.

Interrupt handlers must be able to call or interrupt each other (*nesting*). Otherwise the system would for example be doomed when a print job switches to a missing memory page. Furthermore, the interrupt handling must be *precise*; after handling the interrupt (e.g., after loading the memory page), the program must be resumed at the instruction which caused the problem:

Definition 1 *Let $I_1, I_2, \ldots, I_p$ be the instruction sequence of a program P in absence of interrupts. When servicing an interrupt between I_{i-1} and I_i,* preciseness *requires that before starting the interrupt service routine (ISR), instructions $I_1, \ldots, I_{i-1}$ run to completion and that later instructions $(I_i, \ldots)$ did not change the state of the machine yet.*

Interrupts come in different flavors; that makes matters even worse. They either abort the interrupted program, or resume it in one of two ways:

- repeat the interrupted instruction I
- continue with the instruction which follows I in the non-interrupted execution of the program.

On an interrupt of type *repeat*, the ISR is invoked before the interrupted instruction I, whereas on an interrupt of type *continue*, it is invoked after I. Furthermore, some interrupts can be *masked* so that they can temporarily be ignored.

Before analyzing how such an interrupt mechanism interacts with the pipelined DLX design, we first describe the problems inherent in precise nested processing [2,8].

3.1 Precise Nested Processing

The hardware support for handling single interrupts is fairly obvious [1]. During a jump to the ISR, the machine masks the interrupts and saves the *status* into special purpose registers, so called *exception registers.* The status includes the interrupt cause, the interrupt masks, and the address (exception PC) at which the program is to be resumed. It then executes the code specific to the current interrupt. When leaving the ISR, the PC and the old masks are restored.

On the em nested processing of multiple interrupts, the ISR itself can be interrupted. Then, the exception registers are overwritten although their content is still required for resuming the program properly. In order to secure these data, the ISR first saves the exception registers on the *interrupt stack* and restores them before resuming the interrupted code. These two actions are not allowed to be interrupted.

Since some interrupts are non-maskable, protecting these *critical sections* becomes the hard part of precise interrupt handling. That puts constraints on the software, namely:

- The ISR must be programmed in such a way that non-maskable interrupts (e.g., illegal instruction) are avoided in the ISR, or at least in its critical sections.

- The code of the ISR and the interrupt stack must be held on a permanent memory page, i.e., a page which is never swapped out of main memory.

Based on this convention for the ISR, it was possible to derive a provably correct interrupt mechanism [2] (software + hardware) for the sequential DLX design. Here, correctness means that the mechanism satisfies the following conditions:

1. All the interrupts are detected and assigned to the proper instruction.

2. Finite response time: The ISR is invoked a few cycles after the detection of an interrupt.

3. Priority: Once an interrupt received service, it can only be interrupted by an unmasked event of higher priority.

4. The interrupts are processed in a precise manner (Definition 1).

5. Completeness: Every interrupt which is unmasked long enough will be serviced.

Major constraints for the interrupt hardware arise from the conditions 1 and 5, the proper catching of the interrupts and their precise processing. The other properties are largely guaranteed by the ISR.

3.2 The Impact of Pipelining

When adapting the precise interrupt mechanism to the pipelined DLX design [7,8], the software can be re-used but the interrupt hardware becomes more complicated, especially the hardware related to the detection of the interrupts and the precise processing.

Detection of Interrupts In a pipelined design, several instructions are processed at a time. Although the instructions are in different stages, their interrupt events can occur simultaneously, or even worse, they can overtake one another. Thus, besides catching the events in the cause register, they need to be assigned to the proper instruction.

For that purpose, the cause register is pipelined. Register CAi collects the events which an instruction triggered up to pipeline stage i (Figure 3). The interrupt test is performed in the last stage, i.e., the causes in CA4 are combined with the masks and then checked for an interrupt as in the sequential case. Register CA4 takes the place of the original cause register.

