A Practical Method for Rigorously Controllable Hardware Design

E. Börger and S. Mazzanti

Università di Pisa, Dipartimento di Informatica, Corso Italia, 40, 56125 Pisa, Italy
(boerger,mazzanti@di.unipi.it)

2

Abstract. We describe a method for rigorously specifying and verifying the control of pipelined microprocessors which can be used by the hardware designer for a precise documentation and justification of the correctness of his design techniques. We proceed by successively refining a one-instruction-at-a-time-view of a RISC processor to a description of its pipelined implementation; the structure of the refinement hierarchy is determined by standard instruction pipelining principles (grouped following the kind of conflict they are designed to avoid: structural hazards, data hazards and control hazards).
We illustrate our approach through a formal specification with correctness proof of Hennessy and Patterson's RISC processor DLX but the method can be extended to complex commercial microprocessor design where traditional or purely automatic methods do not scale up. The specification method supports incremental design techniques; the modular proof method offers reusing proofs and supports the designer's intuitive reasoning, in particular "local" argumentations typical for upgrading and optimizing machines. Since our models come in the form of Abstract State Machines, they can be made executable by ASM interpreters and can thereby be used for prototypical simulations.

1 Introduction

It is well known that microprocessors are subject to subtle design errors. Conventional methods like simulation to debug processors before fabrication consume enormous resources in terms of manpower and of machines. In recent years various formal verification techniques have been proposed to overcome the well-known theoretical and practical limits of such conventional techniques and have been applied to the analysis of a certain number of (usually rather simple and unpipelined) microprocessors. Some typical examples standing for many others are [JBG86] [Bow87] [C88] [C89] [Hunt89] [LC91] [Her92] [Be93] [Win94] and [Ta95] which includes an excellent detailed survey.

We develop a practical method which reduces the labor required to do formally supported design and verification of microprocessors by orders of magnitude. The method allows one to define a hierarchy of refinement steps each of which is focussed on a specific feature of the processor to be constructed and

comes with a correctness proof expressing the intuitive reasoning of (i.e. the justification given by) the designer. The guiding principle of these successively refined specifications is to mimic as closely as possible the incremental features in hardware design. We add to this incremental approach a locality principle (see the notions of projection and of relevant locations below) which supports the local reasoning typical for practical hardware design. As a by-product one can break the proof of the properties of interest into elementary inductions and a few natural case distinctions corresponding to the different pipelining conflict types and the methods to solve them; in this way we prepare the ground for additional support by a mechanical verification using automated proof development systems such as PVS, HOL, IMPS or model checking systems.

We concentrate our attention in this paper on control, where notoriously most errors are found during the design of a processor. We do this for the challenging case of microprocessors with an instruction pipeline, exemplified through the standard pipelined RISC processor DLXdeveloped by Hennessy and Patterson [HP90] in order to illustrate the essential features of RISC processors like the MIPS R3000 (see [Hen93]), Intel i860, Sun SPARC, Motorola M88000. Pipelining is a key implementation technique used to make fast CPUs. It provides a simultaneous execution of multiple instructions which exploits the independence between (parts of) instructions, as a result of which the execution speed for programs is improved. Since pipelining is not visible to the programmer, the more it is crucial to ensure that the semantics of instructions is preserved by the concurrency of operations which is inherent in this technique. We prove the correctness of Hennessy and Patterson's pipelined processor with respect to its sequential model (one-instruction-at-a-time view of the processor). The task therefore consists in starting from a mathematical model for the datapath and the sequential control of DLX, refining this model to the pipelined version of DLX and proving the correctness of the refinement process.

The overall structure of our design-driven refinement hierarchy is determined by the major instruction pipelining principles which can be grouped following the kind of conflict they are designed to avoid: structural, data and control hazards. For the crucial transition from the sequential (programmer-view) DLX model to the parallel execution model of its pipelined variant DLX^p we provide a (local projection) technique for extracting from certain segments of a concurrent DLX^p–computation—where at each step many operations concerning different (types of) instructions are performed in parallel—an equivalent sequential DLX–subcomputation of the one instruction (type) under analysis (see the notions of *relevant* and *result locations* and of *instruction cycles* below). For this first refinement step we concentrate on the current techniques to make the pipelined version of DLX free from structural hazards and abstract from the more sophisticated data or control hazards and stalls, i.e. for the proofs we assume the compiler to organize the sequence of instructions in such a way that they are sufficiently independent upon entering the pipe. In the further refinement steps we show that for the refined models DLX^{data}, DLX^{ctrl} and DLX^{pipe} of DLX^p the compiler assumption on data and control hazard freeness can piecemeal be

dispensed with. (For a transparent and easily manageable proof it turned out to be advantageous to distinguish data hazards for not jump instructions—solved in DLX^{data}—and data hazards for jumps instructions—solved in two steps in DLX^{ctrl} and in DLX^{pipe}.[1]) Alltogether we therefore justify the following claim.[2]

Main Theorem (Correctness of DLX^{pipe} with respect to DLX).[3]
For each DLX program P, the result of the sequential execution of P on the machine DLX is the same as the result of the pipelined execution of P on the machine DLX^{pipe}.

Due to the systematic use of successive refinements, organized around the different pipelining problems and the methods for their solution[4], our approach can be applied for the design-driven verification as well as for the verification-driven design of RISC cores (including their rigorous documentation) at any level of abstraction. The modularity of the specification and analysis method provides the possibility to reuse correctness theorems along the refinement hierarchy. Such a decomposition of a complex goal into simpler subgoals corresponds to well established mathematical and engineering practice. Our method is still practicable when instead of DLX one has to deal with more complex microprocessors, more advanced pipeling techniques or more sophisticated memory systems.[5]

The divide-and-conquer approach to design-driven formal verification advocated here has proved to be practically viable for complex systems where traditional approaches failed; see the proofs in [BR95] [BD95] for the correct-

[1] We are grateful to Sofiene Tahar for pointing out to us that an architecture which is similar to our DLX^{pipe} has been implemented in [DeTa94].

[2] It has been suggested to view our theorem as saying that DLX^{pipe} is sequentially consistent with respect to DLX in the sense of Lamport [La79]. This is an oversimplifying interpretation. Lamport's definition is phrased in terms of certain *"execution results"* being *"the same"*. One of the major problems we solve in this paper is to define in a rigorous but transparent manner *a)* what the computer architect understands by *"the result of a DLX execution with pipeling"* and *b)* precisely at which moments during the pipelined execution of DLX the results of this execution have to be checked for *"being the same"* as the result of the sequential DLX computation.

[3] This theorem has been announced in [Bo95].

[4] This is the basic methodological difference between our refinement hierarchy and the interesting hierarchical structuring proposed in [W90] and followed also in [WC95] and [Taku95]. The abstractions in these papers reflect some typical compiler hierarchy levels, leading from the assembler level through the level of microprogrammed code to (code formalizing) the electronic block model which constitutes the gate level of the hardware system. We define our abstractions in order to isolate and reflect as closely as possible the different hazard types and the methods for their solution. It is of course possible to combine the two structuring methods where this is needed to break down the complexity of the overall problem into pieces which can be handled relying on assistance from machines.

[5] The work on this paper grew out from a reverse engineering project of a parallel architecture (see [BoDC95]) where we faced pipelining together with VLIW parallelism. In [BoDC95] we have used our abstraction and refinement technique to structure a real-life processor into simple and rigorously defined basic components.

ness of compiling Prolog programs to the Warren Abstract Machine or Occam programs to the Transputer and the work on the machine checked versions of the WAM correctness proof using KIV [A95] and Isabelle [P96]. During the last years theorem provers have been used to verify also pipelined processors, but either the processors are simple or the verification is rather complex [Cy93, BB93, Ro92, SGGH91, SB90]. For two recent projects to formally verify DLX using HOL and PVS see [TaKu95, Cy95][6]. In the model checking verification of a subset of the pipelined DLX in [BD94], Dill's goal is an automatic verification procedure where the human intervention is confined to the development of operational descriptions of the specification and the implementation. Our primary concern in this paper is to support the actual design work by a simple method which can be used by the computer architect to lay down his design steps and to reason about their effect in a rigorous, checkable and falsifiable way. To this purpose we provide a rigorous simple behavioral modelling of both the specification and the implementation and relate the two by a hierarchy of transparent definitions and (proofs of) properties; we try to break the complexity of the processor by revealing the structure of the run time interaction of its main parts and by linking in an understandable hierarchical way the sequential and the pipelined execution models.

To break the complexity of real-life non-toy systems it is crucial not to be bound by the straitjacket of an a priori given formal framework and to be able to separate the specification and its justification from mechanical verification concerns. One thing is to rigorously support the designer's reasoning and the structuring of his work into intellectually manageable parts; another thing is the detailed logical encoding which is unavoidable to make the specification understandable and checkable not for a human user, but for a machine. Both forms of "understanding" and "proving" have their own logic, needs and merits. Combining the two will enable us to master the complexity of current computer systems. Once the largely creative and hardly mechanizable decomposition effort has led to a hierarchy of stepwise refined rigorous models, related by lemmas stating the properties of interest, the justification of the desired overall behavior of the system can be split into separate, possibly mechanizable, proofs of such lemmas. Flexible and sufficiently expressive systems for machine assisted verification will incorporate such hierarchical decomposition techniques. We advocate a brain–AND–brawn approach (see [Bo95]) for both design-driven post-verification and verification-driven design using on-the-fly-verification.

It will help if the reader is familiar with the semantics of *Abstract State Machines* defined in [G95][7] although what follows can be understood correctly

[6] Cyrluk's specification and implementation can be viewed as a PVS formalization of the semantics of (some of) the rules of our models DLX and DLX^p. Cyrluk, Tahar and Kumar do not define our notions of relevant and of result locations which allow us to structure and to localize the proof obligations boiling them down to the bare minimum. Using these notions we can recover the sequential states from successive pipelined states by simple projections which directly support the way the designer reasons about the relation between sequential and parallel pipelined execution.

[7] Previously Gurevich's ASMs have been called evolving algebras.

by reading our ASM rules as pseudo–code over abstract data types. We therefore abstain from repeating here the definitions of [G95].

2 Parallelizing the sequential DLX to DLX^p

The one-instruction-at-a-time machine DLX can be constructed by a straightforward formalization of the control graphs in [HP90]. We define DLXas Abstract State Machine in the appendix[8] without commenting further and refer to [BM96] for explanations about how the abstractions of this sequential model make our proof method uniform with respect to the size of the register file, the width of the datapath, the instruction set, the memory access (bandwidth), etc.[9] We explain in the rest of this section the few changes which suffice to refine DLX to a machine DLX^p where at each clock cycle simultaneously five basic steps are executed, one for each of five instructions.

The five basic execution steps appearing in DLX are instruction fetch *(IF)*, instruction decode including the fetching of operands *(ID)*, execution proper for ALU operations and (data or branch) address calculation using the ALU*(EX)*, memory access *(MEM)* and writing the computed result back into the final register-file destination *(WB)*. The order in which these basic execution steps follow each other for the execution of an instruction is described in DLX by a 0-ary function *mode*. Ideally one can pipeline DLX by letting the processor execute during each clock cycle simultaneously five basic steps, one for each of five instructions. This can be realized by eliminating from DLX the sequential control by *mode* and by replacing where necessary the *mode* guards by operation code guards corresponding to the pipe stage of the instruction in question. In the resulting new machine, at each moment for each of the five basic execution steps a rule is applied (clock synchronized architectural parallelism).

However one has to guarantee that the five pipe stages which are active on every clock cycle do not compete for resources, each functional architectural unit being available at each step only once. We describe briefly how the rules of DLXcan be refined to DLX^prules which resolve these structural conflicts.

Resolving structural conflicts. The simplicity of the DLX instructions set results in limited resource competition and in simple datapath/control refinements to avoid it. Four major groups of resources have to be doubled so that any

[8] When using instruction related functions like *opcode, fstop, scdop, iop* etc., we usually suppress their standard argument, namely the content of the instruction register IR. Standard terminology and notation are adopted without explanation from [HP90].