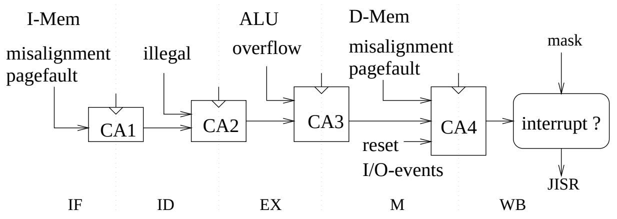

Figure 3: Circuit for collecting interrupt causes

Data Protection The pipelined DLX initiates the jump to the ISR in the write back stage via signal JISR. The instructions I_f, I_d, I_e and I_m which are then processed in the stages IF to M definitely follow the interrupted instruction I_w and must therefore be prevented from updating the machine. However, these instructions are already in the pipeline several cycles before JISR, and they will stay there upto 4 cycles beyond JISR (Figure 4). That makes data protection even worse. Furthermore, on an interrupt of type repeat, I_w also becomes an *invalid* instruction, and needs to be canceled.

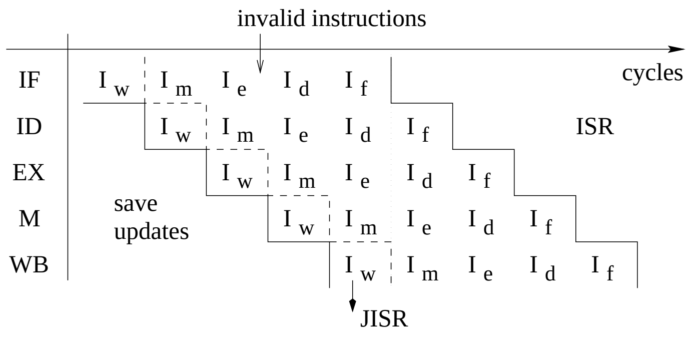

Figure 4: Pipelined execution of the instruction sequence I_w, I_m, I_e, I_d, I_f, in case instruction I_w triggers an interrupt.

In order to keep the canceling easy, signal JISR usually forces NOPs into the stages ID to WB[1]. Then, after the jump to the ISR, the invalid instructions trickle down the pipeline as NOPs, and therefore do not update the machine anymore. However, actions which the invalid instructions performed before the jump to the ISR still need to be rolled back. How that can be accomplished depends on the type of the storage and on the stage in which it is updated.

- *The storage of stage WB*, like the general purpose register file, only encounter an illegal update during the write back of I_w, in case that this instruction triggers a repeat interrupt. Therefore, the write back is disabled on a repeat interrupt only.
- *Registers of early stages*, like the PC, are already updated by illegal instructions before the detection of the interrupt, but on an active signal JISR, these actions must be rolled back. For that purpose, the old values of those register are buffered up to stage WB.
- The memory is updated one cycle before the instruction is checked for interrupts. Since the backup of the memory would be to expensive,

stores must be delayed till they are safe. Here, only the updates of the instructions I_w and I_m are critical. The write of I_m can be canceled by the signal JISR, and I_w can only cause an illegal update if it is a store which triggers a repeat interrupt. In the DLX, these interrupts are signaled one cycle ahead of JISR, so that all illegal memory write access can be canceled on time.

Complexity Analysis Table 5 lists the changes in the cost and cycle time of the DLX designs caused by the interrupt hardware; the cycle times remain the same.

Table 5: Changes in cost and cycle time due to precise nested interrupt processing

	Δ cost [%]			Δt_{DLX}
	DP	CON	DLX	[%]
sequential	+21	+96	+28	0
pipelined	+37	+200	+40	0

Interrupt handling makes the DLX design only 30 to 40% more expensive, but a huge cost increase is encountered in the control. In the sequential DLX, the cost of the control is doubled, and in the pipelined design, it is even tripled. That corresponds to the intuition, that interrupt handling is the hard part of design, and that pipelining makes it even much harder.

4 Out-Of-Order Execution

The current processors[1] are far more complex; they comprise multiple function units which can work in parallel. Since the latency of the function units varies by a lot, designs like PowerPC, Pentium-Pro, and MIPS R10000, allow instructions to overtake each other, in order to achieve a better hardware utilization. However, the data dependencies of the program must still be obeyed.