[9] In order to concentrate on the essential features of the pipelining parallelism, we start here not with the instruction set architecture as seen by the programmer (assembly language one-instruction-at-a-time view), but with its refinement where it becomes visible that each instruction is executed in stages (pipelining steps). For reasons of simplicity we skip the floating point instructions of DLX; although the treatement of hazards is more complex with the (multicycle) floating point operations, the concepts are the same as for the integer pipeline. We do care however not to abstract away crucial control features like the user–requested interrupt handling.

combination of operations can occur in pipe stages which are executed simultaneously in one clock cycle, namely the memory access (to fetch instructions), an addition mechanism (to increment the program counter PC), the memory data register (for overlapping load and store instructions), and latches for the instruction register IR, for PC and for the ALU output C (to hold values which are needed later in the pipeline)[10].

Instruction fetching and incrementing PC**.** A memory access conflict between instruction fetching and load/store instructions is avoided by increasing the memory bandwith, formalized by an additional memory access function mem_{instr} used only for fetching instructions and supposed to be a subfunction of the DLX function mem; in this way we abstract from any particular implementation feature related to using separate instruction and data caches which we intend to treat in a later refinement step. [11] Another resource conflict which would appear at each clock cycle concerns the ALU had we to use it for incrementing PC. The usual solution consists in providing a separate PC-incrementer, namely our abstract function $next$. Thus we have the new rule $FETCH$ below, belonging to the pipe stage set IF; the condition $jumps$, defined by[12]

$$opcode(IR1) \in JUMP \vee (opcode(IR1) \in BRANCH \wedge \overline{opcode(IR1)}(A) = true),$$

ensures that PC can be updated by the $FETCH$–rule only when no jump or branch rule has to update PC in the execution phase. The new rule $OPERAND$, belonging to the pipe stage set ID, is obtained from the DLX homonym by deleting the $mode$ guards and updates.

FETCH	$IR \leftarrow mem_{instr}\ (PC)$, **if** $\neg jumps$ **then** $PC \leftarrow next\ (PC)$	OPERAND	$A \leftarrow fstop\ (IR)$ $B \leftarrow scdop\ (IR)$

Latches for longer living values. Some of the values which appear during the execution of an instruction at a certain pipe stage are needed at later pipe stages and have to be copied in order not to get overwritten by a subsequent instruction occurring in the pipeline. This is the case for (segments of) IR. For reasons of simplicity we abstract from instruction format and decoding details and provide three additional registers $IR1$, $IR2$, $IR3$ to keep copies of a fetched

[10] The concept of simultaneous execution of multiple ASM rules allows us to abstract from the distinction of pipe stages into a writing and a subsequent reading phase (see [TaKu95]). This justifies the simultaneous execution of for example the rules $OPERAND$ and MEM_ADDR or $Pass\_B\_to\_MDR$. It also means that we consider the simultaneous read and write access of the register file (by the rules ID and WB) as not constituting a resource conflict. The explicit introduction of phases would come up to a routine extension of our rules.

[11] The DLX processor does not support self-modifying code. That feature, which can be found in older usually non-pipelined architectures, would require a much more subtle treatment of control hazards than the one present in pipelined processors.

[12] $IR1$ cointains the value IR had in the previous clock cycle, see below. By $\overline{opcode(\ldots)}$ we denote the function encoded by $opcode(\ldots)$. Registers A, B store outputs from register file registers for use in later clock cycles.

instruction through the pipe stages *EX, MEM, WB*, i.e. with the following new preservation rules belonging to the rule sets *ID, EX, MEM* respectively:

$$\boxed{\text{Preserv IR}}\ IR1 \leftarrow IR, \quad \boxed{\text{Preserv } IRi}\ IR(i+1) \leftarrow IRi \quad \textbf{with } i = 1, 2.$$

Two 0-ary functions *PC1, C1* are needed to save the values of *PC, C* for one pipe stage. *PC1* provides at pipe stage *EX* of an instruction I a copy of the value of *PC* after the FETCH stage of I (serving in case I is a jump instruction the execution of which triggers a transfer or an update of that *PC*–value). *C1* provides at pipe stage *WB* a copy of the ALU output value *C* computed in the pipe stage *EX* of I (for instructions with *ALU/SET*–operations, for *JLINK* instructions and for *MOVS2I*).

$$\boxed{\text{Preserv PC}}\ \ PC1 \leftarrow PC \qquad \boxed{\text{Preserv C}}\ \ C1 \leftarrow C$$

For reasons to be explained in the next section the rule for copying the current value of *PC* into *PC1* will have this form only in the last two models DLX^{ctrl} and DLX^{pipe} and a slightly extended form in DLX^{p} and DLX^{data}.

Doubling MDR. In *DLX* the memory data register *MDR* is the only interface between the register-file and the memory and serves for both loading and storing. In the pipelined version for *DLX* a load instruction I which in the pipeline immediately precedes a store instruction I' would compete with I' for writing into *MDR* in its pipe stage *MEM* (when I' in its pipe stage *EX* wants to write *B* into *MDR*). This resource conflict is resolved by doubling *MDR* into two registers *LMDR* and *SMDR* and by refining as follows the *DLX*–rules *MEM_ADDR* and *Pass_B_to_MDR*, both belonging to the set *EX*:[13]

if $opcode\ (IR1) \in LOAD \cup STORE$	**if** $opcode\ (IR1) \in STORE$
then $MAR \leftarrow A + ival\ (IR1)$	**then** $SMDR \leftarrow B$

The *DLX*–rule *MEM_ACC* is divided in DLX^{p} into the following two refined rules, one for *LOAD* and one for *STORE*, both belonging to the set *MEM*:

$$\boxed{\text{STORE}}\ \textbf{if } opcode\ (IR2) \in STORE \ \textbf{then } mem\ (MAR) \leftarrow SMDR \qquad \boxed{\text{LOAD}}\ \textbf{if } opcode\ (IR2) \in LOAD \ \textbf{then } LMDR \leftarrow mem\ (MAR)$$

The new rule *Pass_B_to_MDR* requires a new direct link from the exit of B to the entry of *SMDR* in order to avoid the use of the ALU for this data transfer.

Speeding up the pipe stages. Since all pipe stages proceed simultaneously and the time which is needed for moving an instruction one step down the pipeline is a machine cycle, the length of the latter is determined by the time required for the slowest pipe stage. The two *DLX*–rules *ALU, ALU'* are combined into the following DLX^{p}–rule *ALU* (belonging to the set *EX*), thus eliminating the intermediate step to put the right second operand into *TEMP*.[14]

[13] The register *MAR* stores the address for the memory access. The function *ival* yields the immediate value encoded in an instruction.

[14] The function *iop* detects operation code for immediate operations.

if $opcode\ (IR1) \in ALU \cup SET$ **then if** $iop\ (opcode\ (IR1)) = true$
then $C \leftarrow \overline{opcode\ (IR1)}\ (A,\ ival\ (IR1))$
else $C \leftarrow \overline{opcode\ (IR1)}\ (A,\ B)$

The DLX–rule $SUBWORD$ (which selects and outputs to C the required portion of the word loaded from the memory) is incorporated into the following $WRITE\_BACK$–rule under the guard that the value to be written comes through a loading instruction; if this value has been computed by executing an ALU/SET, $JLINK$, $MOVS2I$ instruction, it comes from $C1$. The price for this refinement is linking the exit of $LMDR$ directly (without passing through $C1$) to the entry of the register-file and adding to the latter a selector for choosing among $C1$ and (the required portion of) $LMDR$. Transfering a subword of $LMDR$ into a destination register in the following rule can be realized without using the ALU by relying upon the usual shift functions of registers like $LMDR$.

WRITE_BACK **if** $opcode\ (IR3) \in ALU \cup SET \cup \{MOVS2I\} \cup JLINK$
then $dest\ (IR3) \leftarrow C1$

if $opcode\ (IR3) \in LOAD$
then $dest\ (IR3) \leftarrow \overline{opcode\ (IR3)}\ (LMDR)$

The remaining DLX^p–rules—namely $MOVESPECIAL$, $JUMP$, $BRANCH$—all belong to the pipe stage EX and are obtained from their DLX-homonyms by deleting the $mode$ guards and updates and by replacing the arguments IR, PC by $IR1$, $PC1$ respectively. This concludes the specification of the ASM model DLX^p which is spelled out in full in the appendix.

3 Justifying the correctness of the parallelization

For the proof of the correctness of DLX^p with respect to DLX we start by defining the notions of result location, of used location and of relevant location which will allow us to recover DLX–states from successive pipelined DLX^p–states by simple projections. We consider only computations which are reachable from appropriate initial states. We say that two computations C in DLX and C^p in DLX^p *correspond* to each other if their *initializations* coincide on the common signature except where explicitly stated otherwise. For DLX–initializations we assume $reg\ (IR) = undef$ and $mode = FETCH$, for DLX^p–initializations $reg\ (PC1) = reg\ (C1) = reg\ (IRi) = undef$ for i=1,2,3. We often use $f\ (undef) = undef$, for each function f. We say that a computation is initialized or starts with an instruction $instr$ if $mem\ (PC) = mem_{instr}\ (PC) = instr$.

Instruction Cycles. We can justify the correctness claim by a series of simple local arguments—one for each instruction (class)—be decomposing computations into segments each of which constitutes a subcomputation during which a given instruction is executed completely. In DLX computations, an *instruction cycle* for $instr$ is any subcomputation which starts with $mode = FETCH$ and $mem_{instr}\ (PC) = instr$ and leads to the next state with $mode = FETCH$;

in DLX^p computations, an *instruction cycle* for *instr* is any subcomputation which starts with *instr* and ends with the first following pipe stage of *instr* at the end of which the values of all the result locations of *instr*, as defined below, are computed. We call this pipe stage the end (pipe) stage of *instr*; whether it is *EX(instr)*, *MEM(instr)* or *WB(instr)* depends on *instr* and is defined in table 1.

We prove the correctness of DLX^p with respect to DLX instructionwise by showing that in every pair (C, C^p) of corresponding DLX/DLX^p–computations, *corresponding instruction cycles* compute the same result. The correspondence between instruction cycles in C and in C^p is defined by the order in which they occur: if I_1, I_2,...and I'_1, I'_2,...are the instruction cycles of C and C^p respectively (in the order in which they appear there), then I_i and I'_i correspond to each other. By I_0, I'_0 we indicate the initial state. We say that I_0, I'_0 formalise the "result" of "no computation step". In particular we will show below that I_i and I'_i are instruction cycles for the same instruction.

Result Locations. The simplicity of the DLX instruction set makes it easy to localize, uniformly for a few classes of instructions, where and when the result of an instruction belonging to a class becomes visible in a DLX/DLX^p–computation, namely in certain registers or memory locations. The pair $<$ *reg, PC* $>$ is defined to be a *result location* for each instruction *instr*. The other result locations for *instr* are determined by table 1.[15] We assume *dest (instr)* = *R31* in case *instr* $\in$ *JLINK*. *reg (fstop (instr)) + ival (instr)* is supposed to be a memory address if *instr* $\in$ $LOAD \cup STORE$.

The result of *instr* is given by the values *f(a)* assigned to the result locations <*f, a*> for *instr* through the execution of *instr*. In DLX–computations it can be read off from the final state of the inspected instruction cycle for *instr*. In DLX^p–computations the result of (an occurrence of) *instr* is smeared over the whole instruction cycle of *instr* and must be collected from different pipe stages, depending on the instruction type. Table 1 defines which result is collected after which pipe stage[16]. This completes the definition of the result of the execution of occurrences of *instr*. The result of *instr* is also called the result of the instruction

[15] In DLX every instruction has only one result proper and this result is written at the end of the instruction's execution.

[16] In this way we provide a simple explicit and local definition of the global and implicit data and time abstraction functions which are introduced in [WC95] to "collect different pieces of the pipelined state stream at different times and package them into a state record to appear in the non–pipelined state stream at a particular time". [W90] could make successful use of the orthogonality of data and temporal abstraction functions in his hierarchical approach to microprogrammed (non pipelined) microprocessor verification. When pipelining is present these two abstractions are not orthogonal any more. [WC95] define an new abstraction function in order "to preserve the illusion that instructions execute sequentially in the architectural model even though the pipelined implementation performs operations in parallel". By using the notion of result locations defined here, together with the notion of relevant locations defined below, we reduce the complexity of such an abstraction function and boil it down to the consideration of local features which are familiar from the design practice.

cycle of (the given occurrence of) *instr*. For notational convenience, the result of a computation is defined as the sequence of the results of its instruction cycles.

Result Location	**Updated by *instr* in**	**to be collected after the end of the pipe stage**
$<$ *reg, dest (instr)* $>$	$ALU \cup SET \cup LOAD \cup JLINK \cup \{MOVS2I\}$	WB(*instr*)
$<$ *reg, IAR*$>$	$\{TRAP,\ MOVI2S\}$	EX(*instr*)
$<$ *mem, arg*$>$	$STORE$	MEM(*instr*)
$<$ *reg, PC*$>$	$JUMP \cup BRANCH$	EX(*instr*)
	$\notin JUMP \cup BRANCH$	IF(*instr*)

Table 1. Result locations and their collection time. *arg* is an abbreviation for the value of *reg (fstop (instr)) + ival (instr)* at the moment of fetching *instr* in DLX.

Used Locations and Hazards. Given an instruction cycle for I in a DLX^p computation, denote by $I \overset{1,2,3}{<} I'$ that an instruction cycle for I' is starting 1,2 or 3 steps after the one for I. Hazards can arise if $I \overset{1,2,3}{<} I'$ and I' uses a result of I. Table 2 defines what is "used" by an *instr* in a run, namely—besides static information like the one encoded in *instr* and accessed using the functions *opcode, nthop, dest, iop, ival*—the content of operand registers in the register file, of PC, of memory locations and of the interrupt address register IAR. The table also defines the critical pipe stage during which the machine needs the correct value of that location. A simple analysis of the DLX^p–rules (see the definition of *Irrelev 1,2* below) shows that conflicts can arise in two ways, namely *a)* if I' uses, as one of its operands, the content of the destination register of a preceding instruction I in the pipe, *b)* if I' enters in the pipe shortly after a jump or branch instruction. For the analysis of these *data and control hazards* we distinguish whether or not the data dependence concerns a jump or branch instruction.

Definition. *I' is data dependent on I iff* $I \overset{1,2,3}{<} I'$ *and one of (i), (ii) holds.*
(i) *dest (I)* $\in$ { *fstop (I'), scdop (I')* } **and** $I' \notin JUMP \cup BRANCH$,
(ii) *dest (I) = fstop (I')* **and** $I' \in JUMP \cup BRANCH$.

A DLX^p computation is *data hazard free* if it contains no occurrence of an instruction which is in the pipe together with an occurrence of an instruction on which it is data dependent.

When a jump or branch instruction I is fetched, the two instruction cycles starting 1 and 2 steps later generate results which would spoil the continuation of the computation once the jump has been executed (after the stage EX(I) which updates PC to its correct value). In order to separate the correctness proof for the parallelization of DLX from the concern about such control hazards, we assume in this section that in the transformation P^p of P the compiler places two

empty instructions (formalized by the value *undef*) after each jump or branch instruction occurring in the DLX-program P; we stipulate that these empty instructions do not start an instruction cycle. Without loss of generality we assume that the empty instructions are put into new locations which are linked by the extended *next* function to the old locations in the standard way.

Letting the "compiler" avoid control conflicts by arranging the instructions of P into P^p–code, we have to work with a slightly extended PC–preservation rule. When a jump or branch instruction I is fetched at address $l = reg\ (PC)$, PC is updated to $l' = next\ (l)$ which in P^p is the address of *undef*. But in the EX(I)–stage the new value of PC must be computed on the basis of the value of $next\ (next\ (l'))$, i.e. the value of $next\ (l)$ for the DLX–program P. Therefore $PC1$ has to store this value when PC—in case a jump or branch instruction has been fetched— contains the address of the empty instruction. Therefore the PC–preservation rule in DLX^p is $PC1 \leftarrow next\ (next\ (PC))$.

DLX^p Correctness Theorem. *Let P be an arbitrary DLX–program, P^p its transformation obtained by inserting two empty instructions after each occurrence of a jump or branch instruction. Let C be the computation of DLX started with program P and C^p the corresponding computation of DLX^p started with P^p. If C^p is data hazard free, then C and C^p have the same result.*

Proof . The decomposition of DLX/DLX^P–computations into instruction cycles allows us to prove the theorem instructionwise, using an induction over the given DLX–computation. For the inductive step we need a stronger inductive hypothesis than what is stated in the theorem. For its formulation we introduce the notion of *relevant locations* which allows us to define locally the relation between sequential states and their pipelined counterparts, avoiding the flushing technique used in [BD94] and [Cy95].

DLX^p–Lemma. *Let P, P^p, C, C^p be as in the DLX^p Correctness Theorem. For $n \geq 0$ let IC_n, IC_n^p be the n-th instruction cycle in C, C^p respectively. a) If C^p is data hazard free, then IC_n, IC_n^p are instruction cycles for the same (occurrence of a) DLX–instruction instr and start with the same values for the relevant locations used by instr. b) If IC, IC^p are instruction cycles for instr in C, C^p respectively which start with the same values for the relevant locations used by instr and if instr is not data dependent on any instruction in the pipe, then IC, IC^p compute the same result.*

A location l used by *instr* is called *relevant* except in the following two cases:

Irrelev 1. $l = < reg,\ IAR >$ **and** $instr = MOVS2I$ enters the pipe 1, 2 or 3 stages after an occurrence of $MOVI2S$ or of $TRAP$; [17]

Irrelev 2. $l = < mem,\ arg >$ **and** $instr \in LOAD$ enters the pipe 1, 2, or 3 stages after an occurrence of a $STORE$ instruction for the same value arg.[18]

[17] No conflict can arise from using $< reg, IAR >$ because $MOVS2I$, the only instruction which uses IAR, can never be in conflict with any preceding instruction. If I writes into IAR, then $I \in \{TRAP, MOVI2S\}$ and I writes into IAR in its third pipe stage; therefore if $I \overset{1,2,3}{<} I'$, then I has already written into IAR when I' uses it.

[18] No conflict can arise from using a memory location because load instructions—the

The projection of relevant and of result locations, out of sequences of computation steps of the pipelined processor, represents the state information which characterizes the sequential execution of the instruction under investigation. As we will see below it is easily shown to be semantically correct in case no potential conflict does occur.[19] Our definition of relevance will be refined in the subsequent upgraded machines by admitting as additional irrelevant locations all those where in a hazardous situation the refined architecture will take care of providing the right values for them when needed. In this way we make it explicit where and how the compiler assumptions can be weakened if the hardware is strenghthened (to solve a given type of conflicts). This illustrates the potential of ASM modelling to deal with hardware/software co-design problems in a rigorous but nevertheless simple and transparent way[20].

The lemma clearly implies the theorem. The proof of the lemma is by induction on n. For $n = 0$ the claim holds by the assumption that C and C^p correspond to each other and therefore are initialized with the same static functions and with the same dynamic functions *reg* and *mem*. In the induction step, by inductive hypothesis, for each $i \leq n$, the i-th instruction cycle IC_i^p in C^p starts with the same values for the relevant locations used by *instr i* as does the i-th instruction cycle IC_i in C and they both compute the same result. Therefore IC_{n+1} and IC_{n+1}^p are instruction cycles for the same instruction *instr* and start with the same values for the relevant locations used by that instruction. Due to the absence of stalls, the $n+1$-th instruction cycle in C^p starts after the first step of IC_n^p in case the instruction $instr_n$ is neither a branch instruction with

---

only ones which use memory locations—can never be in conflict with preceding store instructions—the only ones which write into memory locations. Indeed if $I \in STORE$ and $I' \in LOAD$, then I updates its result location $<$ *mem, reg (fstop (I)) + ival (I)*$>$ in its fourth pipe stage and I' reads the value of the location $<$ *mem, reg (fstop (I)) + ival (I)*$>$ in its fourth pipe stage too. Therefore if $I \overset{1,2,3}{<} I'$ and I' loads the value of the result location of I as updated by I, then I has already updated this result location when I' loads from there. We remind the reader that DLX does not support self-modifying code.

[19] Our localization constitutes a different way to separate the two concerns which are dealt with in [Taku95:pg.1] by splitting the correctness proof into two independent steps, namely a) showing "that each architectural instruction is implemented correctly by the sequential execution of its pipeline states", and b) showing that "under certain constraints from the actual architecture, no conflicts can occur between the simultaneously executed instructions". A similar separation, into the concern about the correct functionality and the concern about the correct processing of instructions by the pipelining, is suggested also in [Taku93, AL95].

[20] [TaKu95] separate the hardware part EBM from the software constraints SW_Constr for their contribution to imply the pipelining correctness property. The correctness proof can then be split into two steps, namely *a)* EBM implements each instruction correctly by the sequential execution of its pipelined stages, *b)* the software constraints SW_Constr guarantee that in EBM no conflicts can occur between any simultaneously executed instructions. Our stepwise refinements of (ir)relevant locations make the hw/sw-interplay between EBM and SW_Constr directly visible.

Location	Used by *instr* in	Critically in stage
$<$*reg, nthop*$>$	$ALU \cup SET \cup BRANCH \cup \{MOVI2S\} \; JUMP\text{-}\{TRAP\}$	EX(*instr*)
	$LOAD \cup STORE$	MEM(*instr*)
$<$ *reg, IAR*$>$	$\{MOVS2I\}$	EX(*instr*)
$<$ *mem, arg*$>$	$LOAD$	MEM(*instr*)
$<$ *reg, PC*$>$	$JUMP \cup BRANCH$	EX(*instr*)
	$\notin JUMP \cup BRANCH$	IF(*instr*)

Table 2. Critical stages for usage of locations.

true branching condition nor a jump; otherwise the $n+1$–th instruction cycle in C^p starts after the third step of IC^p_n due to the following *Jump Lemma* (which is easily proved by induction on the number of fetched jumps).

Jump Lemma. *If a jump or branch instruction I is fetched in a DLX^p–computation, then the following two fetched instructions are empty and at stage* ID(I) *the register PC1 is updated by the correct value to be used for the computation of the possible new PC–value in stage* EX(I).

Since the other result locations depend on the instruction type we are led to a natural case distinction. For each case it is routine to show that through corresponding updates in IC and IC^p, the same value is computed for the result location. (The details are carried out in [BM96]).

4 Data hazards for non jump/branch instructions

In this section we enrich the architecture so that it can handle data hazards for non jump or branch instructions freeing the compiler from its work to avoid these conflicts; we show how one can weaken the data hazard freeness assumption in the DLX^p correctness theorem and guarantee nevertheless the correctness of the architecture by enriching the rules with three standard features, namely the forwarding technique, new hardware links coming with appropriate additional control logic (multiplexers), and stalling. Technically speaking we refine the DLX^p machine to a machine DLX^{data} which is shown to work correctly also for the execution of non jump or branch instructions I' with data dependence on a previous instruction I in the pipe, i.e. such that condition (i) holds:

$$(i) \quad I \overset{1,2,3}{<} I' \wedge (dest\ (I) = fstop\ (I') \vee dest\ (I) = scdop\ (I'))$$

$$\wedge\ I' \notin JUMP \cup BRANCH$$

In DLX no write after write hazard can occur, because writing is allowed only in one pipe stage, namely WB, and because together with any stalled instruction

every later instruction in the pipe is also stalled. DLX has also no write after read hazard because the read stage, namely ID, precedes the write stage.

We will specify the rule refinements piecemeal, following the case distinctions whether the data hazard to be handled involves a memory access or not and whether the distance between the data dependent instructions in the pipe is 1, 2 or 3. This case analysis will justify the correctness of the refined architecture and therefore establish the following theorem.

DLX^{data} **Correctness Theorem.** *Let C be the computation of DLX started with program P and C^{data} the corresponding computation of DLX^{data} started with P^p. Assume that in C^{data} no occurrence of a jump or branch instruction is in the pipe together with an occurrence of an instruction on which it is data dependent. Then C and C^{data} compute the same result.*

Proof method. We define DLX^{data} by incrementing DLX^p, technically speaking as a conservative extension of DLX^p, so that whenever an instruction I' without data dependence to any previous instruction I satisfying (i) occurs in the pipe, DLX^{data} computes I' the same way as DLX^p does. This conservativity of the refinement allows us to prove the correctness of DLX^{data} by a case analysis in which instructions without data dependency in the pipe are dealt with by reusing the DLX^p*–Lemma* whereas the remaining instructions are dealt with by rule refinements corresponding to the cases under analysis.