Intuitively, the *out-of-order execution* of a program P with instruction sequence $I_1, \ldots, I_p$ is considered to be correct if the following holds:

- Exactly the same instructions $I_1, \ldots, I_p$ are executed as in the sequential (in-order) execution.
- Each instruction I_i reads the same operands and writes the same result as in sequential execution.

- Write accesses to the same storage component, i.e., registers and memory (cells), occur in the same order as in sequential execution.

Thus, a correct out-of-order execution requires a *scheduling mechanism* which determines when to start a particular instruction, when to read its operands, and when to write its result. The Scoreboard [3,1] is one of the two main scheduling mechanisms used.

For the rest of the paper, we describe the original scoreboard as introduced in the CDC 6600 in detail and prove that it is incorrect because it can run into deadlocks. In order to discuss that flaw, we first introduce some notation on the structure of the instructions and on the timing of their read and write accesses.

4.1 Basics of the Architecture

Definition 2 *For $n, m \geq 1$, let a machine be 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.*

For simplicity's sake, the scheduling mechanism is applied to an architecture $\mathcal{A}$ of type $(2, 1)$, but it can easily be generalized for larger n and m.

Structure of the Instructions Let $\mathcal{K}$ denote the set of all storage components (registers) of architecture $\mathcal{A}$, including a dummy register which is referenced when ever an instruction reads less than two operands. Let set $\mathcal{O}$ denote all the operations executable by architecture $\mathcal{A}$. Thus, every instruction I_i of a program P executed by $\mathcal{A}$ is of form

$$I_i \;:\quad D(i) \;=\; S1(i)\; op(i)\; S2(i) \tag{2}$$

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

Hardware Resources With respect to the hardware resources, we assume that architecture $\mathcal{A}$ comprises a limited number f of function units

$$\mathcal{F} = \{F_1, \ldots, F_f\}. \tag{3}$$

but that the number of read and write ports per storage component of $\mathcal{K}$ is unbounded. This generalization is not crucial for the correctness, because it can be shown [9] that 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 $\mathcal{A}$, the set F_{op} denotes all function units capable of executing operation op, i.e.,

$$\mathcal{F}_{op} = \{F \in \mathcal{F} \mid F \text{ can perform } op\}. \tag{4}$$

Timing of the Accesses For the correctness of a scheduling mechanism, the timing of three actions in the instruction execution are essential, namely: the issuing, the reading of the operands, and the writing of the result. In order to ague on a cycle by cycle basis, the machine cycles are numbered by $t = 1, 2, \ldots$.

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

- $\iota(i)$ denotes the cycle during which I_i is issued
- $\rho(i)$ 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 $D(i)$.

During a cycle t, an instruction I_i is called *active*, iff it got issued before t and did not write back its result yet, i.e.,

$$\iota(i) < t \leq \omega(i). \tag{5}$$

4.2 The Scoreboard Mechanism

The original *scoreboard*[3,1] 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 (Table 6). 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 $R \in \mathcal{K}$, where

$Res.R = 0$ denotes that no function unit F has an active instruction I with destination R,

$Res.R = r \neq 0$ denotes that register R is currently reserved by unit F_r which is going to produce a result for this register.

Table 6: Data structure of a Scoreboard. Possible entries just after the issue of instruction I: $R_2 = R_1 + R_3$ which has to wait for the result of unit F_f.

Unit	Phase				op	D	Sources					
F	busy	rd	ex	wb			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	$\cdots$	$R_{\vert\mathcal{K}\vert}$
Res	f	1	0	$\cdots$	0

Phase Flags of the Function Units Every instruction I_i of a program P must go through several phases, i.e., issuing, reading operands, execution of the operation, write back of the result. During instruction issue, I_i is assigned to a free function unit $F \in \mathcal{F}_{op(i)}$ which in the following cycles performs the actions required by I_i. Since all 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 phase flags *busy*, *rd*, *ex* and *wb* per unit F with the following meaning:

$F.busy$: If this flag is active (1), then F is busy, i.e., F is reserved for an active instruction I. If $F.busy = 0$, the unit F is free and all the other status information of F corresponds to a retired instruction.