Since the value of any result location different from $< reg, PC >$ is determined by the values of the arguments which are used in stage EX or MEM, it suffices to locally modify the relevant DLX^p–rules in such a way that even in the case of data dependence the correct arguments are provided. One can then weaken the assumption in the DLX^{data}–correctness statement below that corresponding instruction cycles in DLX^p and DLX^{data} start with the same values of the relevant locations used by their instruction; namely we take the hazardous locations out of the set of the relevant ones (exactly because in DLX^{data} they are taken care of by the architecture). This is a typical example how we use conservative refinements together with the localization or projection technique to mimic the way the computer architect proceeds when he enriches the processor.

Proof. As in the preceding section it suffices to prove the following lemma.

DLX^{data}**–Lemma.** *Let P , P^p, C, C^p, C^{data}, IC_n be as in the DLX^p–Lemma and in the theorem and let IC_n^{data} be the n-th instruction cycle in C^{data}.*
a) If C^{data} is free of data hazards for jump or branch instructions, then IC_n and IC_n^{data} are instruction cycles for the same DLX–instruction I' and start with the same values for the relevant locations used by I'.

Let IC, IC^p, IC^{data} be instruction cycles for any I' in C, C^p, C^{data} respectively which start with the same values for the relevant locations used by I'. Then the following two properties hold:
b) If I' is not data dependent on any I in the pipe, then IC^{data} and IC^p, and therefore also IC, compute the same result.
c) If $I' \notin JUMP \cup BRANCH$ is data dependent on some $I \overset{1,2,3}{<} I'$, then IC^{data} and IC compute the same result.

The DLX^{data}*–Lemma* is proved by induction on the number n of instruction cycles. For $n = 0$ the claim is satisfied by the assumption that C, C^p and C, C^{data} correspond to each other. The inductive step for *a)* is proved in the same way as shown for the DLX^p*–Lemma*; the Jump Lemma is true also for DLX^{data} because the same program modification P^p of P is used for C^{data} as for C^p.

For *b)* let I' be data independent of any I which precedes it in the pipe. Then it is easily checked that for each DLX^{data}–rule which is applied in IC^{data} for the execution of (this occurrence of) I', in any of its five pipe stages, the branch is taken which constitutes the DLX^p–part of that rule. Since by assumption IC^p and IC^{data} start with the same values for the relevant locations used by I', the effect of these DLX^{data}–rules applications to I' in C^{data} is the same as that of the DLX^p–rules in IC^p and in particular the values of the result locations of I' computed in IC^p and IC^{data} coincide. From the DLX^p*–Lemma* it follows that also IC and IC^{data} compute the same result.

For *c)* assume we have instructions I, I' in C^{data} satisfying (i). By the assumption on jump/branch instructions we know that the following holds: a) $I \in ALU \cup SET \cup LOAD \cup JLINK \cup \{MOVS2I\}$ and
b) $I' \neq \{MOVS2I\}$, i.e. $I' \in ALU \cup SET \cup LOAD \cup STORE \cup \{MOVI2S\}$. The reason is that only in these cases, *dest (I), fstop (I'), scdop (I')* respectively are defined (see table 3 and remember that *dest (I) = R31* for $I \in JLINK$). Therefore it is natural to distinguish three cases depending on whether the data hazard involves a memory access or not. We distinguish two subcases depending on the distance between data dependent instructions in the pipe. For each case we are going to show that the values of the result locations of I' in IC are the same as the ones produced by executing I' through the refined rules in IC^{data}. Let $MEM = LOAD \cup STORE$ and $REG = INSTRUCTION - MEM$. In going through these cases we explain also the required refinement of DLX^p–rules to DLX^{data}–rules (which are fully spelled out in the appendix).

function	**instructions**
dest (instr)	$instr \in ALU \cup SET \cup LOAD \cup JLINK \cup \{MOVS2I\}$
fstop (instr)	$instr \in ALU \cup SET \cup MEM \cup JLINK \cup \{MOVI2S\} \cup BRANCH \cup PLAINJ$
scdop (instr)	$instr \in ALU \cup SET \cup STORE$

Table 3. Domain of definition of *dest, fstop, scdop.*

4.1 Case $I \in$ REG

In this case it follows from a) that $I \in ALU \cup SET \cup JLINK \cup \{MOVS2I\}$. *dest (I)* receives its correct value, to be used by I', when it is updated in the

WB–stage of I by the value in *C1*; the latter has been copied in the *MEM*–stage of I from C where it has appeared in the *EX*–stage of I (as the result of an $ALU \cup SET$–operation or as content of *PC1* or of *IAR*). If I' enters the pipe 3 or 2 steps after I, then the *ID*–stage of I'—in which the operands of I' are read—overlaps with the *WB*–stage or with the *MEM*–stage of I during which the expected operand value is available in *C1* or C respectively.

In case I' enters the pipe one step after I, the expected operand value is computed during the *ID*–stage of I' and is available in the *EX*–stage of I' but not before. As a consequence the data hazard can be resolved in those two cases by refining the *ID*–rules *OPERAND* (for the first case) and the *EX*–rules *ALU, MOVI2S, MEM_ADDR, Pass_B_to_MDR* (for the second case).

Subcase $I \overset{2,3}{<} I'$**.** In this case the architecture can resolve the data hazard between I' and I by the following refinement of the DLX^p*–OPERAND* rule which guarantees that in case of conflict the correct value of A or B is taken from *C1* or C and not from *nthop (I')*:

if *nthop (IR)* $\in$ {*dest (IR3), dest (IR2)*}
then if *nthop (IR)* = *dest (IR3)* $\neq$ *dest (IR2)* **then** *nthReg* $\leftarrow$ *C1*
if *nthop (IR)* = *dest (IR2)* **then** *nthReg* $\leftarrow$ *C*
else *nthReg* $\leftarrow$ *nthop (IR)*

In case of two successive updates of *dest (I)*, the last one counts (due to the sequentiality of the execution of P in DLX). In the sequel we will refer to the above case distinction in the refined rule *OPERAND* by the following notation (where $nth \in \{fst,\ scd\}, fstReg = A,\ scdReg = B$):

$$C' = \begin{cases} \textit{C1} \textbf{ if } \textit{nthop (IR) = dest (IR3)} \textbf{ and } \textit{nthop (IR)} \neq \textit{dest (IR2)} \\ \\ \textit{C} \quad \textbf{if } \textit{nthop (IR) = dest (IR2)} \end{cases}$$

Reflecting the strengthening of the architecture by the rule refinement, in the DLX^{data}*–Lemma* we weaken the assumptions by enlarging the set of non relevant I'–used locations by:

Irrelev 3. $< reg, nthop(I') >$ such that $I' \notin JUMP \cup BRANCH$ and for some $I \overset{3,2}{<} I'$ with $I \in REG$ holds *nthop (I') = dest (I).*

Therefore the DLX^{data} *OPERAND* rule guarantees that the correct arguments for the *EX*–or *MEM*–stage rules of I' are loaded into A, B in both cases, *a)* when I' has no data dependency from any instruction in the pipe and *b)* in the case of data dependence on $I \in REG \wedge I \overset{3,2}{<} I'$. The price for this hazard resolution is a direct link between the register file–exits A, B and C, *C1*. For the case $I \overset{3}{<} I'$ our solution avoids the introduction of two file accesses (one for writing followed by one for reading [HP90]) per clock cycle.

Subcase $I \overset{1}{<} I'$**.** If I' immediately follows I in the pipe, then the I'–operand value, to be computed by I, comes out of the ALU and goes into C at the end of the *EX*–stage of I. Thus by *forwarding* the ALU–result as next ALU–input

directly without passing through C and A, B, the ALU is enabled to compute the EX–stage of I' with the correct arguments.

The formalization of this forwarding technique consists in a refinement of the EX–rules for the cases with can arise here for I', namely $I' \in ALU \cup SET$, $I' \in$ MEM, $I' = MOVI2S$. In each case we add to the corresponding EX–rule of DLX^p a clause which in the data hazard case provides the argument C instead of A or B respectively. This is at the expense of introducing a direct link between C and both ALU ports (for $I' \in ALU \cup SET$) and IAR (for $I' \in = MOVI2S$) and MAR and $SMDR$ (for $I' \in MEM$) together with some control logic (multiplexers) for selecting the forwarded value as the ALU input rather than the value from the register file. For example for $I' \in$ MEM we obtain the following rule refinements (both rules will be furthermore refined by an additional clause below):

MEM_ADDR	Pass_B_to_SMDR
if *opcode (IR1)* $\in LOAD \cup STORE$	**if** *opcode (IR1)* $\in STORE$
then if *fstop (IR1)* = *dest (IR2)*	**then if** *scdop (IR1)* = *dest (IR2)*
then $MAR \leftarrow val_{fst} + ival\ (IR1)$	**then** $SMDR \leftarrow val_{scd}$
else $MAR \leftarrow A + ival\ (IR1)$	**else** $SMDR \leftarrow B$

$$\textbf{where } val_{nth} = \begin{cases} C & \textbf{if } nthop\ (IR1) = dest\ (IR2) \\ nthReg & \textbf{otherwise} \end{cases}$$

and $nth \in \{fst,\ scd\}, fstReg = A,\ scdReg = B.$

Similarly one proceeds for the refinements of the rules ALU and $MOVI2S$, see the DLX^{data} appendix. Since the refinement of these EX–rules solves the data conflict under study, we add the following non relevant I'–used locations:

Irrelev 4. $< reg, nthop(I') >$ such that for some $I \overset{1}{<} I'$ with $I \in REG$ one of the following holds: *a) opcode (I') $\in ALU \cup SET$, iop (opcode (I')) = true, dest (I) = fstop (I'), nth = fst;*
b) opcode (I') $\in ALU \cup SET$, iop (opcode (I')) = false, dest (I) = nthop (I');
c) opcode (I') $\in MEM \cup \{MOVI2S\}$, dest (I) = fstop (I'), nth = fst;
d) opcode (I') $\in STORE$, dest (I) = scdop (I'), nth = scd.

Therefore the refined EX–rules of DLX^{data} provide the correct arguments for the EX–stage rules of I' in both cases, through A, B when I' has no data dependency on any instruction in the pipe, and through the forwarded freshly computed I–result in the data dependency case $I \in REG$ and $I \overset{1}{<} I'$.

4.2 Case $I \in$ MEM and $I' \in$ REG

$I \in MEM$, $I' \in REG$ and (i), (a), (b) above yield $I \in LOAD$ **and** $I' \in ALU \cup SET \cup \{MOVI2S\}$. The value val loaded by an instruction I is available only at the end of I's MEM–stage, namely in $LMDR$. Therefore non–MEM–instructions I' which enter the pipe 3 or 2 steps later than I and use val as operand, can grep it from $LMDR$ in their ID or EX–stage respectively. As for the case $I \in REG$, it suffices to refine the rule $OPERAND$ and the relevant EX–stage rules (here

ALU and $MOVI2S$) furthermore. If however I' enters the pipe immediately after I, then the pipeline has to be stopped for one stage, starting at the latest just before the EX–stage of I', in such a way that after the pipeline takes off again, I' can grep from $LMDR$ the value I meantime has loaded there.

Subcase $I \overset{3,2}{<} I'$**.** For the refinement of the $OPERAND$–rule, making use of our abbreviated notation above it suffices to refine C' by adding the case of data dependency of the ante–ante–preceding instruction:

$$C' = \begin{cases} C1 & \textbf{if } nthop\ (IR) = dest\ (IR3) \quad last\ modification\ in \\ & \textbf{and } nthop\ (IR) \neq dest\ (IR2) \ ante-ante-preceding \\ & \textbf{and } opcode\ (IR3) \notin LOAD \quad not\ load\ instr \\ \\ \overline{LMDR} & \textbf{if } nthop\ (IR) = dest\ (IR3) \quad last\ modification\ in \\ & \textbf{and } nthop\ (IR) \neq dest\ (IR2) \ ante-ante-preceding \\ & \textbf{and } opcode\ (IR3) \in LOAD \quad load\ instr \\ \\ C & \textbf{if } nthop\ (IR) = dest\ (IR2) \quad last\ modification\ in \\ & \qquad\qquad\qquad\qquad\quad ante-preceding\ instr \end{cases}$$

where $nth \in \{ fst, scd \}$, $fstReg = A$, $scdReg = B$, $\overline{LMDR} = \overline{opcode(IR3)}\ (LMDR)$. Similarly an additional clause is introduced in the preceding refinement for the rules ALU, $MOVI2S$ for which we refine the definition of val_{nth} as follows (see the DLX^{data} appendix for details):

$$val_{nth} = \begin{cases} C & \textbf{if } nthop\ (IR1) = dest\ (IR2) \\ \\ \overline{LMDR} & \textbf{if } nthop\ (IR1) = dest\ (IR3) \textbf{ and } opcode\ (IR3) \in LOAD \\ & \textbf{and } nthop\ (IR1) \neq dest\ (IR2) \\ \\ nthReg & \textbf{otherwise} \end{cases}$$

where $nth \in \{ fst, scd \}$, $fstReg = A$, $scdReg = B$.

This further refinement of the rules $OPERAND$, ALU, $MOVI2S$ comes together with adding the following nonrelevant locations.

Irrelev 5. $< reg, nthop(I') >$ such that for some $I \in LOAD$ with $I' \in REG - (JUMP \cup BRANCH)$ **and** $nthop\ (I') = dest\ (I)$ one of the following holds:
a) $I \overset{3}{<} I'$; **b)** $I \overset{2}{<} I'$, $iop\ (opcode\ (I')) = true$, $nth = fst$; **c)** $I \overset{2}{<} I'$, $iop\ (opcode\ (I')) = false$.

Therefore the furthermore refined ID–rule $OPERAND$ of DLX^{data} guarantees that the correct arguments for the EX–or MEM–stages rules for I' are loaded into A, B in case of non data dependency of I', but also in the data dependency case with an $I \overset{3}{<} I'$, $I' \in REG$, $I \in LOAD$; the refined EX–rules ALU, $MOVI2S$ provide the correct arguments for the EX–stage rule applications through A, B (in case of no data conflict) or through the forwarded value freshly loaded by I in case of data dependence on $I \overset{2}{<} I'$, $I' \in REG$, $I \in LOAD$.

Subcase $I \overset{1}{<} I'$**.** In this case the pipelined execution of I' (and therefore also of later instructions) has to be stopped at the latest just before the *EX*–stage of I', until the value to be loaded by I becomes available, namely in *LMDR*. It is common practice to add a *pipeline interlock* which detects this situation and stops the pipelining until the situation has been resolved. We formalize this by introducing a new function *load_risk*, defined by:

opcode (IR2) $\in$ *LOAD* **and** *reg (IR1)* $\in$ *REG* $-$ (*JUMP* $\cup$ *BRANCH*)
and *dest (IR2)* $\in$ {*fstop (IR1) , scdop (IR1)*}.

By putting the rules of stage *EX, ID* and *IF* under the additional guard $\neg$*load_risk* we obtain that in case of *load_risk* they are not executed whereas the *MEM*–and *WB*–rules are executed. By adding to the *FETCH*–rule the clause *if load_risk then IR2* $\leftarrow$ *undef*, we obtain that immediately after the execution of this *FETCH*–rule the condition *load_risk* will be *false* (because *opcode (IR2)* $\in$ *LOAD* is false by *opcode (undef) = undef*) and the full pipelined execution will be resumed. At this point I' = *reg (IR1)* still holds but I has been copied by *Preserv IR2* from *IR2* to *IR3*; therefore the subcase of data dependency considered here is reduced to the previous subcase and resolved by the refined EX–rules in DLX^{data}.

By the introduction of the *load_risk* guard to the rules in *IF* $\cup$ *ID* $\cup$ *EX* and of the new *load_risk* rule, the architecture takes care of providing the right arguments for the execution of any $I' \in$ *REG* $-$ (*JUMP* $\cup$ *BRANCH*) which is data dependent on a load instruction $I \overset{1}{<} I'$, without changing the behavior for instructions without data conflict. This yields the following additional non relevant location:
Irrelev 6. $< reg, nthop(I') >$ such that $I' \in$ *REG* $-$ (*JUMP* $\cup$ *BRANCH*) **and** some $I \in$ *LOAD* satisfies $I \overset{1}{<} I' \wedge$ *nthop (I') = dest (I).*

4.3 Case *I*, *I'* ∈ MEM

Subcase $I \overset{2,3}{<} I'$**.** The data conflict can be resolved by using the *OPERAND*–rule or the once more refined *EX*–stage rules in order to provide the value loaded by I as operand for I'. The *OPERAND* rule as refined in the previous case already resolves the conflict if $I \overset{3}{<} I'$. If $I \overset{2}{<} I'$, the two *EX*–rules for the *MEM*–instruction I' are *MEM_ADDR* and *Pass_B_to_SMDR*; their refinement is obtained by including into the guard, for the forwarding case, as new disjunct *fstop (IR1) = dest (IR3)* **and** *opcode (IR3)* $\in$ *LOAD* for *MEM_ADDR* and *scdop (IR1) = dest (IR3)* **and** *opcode (IR3)* $\in$ *LOAD* for *Pass_B_to_MDR*. This yields the two final *EX*–stage rules of DLX^{data} shown in the appendix. This refinement implies introducing direct links between *LMDR* and *MAR, SMDR* and the following new non relevant locations:
Irrelev 7. *a)* $< reg, nthop(I') >$ for $I' \in$ *MEM* and some $I \in$ *LOAD* satisfying $I \overset{3}{<} I'$ and *nthop (I') = dest (I)*;

b) $< reg, fstop(I') >$ for $I' \in MEM$ and some $I \in LOAD$ satisfying $I \overset{2}{<} I'$ and *fstop (I') = dest (I)*;

c) $< reg, scdop(I') >$ for $I' \in STORE$ and some $I \in LOAD$ satisfying $I \overset{2}{<} I'$ and *scdop (I') = dest (I)*.

Subcase $I \overset{1}{<} I'$**.** The *MEM*–instruction I' can use the value loaded by the preceding instruction I in two ways, as datum to be stored (case a) or as address for the load or store operation (case b).

Case a. *dest (I) = scdop (I')*. In this case $I' \in STORE$ and the value loaded by I is needed by I' in its *MEM*–stage—during which it is available in *LMDR*. Therefore this case can be handled again by forwarding, formalized through refining the *STORE*–rule (see the DLX^{data}–appendix) at the expense of a direct link between *LMDR* and the memory input port. Since the refined rule resolves the data conflict for the case under study, the claim of the lemma follws if we add the following non relevant locations:

Irrelev 8. $< reg, scdop(I') >$ for $I' \in STORE$ **and** some $I \in LOAD$ satisfying $I \overset{1}{<} I'$ **and** *scdop (I') = dest (I)*.

Case b. *dest (I) = fstop (I')*. In this case I' needs its first operand during its *EX*–stage when the memory address is computed. But *dest (I)* is loaded into *LMDR* only during the *MEM*–stage of I so that the pipeline must be interrupted again for one clock cycle, namely we have to uphold the execution of the rules for the *EX*–stage of I' and therefore also for the two preceding stages *ID* and *IF*. This can be formalized by refining the guard *load_risk* through the additional case *dest (IR2) = fstop (IR1)* **and** *reg (IR1)* $\in$ *MEM*. Thereby the modified rules resolve the data conflict in this case, establishing the claim of the lemma with the following additional non relevant locations:

Irrelev 9. $< reg, fstop(I') >$ for $I' \in MEM$ **and** some $I \in LOAD$ satisfying $I \overset{1}{<} I'$ **and** *fstop (I') = dest (I)*.

5 Handling control hazards

We extend now DLX^{data} to a machine DLX^{pipe} which—as we will show—handles also control hazards correctly without help from the compiler.

Control hazards are those created by jump instructions (under which we subsume also branch instructions). They present two problems, namely

a) to guarantee that after fetching a jump instruction I', the next instruction which will be fetched is the one I' requires to jump to, i.e. the instruction whose address is the value of PC as updated through the execution of I' ,

b) the data dependence of a jump instruction on a preceding instruction in the pipe.

As part of our *divide and conquer* approach we have postponed these two problems up to now by *a)* assuming, for the correctness proofs, that DLX^{data}–computations are always started with the "compiled" version P^p of P into which two empty instructions are inserted after each jump instruction in P (allowing

us to use the *Jump Lemma*), and by *b)* assuming that there are no data dependent jump instructions in DLX^{data}–computations. In this section we transform DLX^{data} first to a model DLX^{ctrl} with the same functionality as DLX^{data} but which does not need any more the compilation of empty instructions after jumps. Then we refine DLX^{ctrl} to DLX^{pipe} and prove that it handles also data dependent jump instructions correctly.

5.1 Computing jump addresses in the ID phase

The problem here is to guarantee at run time that when a *JUMP* or *BRANCH* instruction I is fetched, no other instruction I' is fetched before the computation of the new value of PC, to be determined by I, is done. Since after fetching I it needs at least one clock cycle for I to compute the new value for PC [HP90], fetching has to be stopped for at least one pipe stage. One can avoid to stall the pipe for a second pipe stage by a special decoding which permits to detect jump instructions immediately after the *IF*–stage, combined with anticipating the computation of the new PC–value in the *ID*–stage (instead of the *EX*–stage used in DLX^{data}). As effect we will obtain that in DLX^{ctrl}, one pipe stage after the *IF*–stage of a jump instruction I, the value of PC is already the correct PC result value of I.

Formally we replace the *EX*–rules *JUMP, BRANCH* and the PC–updating part of *TRAP* in DLX^{data} by new *ID*–rules which are obtained by substituting *IR1, PC1, A* by *IR, PC, fstop (IR)* respectively. The *IAR*–updating part $TRAP_{IAR}$ of *TRAP* and the *LINK*–rule remain in *EX*–stage, because they update result locations different from PC whose computation needs not to be changed in going to DLX^{ctrl}. The zero–test in *BRANCH*–instructions can be done without using the ALU by relying upon the usual standard output of registers. (See below for one more addition to the *BRANCH*–rule.)

The *FETCH*–rule of DLX^{data} is refined by introducing an additional guard *pc_risk* [21] which prevents *IR* and PC to be updated in case a jump instruction, fetched one clock cycle ago, triggers the correct update of PC through one of the new *ID*–stage rules *JUMP, BRANCH* or $TRAP_{PC}$. In this case *IR* is set to *undef* so that in the next clock cycle *pc_risk* will be false and the *FETCH*–rule will have again the same effect in DLX^{data} and in DLX^{ctrl}. We define *pc_risk* as *opcode (IR)* $\in$ *JUMP* $\cup$ *BRANCH* and delete the guard $\neg$*jumps* in the *FETCH*–rule. Since only the rules $TRAP_{IAR}$ and *LINK* in DLX^{ctrl} still use *PC1*, we can replace the DLX^{data}–rule for preservation of PC by its else-branch $PC1 \leftarrow PC$. For the complete rule set of DLX^{ctrl} see the appendix.

The DLX^{ctrl} **Correctness Theorem** is the same as for DLX^{data} with C^{data} replaced by the corresponding computation C^{ctrl} of P by DLX^{ctrl}. The proof is by reduction to the DLX^{data} correctness theorem and relies upon the fact that corresponding applications of homonymous rules in DLX^{data} and DLX^{ctrl} compute the same result. (We consider the update guarded by $\neg$*jumps*

[21] Using this guard (and similar guards load-update-risk etc. below) is similar to the introduction of SW-constraints in [TaKu95].

in the *FETCH*–rule of DLX^{data} as homonymous to the corresponding new update in the *FETCH*–rule of DLX^{ctrl}.) The proof follows by induction on the lenght n of C from the following analogue of the DLX^{data}–Lemma.

DLX^{ctrl}–**Lemma**. *Let P, P^p, C, C^{data}, IC_n, IC_n^{data} be as in the DLX^{data}–Lemma and let IC_n^{ctrl} be the nth instruction cycle in the computation C^{ctrl} of P by DLX^{ctrl}.*
a) If C^{ctrl} (and therefore also C^{data}) is free of occurrences of jump or branch instructions which are data dependent on any instruction in the pipe , then IC_n^{ctrl} and IC_n^{data} (and therefore also IC_n) are instruction cycles for the same DLX–instruction I and start with the same values for the relevant locations used by I.