$F.rd = 1$ indicates that unit F has *read* the operands of its current instruction and has passed to the execution phase.

$F.ex = 1$ indicates that unit F has finished the actual computation and has switched from the *execute* phase to the write back phase.

$F.wb = 1$ indicates that unit F has written back its result into its destination register.

Instruction Information of the Function Units The scoreboard also provides entries 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:

$F.Vx = 0$: the data of operand $F.Sx$ is not valid yet, or unit F has already read it.

These flags are a combination of a true valid flag and the flag *rd*. During instruction execution, these flags are supposed to switch from 0 to 1 when the data become valid and back to 0 after the data are read, and then remain inactive till a new instruction is issued to unit F.

The two entries $F.Px \in \{0, \ldots, f\}$ specify the function unit which is going to *produce* operand $F.Sx$ of instruction I while I is active. The value 0 indicates that the operand was already valid on the instruction issue of I.

4.3 The Bookkeeping

The scoreboard requires some initialization, although this is not stated explicitely in the literature[3,1]. On power up, we therefore clear the whole scoreboard, i.e., all its entries are initialized to zero.

The scoreboard then issues the instructions $I_1, \ldots I_p$ of a program P in sequential order to the adequate function units, but only one instruction at a time; thus

$$\forall i = 1, \ldots, p-1 \quad : \quad \iota(i) < \iota(i+1). \tag{6}$$

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 scoreboard postpones the reading of the source registers till both registers hold the current value, i.e., till both sources are valid.
- After the function unit run to completion, the scoreboard tries to schedule the write back. This must be postponed till no function unit with register $F.D$ as a source requires the old value any longer.
- 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 and its destination $F.D$ are then freed.

Note, in the data structure for the function units only the busy flag is cleared while retiring. All the other entries keep their value till a new instruction is issued to that unit.

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

Instruction Issue Let instruction I_i with $D(i) = S1(i)\ op(i)\ S2(i)$ to be issued next. It is issued as soon as the destination register $D(i)$ and a function unit 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. If that is the case, then the source is invalid. Table 7 summarizes these actions.

Table 7: Bookkeeping of the issue phase

```
while ($I_i$ not issued yet) $\wedge$ ($I_{i-1}$ issued)
if (Res.D(i) = 0) $\wedge$ ($\exists$ F $\in F_{op(i)}$ : F.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);
}
```

Read Operands After issuing an instruction to function 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 the valid flags of both operands are set. If so, the registers are read, the flags $F.Vx$ are cleared and the phase flag $F.rd$ is activated (Table 8).

Table 8: Bookkeeping of the read phase performed for a function unit F

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

Execute Phase After reading its operands, unit F remains in the execute phase till the computation run to completion. Then, the flag *F.ex* is activated and the instruction switches to the next phase (Table 9).

Table 9: Bookkeeping of the execute phase performed for a function unit F

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

Write Back Phase In this phase, unit F tries to write back its result. However, the scoreboard must avoid the writing, as long as there is another function unit which has register *F.D* as a source and still has to read the old value of *F.D*. While the unit F writes its result into register *F.D*, the scoreboard gives this register free. (Table 10).

Table 10: Bookkeeping of the write back phase performed for a function F

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

Notification Phase In the 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 (Table 11). Note, that also units with instructions depending on an earlier result of unit F are notified.

Table 11: Bookkeeping of the notification phase performed for a function F

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;

Correctnesss The bookkeeping in the write back and notification phase of instruction I is not obvious. Let I be executed by unit F. One would expect, that only function units with instructions issued *before* I are checked whether they still need the old value of the destination register $F.D$. One would further expect, that only units with instructions started *after* I are notified of F updating register $F.D$. Thus, it is not obvious that the above protocol is correct, although it is consistent with the original scoreboard mechanism [3,1].

5 A Counter Example

Consider a slow function unit F – say a divider – waiting for its source S1 to be produced by a fast unit. In case the fast unit completes another operation after the divider read its argument but before it retired, then the generalized set condition in the notify mechanism forces the valid flag $F.V1$ back to 1. This flag can not be cleared till a new instruction is issued to the divider F.

In this case, $F.V1 = 0$ still implies that the data is not valid or already read, but the revers does not hold. $F.V1 = 1$, even so unit F has already read its operands.

Along these lines, we now construct a program for an architecture $\mathcal{A}$ which forces the scoreboard into a deadlock.

5.1 The Model Architecture

For this purpose, we use an architecture $\mathcal{A}$ with exactly one adder and divider and with at least one multiplier. The latency of the divider and of the multiplier must be at least 3 cycles longer than the one of the adder (Table 12).

Table 12: Latency (in cycles) of the model architecture

	Add	Mul	Div
general $\mathcal{A}$	ta	$\geq ta + 3$	$\geq ta + 3$
$\mathcal{A}$	1	4	5
CDC 6600, FPU	4	10	8, 29

The floating point core of the CDC 6600 is a possible candidate. However, we would like to reduce the cycles till a deadlock occurs, and therefore choose the smaller latencies of $\mathcal{A}$ listed in table 12.

5.2 Causing a Deadlock

We will now analyze, how the scoreboard performs when running the following program on the model architecture $\mathcal{A}$. Note that only register $R3$ causes data hazards, and that the second addition is data independent of all the preceding instructions.

Table 13: Code of the counter example

(1)	R3 = R1 + R2
(2)	R4 = R4 / R3
(3)	R3 = R1 * 42
(4)	R5 = R5 + R5
(5)	R3 = ...

During the execution of an instruction I on unit F, only a few of the scoreboard entries related to that unit do change, namely the valid flags $F.Vx$, the busy flag and the phase flags. Figure 5 therefore only traces these relevant signals of the three functional units over the run time of the above program.

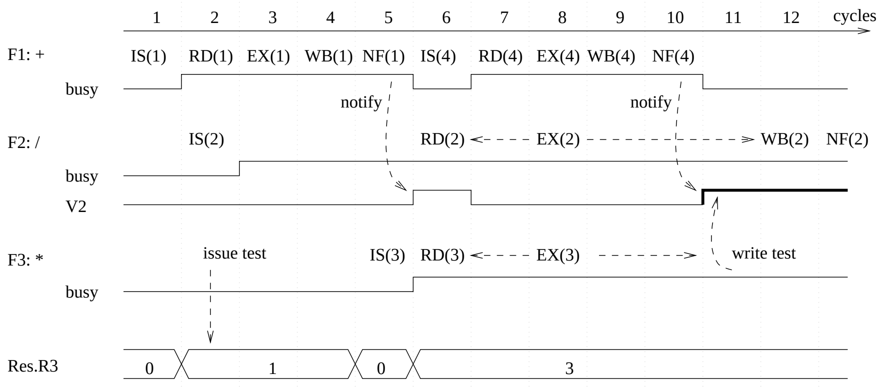

Figure 5: Signal propagation relevant to the deadlock. IS(i), RD(i), EX(i), WB(i) and NF(i) denote the issue, operand read, execute, write back and notification cycles of instruction I_i.

The first instruction is issued to unit F1 in cycle $t = 1$. F1 does not encounter any problems and therefore reads its operands, performs the computation and then writes the result in cycle 4.

In the second cycle, instruction I_2 is issued to the divider. Since register R3, the destination of instruction I_1, serves as its second source, the valid flag F2.V2 is cleared. The divider must postpone the reading of its operands till it received a notification from the adder. This notification occurs at the end of cycle 5. Thus, the divider reads its operands in cycle 6 and starts its computation in cycle 7.

The third instruction has the same destination as instructions I_1, namely $D(3) = R3 = D(1)$, and can therefore not be issued before cycle 5, i.e., before the adder cleared the flag Res.R3. The operands of the multiplication are valid by then, and I_3 starts the computation in cycle 7 and finishes it in cycle 10.