Let IC, IC^{data}, IC^{ctrl} be instruction cycles for any I in C, C^{data}, C^{ctrl} resp. which start with the same values for the relevant locations used by I. Then the following two properties hold:
b) If I is not data dependent on any instruction in the pipe, then IC^{data} and IC^{ctrl} compute the same result, namely the result of the computation of I in IC.
c) If $I \notin JUMP \cup BRANCH$ is data dependent on some $I' \overset{1,2,3}{<} I$, then IC^{ctrl} and IC^{data} (and therefore IC) compute the same result.

Proof: The proof is by induction on n. For $n=0$ the claim holds because C and C^{data}, C^{ctrl} are initialized correspondingly. For the inductive step of *a)* the proof for IC_n^{ctrl} and IC_n^{data} goes along the same lines as for the DLX^p–*Lemma*.

For *b)* and *c)* we have only to show that IC^{data} and IC^{ctrl} compute the same result. By the DLX^{data}–*Lemma* we can then infer that this is the result computed by IC in DLX. Since DLX^{ctrl} has the same rules as DLX^{data} except for those which update the result location $< reg, PC >$, it suffices to check that IC^{ctrl} and IC^{data} compute the same value for $< reg, PC >$.

We distinguish two cases depending on whether the non empty instruction I fetched at the beginning of IC^{ctrl} and IC^{data} is or is not in $JUMP \cup BRANCH$.

Case 1. $I \in JUMP \cup BRANCH$
The *Jump Lemma* guarantees that I is followed in C^{data} by two empty instructions and that *PC1* is updated in the stage ID(I) by the correct value to be used in the stage EX(I) for the computation of the new *PC*–value. As a result of the execution of I in C^{data} the new value to be computed for the register *PC* is ready after the rules for the EX(I) pipe stage have been executed. This value is the correct value because by assumption I is not data dependent on any instruction in the pipe, therefore in stage ID(I) in IC^{data} the correct values are loaded into A and B and then used in stage EX(I) together with *reg (PC1)* to compute the new value by which *PC* is updated in this stage. The two empty instructions which follow any jump or branch instruction in P^p guarantee that during the *ID*–stage of I, $\neg jumps$ holds so that *PC* is again updated by *next (PC)* and therefore *IR* is updated in stage ID(I) and EX(I) with *undef* (when the *FETCH* rule is applicable at all), thus "stalling" the pipe for two stages.

In C^{ctrl} the correct next *PC*–value is ready after the rules for the pipe stage

ID(I) have been executed; indeed the new *JUMP–, TRAP–* and *BRANCH*–rules update the register PC in stage ID(I) by the correct value, due to the assumption that I is not data dependent on any instruction in the pipe. (Remember the assumption made when defining P^p that if l is the value of PC from where I has been fetched, then the value of *next (l)* in DLX—which is used in DLX^{ctrl} through PC as basis for the computation of the new value to which PC is then updated in EX(I)—coincides in DLX^{data} with the value *next (next (l'))* where *l' = next (l)* in DLX^{data}. Through *PC1* this value is used in DLX^{data} as basis for the computation of the same new value of PC).

$TRAP_{IAR}$ and the *LINK* rule are the only ones in DLX^{ctrl} which still use *PC1*. Since the DLX^{ctrl}–computations start with P instead of P^p, $TRAP_{IAR}$ and *LINK* need the value to which PC was updated when I was fetched. Therefore the simple copying rule $PC1 \leftarrow PC$ in DLX^{ctrl} provides the correct value which in DLX^{data} was provided by *next (next (next(PC)))*.

The guard *pc_risk* in the *FETCH*–rule prevents, in the case under consideration, a possibly inconsistent update of PC by *next (PC)* and updates IR by *undef*. This guarantees that except for copying *undef* into IR_i, no rule is applicable in ID(*undef*), EX(*undef*), MEM(*undef*), WB(*undef*).

Case 2. $I \notin JUMP \cup BRANCH$
Since I is not empty, by the *Jump Lemma* in the two previous clock cycles no jump instructions have been fetched in C^{data}; therefore the register PC is correctly updated to *next (PC)* when I is fetched in C^{data} (in which moment *not load_risk* is true). The same effect is obtained in C^{ctrl} by applying the *not pc_risk ? IF*–rule. (Since $I \notin JUMP \cup BRANCH$, in this case we need not consider the effect of the rules *JUMP, BRANCH* and *TRAP*.)

5.2 Data hazards for jump instructions

In this section we refine DLX^{ctrl} to our final model DLX^{pipe} which takes care also of jump instructions $I' \in JUMP \cup BRANCH$ with data dependence—namely *dest (I) = fstop (I')*—on an instruction I preceding I' in the pipe by 1, 2 or 3 steps. From table 3 we know that in this case

$$I \in ALU \cup SET \cup LOAD \cup JLINK \cup \{MOVS2I\}.$$

We distinguish two cases depending on the distance between I and I' in the pipe and on whether I is a *LOAD* instruction or not. For distance 3 and for distance 2 to a non–load instruction I, the *forwarding* technique can be applied; distance 2 to a load–instruction I and distance 1 create a *stall*.

Case $I \overset{3}{<} I'$ or ($I \overset{2}{<} I'$ and $I \notin$ LOAD). If I' is fetched 3 clock cycles later that I, then it reads its first operand in its *ID*–stage when I is in its *WB*–stage and has the new value for *dest (I)* available in *C1* (if $I \notin LOAD$) or in *LMDR* (if $I \in LOAD$). Therefore it suffices to forward this value—at the expense of direct

links between PC and $C1$, $LMDR$—in a refinement of the two rules for $JUMP$ and $BRANCH$ by the following additional clauses; for $JUMP$:

$$\begin{array}{l}\textbf{if } fstop\ (IR) = dest\ (IR3) \\ \textbf{then if } opcode\ (IR3) \notin LOAD \textbf{ then } PC \leftarrow C1 \\ \qquad \textbf{if } opcode\ (IR3) \in LOAD \textbf{ then } PC \leftarrow \overline{LMDR}\end{array}$$

For a short display of the jump rule we will abbreviate this as follows:

$$\textbf{if } fstop(IR) = dest(IR3) \textbf{ then } PC \leftarrow PC'$$

$$PC' = \begin{cases} C1 & \textbf{if } opcode\ (IR3) \notin LOAD \\ \overline{LMDR} & \textbf{if } opcode\ (IR3) \in LOAD \end{cases}$$

where $\overline{LMDR} = \overline{opcode(IR3)}(LMDR)$. In the $BRANCH$–rule we add the clause:

$$\begin{array}{l}\textbf{if } fstop\ (IR) = dest\ (IR3) \textbf{ then if } reg\ (PC') = 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } PC \leftarrow PC +_{PC}\ \ ival(IR)\end{array}$$

The same forwarding technique allow us to cope with the data hazard in case I' is fetched 2 steps after a non–load instruction I on which it depends. In this case the expected new value of *dest (I)* can be forwarded from C for use in $JUMP$ and $BRANCH$ which are therefore refined once more as follows:

$$\begin{array}{l}\boxed{\text{BRANCH}}\ \textbf{if } not\ load\_risk \\ \qquad \textbf{then if } opcode\ (IR) \in BRANCH \\ \qquad\qquad \textbf{then if } fstop\ (IR) \in \{dest\ (IR3),\ dest\ (IR2)\} \\ \qquad\qquad\qquad \textbf{then if } reg\ (PC') = 0 \\ \qquad\qquad\qquad\qquad \textbf{then } PC \leftarrow PC +_{PC}\ \ ival(IR) \\ \qquad\qquad\qquad \textbf{else if } reg\ (fstop\ (IR)) = 0 \\ \qquad\qquad\qquad\qquad \textbf{then } PC \leftarrow PC +_{PC} ival\ (IR)\end{array}$$

$$\begin{array}{l}\boxed{\text{JUMP}}\ \textbf{if } not\ load\_risk \\ \qquad \textbf{then if } opcode\ (IR) \in PLAINJ \cup JLINK \\ \qquad\qquad \textbf{then if } iop\ (opcode\ (IR)) = true \\ \qquad\qquad\qquad \textbf{then } PC \leftarrow PC +_{PC} ival\ (IR) \\ \qquad\qquad \textbf{else if } fstop\ (IR) \in \{dest\ (IR3),\ dest\ (IR2)\} \\ \qquad\qquad\qquad \textbf{then } PC \leftarrow PC' \\ \qquad\qquad\qquad \textbf{else } PC \leftarrow fstop\ (IR)\end{array}$$

$$\textbf{where } \quad PC' = \begin{cases} C1 & \textbf{if } fstop\ (IR) = dest\ (IR3) \quad last\ modification\ in \\ & \textbf{and } fstop\ (IR) \neq dest\ (IR2)\ ante - ante - preceding \\ & \textbf{and } opcode\ (IR3) \notin LOAD \quad not\ load\ instr \\ \\ \overline{LMDR} & \textbf{if } nthop\ (IR) = dest\ (IR3) \quad last\ modification\ in \\ & \textbf{and } fstop\ (IR) \neq dest\ (IR2)\ ante - ante - preceding \\ & \textbf{and } opcode\ (IR3) \in LOAD \quad load\ instr \\ \\ C & \textbf{if } fstop\ (IR) = dest\ (IR2) \quad last\ modification\ in \\ & \qquad\qquad\qquad\qquad\qquad\quad ante - preceding\ instr \end{cases}$$

These refined rules guarantee that DLX^{ctrl} provides the correct argument for the branching test and also the correct PC–value the machine has to jump to, even in case of the data dependency considered here. This justifies the claim for the corresponding case in the DLX^{ctrl}–lemma below for which we can enlarge the set of irrelevant locations as follows:

Irrelev 10. $< reg,\ fstop(I') >$ *such that* $I' \in JUMP \cup BRANCH$ **and** *fstop (I') = dest (I) for some I satisfying* $I \overset{3}{<} I'$ **or** *(*$I \overset{2}{<} I'$ *and* $I \notin LOAD$*).*

Case ($\boldsymbol{I \overset{2}{<} I'}$ **and** $\boldsymbol{I \in}$ **LOAD) or (**$\boldsymbol{I \overset{1}{<} I'}$ **and I** $\boldsymbol{\notin}$ **LOAD).** In this case I' needs its first operand in its ID–stage when I, in its MEM–or EX–stage, is providing the expected new value in $LMDR$ or C respectively. Therefore the pipe has to be stopped for one clock cycle to prevent the ID–stage of I' and the preceding IF–stage from proceeding further. This can be done by putting the IF–and ID–rules under an additional guard *pc_data_risk: BOOL* which formalizes this case[22], and by adding the new rule **if** *pc_data_risk* **then** $IR1 \leftarrow undef$. We incorporate this additional rule into the refined $FETCH$–rule. We prove now that this refinement resolves the data hazard between I' and I.

After I' has been fetched, *pc_data_risk* is true. Therefore during the following clock cycle, the rules for the pipe stage of I (MEM or EX respectively) and in the first case also for the instruction preceding I in the pipe are executed, but $FETCH$ loads *undef* into *IR1*, keeping IR and PC unchanged, and none of the ID–rules can fire; moreover *IR2* is loaded with *IR1*—which in the case under study is a non–load instruction. We show now that as a result of that, *pc_data_risk* becomes false after one clock cycle: *reg (IR1) = undef* implies *dest (reg (IR1)) = undef* $\neq$ *fstop (reg (IR))*, so that the second or–condition of *pc_data_risk* is not satisfied; by $IR2 \notin LOAD$ also the first or–condition of *pc_data_risk* not satisfied. Therefore the pipeline restarts and the data dependency has developed into a conflict which has been dealt with already in the preceding case. This establishes the corresponding case in the proof of the DLX^{ctrl}-lemma below for which we enlarge the set of irrelevant locations as follows, anticipating already the next subcase $I \overset{1}{<} I'$ and $I \in$ LOAD:

Irrelev 11. $< reg,\ fstop(I') >$ *such that* $I' \in JUMP \cup BRANCH$ **and** *fstop (I') = dest (I) for some I satisfying* ($I \overset{2}{<} I'$ **and** $I \in LOAD$) **or** $I \overset{1}{<} I'$.