The fourth instruction, also an addition, writes the result in cycle 9 and notifies other units in cycle 10. Since the division is still active, and since its source F2.S2 depended on an earlier result of the adder, the corresponding valid flag F2.V2 is activated again. However, there consists *no* data dependency 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 dependencies, especially to those with the division. Since the source register F2.S2 of the divider equals the destination F3.D 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 entry F2.S2 is changed.

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 ($i > 4$, $op(i) = /$) is issued to unit F2:

$$\iota(i) < \omega(3). \tag{7}$$

On the other hand, there exists a data dependency 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

$$\iota(5) < \iota(i) < \omega(3) < \iota(5), \tag{8}$$

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

5.3 Correction of the Scoreboard

The deadlock was due to the combination of unusual valid flags and a generalization of the set conditions in the notify mechanism. During write back of instruction I_i, the scoreboard also notified instructions started prior to I_i. Besides being wrong, this bookkeeping is also counter intuitive.

An obvious modification would therefore be to restrict the test in the notify phase of I_i to active units $F(j)$ processing an instruction I_j, $j > i$.

However, this restriction requires a nontrivial hardware extension, because the scoreboard must store the order of the active instructions. Instead, the following to modifications could be used to compensate for the generalized notify test:

- to make the flags Vx true valid flags and then test for operands "valid and unread", i.e., $F.Vx \wedge /F.rd$ instead of $F.Vx$,
- to clear the entries $F.Px$ when leaving the read phase. After fetching its arguments, F no longer depends on the result of unit $F.Px$ anyway. Once these entries are cleared, they cannot cause the corresponding valid flags $F.Vx$ to be falsely activated again.

In our counter example, any of these modifications avoids the deadlock, but the correctness of the modified scoreboard mechanism still needs to be proven. Due to lack of space, the proof is omitted, but it appears in paper[9].

6 Summary

Many rules of thumb in engineering can be formally tested in our framework. In this paper, we motivated the following rules:

- Pipelining makes the control of a fixed-point RISC core about one third cheaper.
- Full result forwarding increases the cycle time of the DLX design by 30%. Nevertheless, in connection with a standard 5-stage pipeline, it results in a speedup of a factor 2.3.
- When using partial instead of full result forwarding, another 10 to 26% performance increase can be achieved.
- In the sequentail DLX design, the hardware for the precise, nested processing of interrupts increases the cost of the data paths by 20% and the cost of the control by 100%. Pipelining even doubles these cost increases.

It was also shown what is required in order to support the precise nested processing of multiple interrupts, and how pipelining worsens the correct interrupt handling. Furthermore, with respect to the correctness of designs, it is proven that the original scoreboard mechanism as introduced in the CDC 6600 is not correct, because it can run into deadlocks.

Acknowledgment

The problem to prove the correctness of the scoreboard mechanism was suggested by Wolfgang J. Paul.

References

1. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, INC., San Mateo, CA, 1990.
2. S.M. Müller and W.J. Paul. *The Complexity of Simple Computer Architectures.* Lecture Notes in Computer Science 995. Springer, 1995.
3. J.E. Thornton. *Design of a Computer: The Control Data 6600.* Scott Foresman, Glenview, Ill, 1970.
4. C. Nakata and J. Brock. *H4C Series: Design Reference Guide. CAD, 0.7 Micron* L_{eff}. Motorola Ltd., 1993. Preliminary.
5. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
6. D.A. Patterson and J.L. Hennessy. *The Hardware/Software Interface.* Morgan Kaufmann Publishers, INC., San Mateo, CA, 1994.
7. R. Knuth. *Quantitative Analyse von DLX-Pipeline-Architekturen.* PhD thesis, Universität des Saarlandes, Saarbrücken, FB. Informatik, 1996.
8. S.M. Müller and R. Knuth. Correctness of a mechanism for precise nested processing of interrupts in pipelined designs. Technical report, CS Department, University of Saarland, Germany, http://www-wjp.cs.uni-sb.de/~smueller, 1996.
9. S.M. Müller and W.J. Paul. Making the original scoreboard mechanism deadlock free. In *Proc. 4th Israeli Symposium on Theory of Computing and Systems (ISTCS).* IEEE Computer Society, 1996.