Case $\boldsymbol{I \overset{1}{<} I'}$ **and** $\boldsymbol{I \in}$ **LOAD.** In this case I' has to wait two clock cycles during which I can load the needed value. The pipelining is stopped in this case

[22] i.e. *pc_data_risk* = *opcode (IR)* $\in BRANCH \cup JUMP$ **and** *[(fstop (IR) = dest (IR2)* **and** *opcode (IR2)* $\in LOAD$*)* **or** *fstop (IR) = dest (IR1)].* Anticipating the next subcase, we have formulated the condition *fstop (IR) = dest (IR1)* for both subcases, namely *reg (IR1)* $\notin LOAD$ or *reg (IR1)* $\in LOAD$.

for two clock cycles during which *pc_data_risk* is true (first through its second or–clause, then through its first clause.)

This concludes the upgrade DLX^{ctrl} of DLX^{pipe} , see the appendix. We can prove now our main theorem by induction over the given DLX–computation using the following lemma and the DLX^{ctrl}–Correctness Theorem.

DLX^{pipe}–**Lemma.** *Let P, C, C^{ctrl}, IC_n, IC_n^{ctrl} be as in the DLX^{ctrl}–Lemma and let IC^{pipe} be the n–th instruction cycle in the computation C^{pipe} of P by DLX^{pipe}.*
a) IC_n^{ctrl} and IC_n^{pipe} (and therefore IC_n) are instruction cycles for the same DLX–instruction I' and start with the same values for the relevant locations used by I'.

Let IC, IC^{ctrl}, IC^{pipe} be instruction cycles for any I in C, C^{ctrl}, C^{pipe} respectively which start with the same values for the relevant locations used by I'. Then the following two properties hold:
b) If I' is not data dependent on any I in the pipe or $I' \notin JUMP \cup BRANCH$ is data dependent on some $I \overset{1,2,3}{<} I'$, then IC^{pipe} and IC^{ctrl} (and therefore IC) compute the same result.
c) If $I' \in JUMP \cup BRANCH$ is data dependent on some $I \overset{1,2,3}{<} I'$, then IC^{pipe} and IC compute the same result.

Proof. The proof of the lemma is by induction on n. For $n = 0$ the claim follows from the assumption that C, C^p, C^{ctrl} are initialized correspondingly. For the inductive step, *a)* is proved as in the *DLX^p–Lemma*.

b) follows from the *DLX^{ctrl}-Lemma* and the conservativity of the extension DLX^{pipe} of DLX^{ctrl}; namely the same branches are taken, in the rules of C^{ctrl} and of C^{pipe}, by all instructions $I' \notin JUMP \cup BRANCH$ which are data dependent on some $I \overset{1,2,3}{<} I'$ and also by instructions I' which depend on no other instruction in the pipe. This is the case because in refining DLX^{ctrl} to DLX^{pipe}, only the rules *JUMP* and *BRANCH* have been extended, and only for a data dependence case, and because the additional guard *pc_data_risk*, which has been introduced for the rules in *IF* and in *ID*, does concern only instructions in $JUMP \cup BRANCH$ with a data dependence.

For *c)* we distinguish the three possible cases, namely that for some I, *a)* $I \overset{3}{<} I'$ or ($I \overset{2}{<} I'$ and $I \notin LOAD$), *b)* ($I \overset{2}{<} I'$ and $I \in LOAD$) or ($I \overset{1}{<} I'$ and $I \notin LOAD$), *c)* $I \overset{1}{<} I'$ and $I \in LOAD$. For each case we have shown above that the values of the result locations, as produced by executing I' in IC^{pipe} and IC respectively, are the same; in fact as part of the explanation of the refinement of the DLX^{ctrl}–rules to the DLX^{pipe}–rules we have proved that the data hazard is resolved correctly by the DLX^{pipe}–refinement of the *JUMP* and *BRANCH* rules (and the additional guard *pc_data_risk* for the rules in *IF* and *ID*) and that the result of executing I' in DLX^{pipe} coincides with the result of executing I' in IC.

Conclusion. We have developed a practical method to handle aspects of modern processor design which are most susceptible to errors. Our method supports modular design and analysis techniques and provides the possibility to pinpoint design errors at an early stage. The models we define are Abstract State Machines in the sense of Gurevich and therefore can be (implemented and) executed using ASM interpreters, providing the possibility to use the models as prototypes for simulation (see also the discussion of the falsifiability property of ASM prototypes in [Bo95]). Using ASMs is economical and quickly learnt: it requires no special theoretical training and directly supports the designer's operational view at the appropriate level of abstraction.

Our method is applicable to more complex processors than DLX, to more advanced pipelining techniques than the basic ones discussed in this paper, and to more sophisticated memory systems. Applications of the method become really interesting where mechanical tool oriented methods face intrinsic limitations (see for example the *"major bottleneck"* for model checking techniques, identified in [BD94] as the computational efficiency of logical decision procedures). We have given some hints indicating that the approach to hardware design and analysis proposed in this paper and in [BoDC95] can also be turned into a practical framework which can be used by the computer architect to formulate and analyse hardware/software co-design problems in a rigorous yet transparent way.

Acknowledgement. The first author expresses his thanks to *DIMACS* (Joint Technology Center of Rutgers and Princeton University, Bellcore and ATT Bell Labs) in New Brunswick for the hospitality during the Fall of 1995 and to *GMD-FIRST* (Institute for Computer Architecture and Software Technology of the German National Research Center for Information Technology) in Berlin for the hospitality during the Fall of 1996 when part of this work was done. We are grateful to the following colleagues for their interest in this research, for their criticism and for many stimulating and illuminating discussions: Rajeev Alur, Jörg Bormann, David van Campenhout, David Cyrluk, David Dill, Hans Eveking, Friedrich von Henke, Thomas Henzinger, Ramayya Kumar, Kent Palmer, Natarajan Shankar, Mandajan Srivas and Klaus Waldschmidt.

References

[AL95] M.Aagaard and M.Leeser. *Reasoning About Pipelines with Structural Hazards,* Springer LNCS 901, 1995.

[A95] W. Ahrendt. *Von PROLOG zur WAM, Verifikation der Prozedurübersetzung mit KIV.* Diploma thesis, University of Karlsruhe, December 1995, pp.115.

[ALLSW94] T.Arora, T.Leung, K.Levitt, T.Schubert, and P.Windley: *Report on the UCD microcoded Viper verification project.* Springer LNCS 780, 1994, 239-252.

[Be93] D.L.Beatty. *A Methodology for Formal Hardware Verification, with Application to Microprocessors.* PhD thesis, School of CS, CMU, Aug. 1993.

[Bo95] E. Börger. *Why use evolving algebras for hardware and software engineering.* in: M. Bartosek, J. Staudek, J. Wiedermann (Eds), SOFSEM'95, Springer LNCS 1012, 1995, 236–271.

[BoDC95] E. Börger and G. Del Castillo. *A formal method for provably correct composition of a real–life processor out of basic components (The APE100 reverse engineering project).* In: Proc. First IEEE International Conference *Engineering of Complex Computer Systems* (ICECCS'96). IEEE Comp. Soc. Press, Los Alamitos CA, 1995, 145-148.

[BD95] E. Börger and I. Durdanović. *Correctness of Compiling Occam to Transputer Code.* in: Computer Journal 39, 1996, 52–92.

[BM96] E. Börger and S. Mazzzanti. *A Correctness Proof for Pipelinening in RISC Architectures* DIMACS Technical Report 96-22, July 1996, pp.60.

[BR95] E. Börger and D. Rosenzweig. *The WAM - Definition and Compiler Correctness.* in: *Logic Programming: Formal Methods and Practical Applications* (C.Beierle, L.Plümer, Eds.), Elsevier Science, 1995, 20-90 (chapter 2).

[Bow87] J.P.Bowen. *Formal specifacation and documentation of microprocessor instruction sets.* In: Microprocessing and Microprogramming 21, 223–230, 1987.

[BB93] O. Buckow and J. Bormann. *Formale Spezifikation und Verifikation eines SPARC-kompatiblen Prozessors mit einem interaktiven Beweissystem,* Siemens Research and Development, München 1993.

[BD94] J.R. Burch and D.L.Dill. *Automatic verification of pipelined microprocessor control.* Conf. on Computer-Aided Verification, 1994.

[C88] J.A. Cohn. *A proof of correctness of the VIPER microprocessor: The first level.* In G.Birtwhistle and P.Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis,* pages 27-72. Kluwer Academic Publisher, 1988.

[C89] J.A. Cohn. *Correctness properties of the Viper block model: The second level.* In: G.Birtwistle, editor, *Proceedings of the 1988 Design Verification Conference.* Springer-Verlag, 1989.

[Cy93] D.Cyrluk. *Microprocessor verification in PVS: A methodology and simple example.* Technical Report SRI-CSL-93-12, SRI CS Lab, 1993.

[Cy95] D.Cyrluk. *A PVS specification, implementation and verification of DLX.* SRI, Palo Alto (Oral Communication).

[Deho94] M. Dehof. *Formale Spezifikation und Verifikation des DLX-RISC-Prozessors.* Diploma Thesis, Inst. f. Technik der Informationsverarbeitung, University of Karlsruhe, August 1994.

[DeTa94] M. Dehof and S. Tahar. *Implementierung des DLX-RISC-Prozessors in einer Standardezellen-Entwurfsumgebung,* Technical Report No. SBF 358-C2-9/94, Institute of Computer Design and Fault Tolerance, University of Karlsruhe, Germany, March 1994.

[G95] Y. Gurevich. *Evolving Algebras 1993: Lipari Guide.* In: Specification and validation methods, Ed. E. *Börger*, Oxford University Press, 1995.

[HMC94] E. Harcourt, J.Mauney, and T.Cook. *From processor timing specifications to static instruction scheduling.* In *Static Analysis Symposium,* September 1994.

[Hen93] J.L. Hennessy. *Designing a computer as a microprocessor: Experience and lessons from the MIPS 4000.* A lecture at the Symposium on Integrated Systems, Seattle, Washington, March, 1993.

[HP90] J.L. Hennessy and D.A. Patterson. *Computer Architecture: a Quantitative Approach* Morgan Kaufman Publisher, 1990. Revised second edition 1996.

[Her92] J. Herbert. *Incremental design and formal verification of microcoded microprocessor.* In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *Theorem Provers in Cicruit Design, Proocedings of the IFIP WG 10.2 International Working Conference, Nijmegen, The Netherlands.* North-Holland, June 1992.

[Hug95] J.Huggins. *Kermit: specification and verification.* In: Specification and validation methods, Ed. E. *Börger*, Oxford University Press, 1995.

[Hunt89] W.A. Hunt. *Microprocessor design verification.* Journal of Automated Reasoning, 5:429–460, 1989.

[J88] J.Joyce. *Formal verification and implementation of a microprocessor.* In: G.Birtwhistle and P.A.Subrahmanyan (Eds), VLSI specification, verification, and synthesis. Kluwer Ac. Press 1988.

[J89] J.Joyce. *Multi-level verification of microprocessor-based systems.* PhD thesis, Cambridge Dec. 89.

[JBG86] J.Joyce, G.Birtwistle, and M.Gordon. *Proving a computer correct in higher order logic.* TR 100, Computer Lab., University of Cambridge, 1986.

[La79] L.Lamport. *How to make a multiprocessor computer that correctly executes multiprocess programs.* IEEE Trans. on Computers, 1979, C-28,690-691.

[LC91] M.Langevin and E.Cerny. *Verification of processor-like circuits.* In: P.Prinetto and P.Camurati (Eds), *Advanced research Workshop on Correct Hardware Dwsign Methodologies*, June 1991.

[P96] C.Pusch. *Verification of Compiler Correctness for the WAM.* In: J. von Wright, J. Grundy, J. Harrison (eds.), *Theorem Proving in Higher Order Logics* (TPHOLs'96, Turku), Springer LNCS 1125, pp. 347-362, 1996.

[Ro92] A.W.Roscoe. *Occam in the specification and verification of microprocessors.* Philosophical Transactions of the Royal Society of London, Series A: Physical Sciences and Engineering, 339(1652): 137-151, Apr.15, 1992.

[SGGH91] J.B.Saxe, S.J.Garland, J.V.Guttag and J.J.Horning. *Using transformations and verification in circuit design.* TR 78, DEC System Res. Center, 1991.

[SB90] M. Srivas and M. Bickford. *Formal verification af a pipelined microprocessor.* IEEE Software, 7(5):52–64, September 1990.

[Ta95] S.Tahar. *Eine Methode zur formalen Verifikation von RISC-Prozessoren.* Fortschrittberichte VDI, Reihe 10: Informatik/Kommunikationstechnik Nr. 350, VDI-Verlag, Düsseldorf 1995, pp.XIV+162.

[TaKu93] S.Tahar and R.Kumar. *Towards a methodology for the formal hierarchical verification of RISC processors.* Proc. IEEE Int.Conf. on Computer Design (ICCD93), Cambridge/Mass; Oct.1993, pp. 58-62.

[TaKu94] S.Tahar and R.Kumar. *Implementational issues for verifying RISC–pipeline conflicts in HOL.* Springer LNCS 859, 1994, pp. 424-439.

[TaKu95] S. Tahar and R.Kumar. *A practical methodology for the formal verification of RISC processors.* FZI TR Sept. 95, Karlsruhe, pp. iii+46.

[Win90] P.J.Windley. *A hierarchical methodology for verifying microprogrammed mircoprocessors.* Proc. IEEE Symp. on Security and Privacy, Oakland, May 1990, pp. 345-357.

[Win94] P.J. Windley. *Formal modelling and verification of microprocessors.* IEEE Transactions on Computers, 1994.

[Windley94] P.J. Windley. *Specifying instruction-set architecture in HOL: a primer.* Springer LNCS 859, 1994, pp. 440-455.

[WC95] P.J. Windley and M.L. Coe. *A Correctness Model for Pipelined Microprocessors.* Springer LNCS 901.

Appeared as Invited Lecture in: in: Bowen, J.P., Hinchey, M.G., Till, D. (eds), ZUM'97: The Z Formal Specification Notation, Springer LNCS 1212 (1997), 151-187.

A The sequential machine *DLX*

FETCH

if $mode = FETCH$
then $IR \leftarrow mem\ (PC)$
$PC \leftarrow next\ (PC)$
$mode := OPERAND$

ALU **if** $mode = ALU$
then if $iop\ (opcode) = true$
then $TEMP \leftarrow ival$
else $TEMP \leftarrow B$
$mode := ALU'$

ALU'

if $mode = ALU'$
then $C \leftarrow \overline{opcode}\ (A,\ TEMP)$
$mode := WRITE\_BACK$

MEM_ADDR

if $mode = MEM\_ADDR$
then $MAR \leftarrow A + ival$
if $opcode \in STORE$
then $mode := Pass\_B\_to\_MDR$
else $mode := MEM\_ACC$

STORE

if $mode = MEM\_ACC$
$\wedge\ opcode \in STORE$
then $mem\ (MAR) \leftarrow MDR$
$mode := FETCH$

SUBWORD

if $mode = SUBWORD$
then $C \leftarrow \overline{opcode}\ (MDR)$
$mode := WRITE\_BACK$

BRANCH

if $mode = JUMPS$
$\wedge opcode \in BRANCH$
then if $reg\ (A) = 0$
then $PC \leftarrow ival + PC$
$mode := FETCH$

MOVS2I

if $mode = IAR\ \wedge\ opcode = MOVS2I$
then $C \leftarrow IAR$
$mode := WRITE\_BACK$

OPERAND

if $mode = OPERAND$
then $A \leftarrow fstop \quad B \leftarrow scdop$
$mode := new\_mode$
where $new\_mode =$
ALU **if** $opcode \in ALU \cup SET$
IAR **if** $opcode \in \{MOVS2I, MOVI2S\}$
$JUMPS$ **if** $opcode \in JUMP \cup BRANCH$
MEM_ADDR **if** $opcode \in LOAD$
$\cup\ STORE$

WRITE_BACK

if $mode = WRITE\_BACK$
then $dest \leftarrow C$
$mode := FETCH$

Pass_B_to_MDR

if $mode = Pass\_B\_to\_MDR$
then $MDR \leftarrow B$
$mode := MEM\_ACC$

LOAD

if $mode = MEM\_ACC \wedge opcode \in LOAD$
then $MDR \leftarrow mem\ (MAR)$
$mode := SUBWORD$

TRAP

if $mode = JUMPS \wedge\ opcode = TRAP$
then $IAR \leftarrow PC$
$PC \leftarrow ival$
$mode := FETCH$

JUMP

if $mode = JUMPS$
and $opcode \in PLAINJ \cup JLINK$
then if $iop\ (opcode) = true$
then $PC \leftarrow PC + ival$
else $PC \leftarrow A$
if $opcode \in PLAINJ$
then $mode := FETCH$
else $C \leftarrow PC$
$mode := WRITE\_BACK$

MOVI2S

if $mode = IAR \wedge\ opcode = MOVI2S$
then $IAR \leftarrow A$
$mode := FETCH$

B The parallel machine DLX^p

IF Let $jumps = opcode\ (IR1) \in JUMP \vee (opcode\ (IR1) \in BRANCH \wedge reg\ (A) = 0)$

FETCH $IR \leftarrow mem_{instr}\ (PC)$, **if** $\neg jumps$ **then** $PC \leftarrow next\ (PC)$

ID

Preserv IR: $IR1 \leftarrow IR$

Preserv PC: $PC1 \leftarrow next\ (next\ (PC))$

OPERAND: $A \leftarrow fstop, B \leftarrow scdop$

EX

ALU **if** $opcode\ (IR1) \in ALU \cup SET$
then if $iop\ (opcode\ (IR1)) = true$
then $C \leftarrow \overline{opcode\ (IR1)}\ (A,\ ival\ (IR1))$
else $C \leftarrow \overline{opcode\ (IR1)}\ (A,\ B)$

Preserv IR1: $IR2 \leftarrow IR1$

MEM_ADDR
if $opcode\ (IR1) \in LOAD \cup STORE$
then $MAR \leftarrow A + ival\ (IR1)$

Pass_B_to_MDR
if $opcode\ (IR1) \in STORE$
then $SMDR \leftarrow B$

MOVS2I
if $opcode\ (IR1) = MOVS2I$
then $C \leftarrow IAR$

MOVI2S
if $opcode\ (IR1) = MOVI2S$
then $IAR \leftarrow A$

JUMP
if $opcode\ (IR1) \in PLAINJ \cup JLINK$
then if $iop\ (opcode\ (IR1)) = true$
then $PC \leftarrow ival\ (IR1) + PC1$
else $PC \leftarrow A$

TRAP
if $opcode\ (IR1) = TRAP$
then $IAR \leftarrow PC1$
$PC \leftarrow ival\ (IR1)$

BRANCH **if** $opcode\ (IR1) \in BRANCH$
then if $reg\ (A) = 0$
then $PC \leftarrow PC1 + ival\ (IR1)$

LINK
if $opcode\ (IR1) \in JLINK$
then $C \leftarrow PC1$

MEM

STORE **if** $opcode\ (IR2) \in STORE$
then $mem\ (MAR) \leftarrow SMDR$

LOAD **if** $opcode\ (IR2) \in LOAD$
then $LMDR \leftarrow mem\ (MAR)$

Preserv C: $C1 \leftarrow C$

Preserv IR2: $IR3 \leftarrow IR2$

WB WRITE_BACK **if** $opcode\ (IR3) \in ALU \cup SET \cup \{MOVS2I\} \cup JLINK$
then $dest\ (IR3) \leftarrow C1$

if $opcode\ (IR3) \in LOAD$
then $dest\ (IR3) \leftarrow \overline{opcode\ (IR3)}\ (LMDR)$

C Data hazards handling machine DLX^{data}

IF

FETCH **if** *not load_risk*
then $IR \leftarrow mem_{instr}\ (PC)$
if $\neg$*jumps* **then** $PC \leftarrow next\ (PC)$
else $IR2 \leftarrow undef$

jumps = *opcode (IR1)* $\in$ *JUMP* **or** *(opcode (IR1)* $\in$ *BRANCH* **and** *reg (A) = 0)*
load_risk = *opcode (IR2)* $\in$ *LOAD* **and**
[(dest (IR2) $\in$ *{fstop (IR1), scdop (IR1)}*
or *(dest (IR2) = fstop (IR1)* **and** *IR1* $\in$ *MEM)].*

ID

Preserv IR
if *not load_risk*
then $IR1 \leftarrow IR$

Preserv PC
if *not load_risk*
then $PC1 \leftarrow next(next(PC))$

OPERAND
if *not load_risk*
then if *nthop (IR)* $\in$ *{dest (IR3), dest (IR2)}*
then $nthReg \leftarrow C'$
else $nthReg \leftarrow nthop\ (IR)$

where $C' =$

C1	**if** *nthop (IR) = dest (IR3)*	*last modification in*
	and *nthop (IR)* $\neq$ *dest (IR2)*	*ante − ante − preceding*
	and *opcode (IR3)* $\notin$ *LOAD*	*not load instr*
$\overline{LMDR}$	**if** *nthop (IR) = dest (IR3)*	*last modification in*
	and *nthop (IR)* $\neq$ *dest (IR2)*	*ante − ante − preceding*
	and *opcode (IR3)* $\in$ *LOAD*	*load instr*
C	**if** *nthop (IR) = dest (IR2)*	*last modification in*
		ante − preceding instr

and *nth* $\in$ { *fst, scd* }, *fstReg = A, scdReg= B,* $\overline{LMDR} = \overline{opcode(IR3)}$ *(LMDR).*

EX

ALU
if *not load_risk* **and** *opcode (IR1)* $\in$ *ALU* $\cup$ *SET*
then if *iop (opcode (IR1)) = true*
then if *fstop (IR1) = dest (IR2)*
or [*fstop (IR1) = dest (IR3)* **and** *opcode (IR3)* $\in$ *LOAD*]
then $C \leftarrow \overline{opcode\ (IR1)}\ (val_{fst},\ ival\ (IR1))$
else $C \leftarrow \overline{opcode\ (IR1)}\ (A,\ ival\ (IR1))$
else if *dest(IR2)* $\in$ *{fstop(IR1), scdop(IR1)}*
or [*dest(IR3)* $\in$ *{fstop(IR1), scdop(IR1)}*
and *opcode(IR3)* $\in$ *LOAD*]
then $C \leftarrow \overline{opcode\ (IR1)}\ (val_{fst},\ val_{scd})$
else $C \leftarrow \overline{opcode\ (IR1)}\ (A,\ B)$

$$\textbf{where } val_{nth} = \begin{cases} C & \textbf{if } nthop\ (IR1) = dest\ (IR2) \\ \overline{LMDR} & \textbf{if } nthop\ (IR1) = dest\ (IR3) \textbf{ and } opcode\ (IR3) \in LOAD \\ & \quad \textbf{and } nthop\ (IR1) \neq dest\ (IR2) \\ nthReg & \textbf{otherwise} \end{cases}$$

MEM_ADDR

if *not load_risk*
then if $opcode\ (IR1) \in LOAD \cup STORE$
then if $fstop(IR1) = dest(IR2)$
or $[fstop(IR1) = dest(IR3)$
and $opcode(IR3) \in LOAD]$
then $MAR \leftarrow val_{fst} + ival\ (IR1)$
else $MAR \leftarrow A + ival\ (IR1)$

Pass_B_to_MDR

if *not load_risk* **and**
$opcode\ (IR1) \in STORE$
then if $scdop(IR1) = dest(IR2)$
or $[scdop(IR1) = dest(IR3)$
$\wedge\ opcode(IR3) \in LOAD]$
then $SMDR \leftarrow val_{scd}$
else $SMDR \leftarrow B$

MOVI2S

if *not load_risk*
then if $opcode\ (IR1) = MOVI2S$
then if $fstop\ (IR1) = dest\ (IR2)$
or $[fstop\ (IR1) = dest\ (IR3)$
and $opcode\ (IR3) \in LOAD]$
then $IAR \leftarrow val_{fst}$
else $IAR \leftarrow A$

MOVS2I

if *not load_risk*
then if $opcode\ (IR1) = MOVS2I$
then $C \leftarrow IAR$

Preserv IR1 **if** *not load_risk* **then** $IR2 \leftarrow IR1$

TRAP

if *not load_risk*
then if $opcode\ (IR1) = TRAP$
then $IAR \leftarrow PC1$
$PC \leftarrow ival\ (IR1)$

JUMP

if *not load_risk*
then if $opcode\ (IR1) \in PLAINJ \cup JLINK$
then if $iop\ (opcode\ (IR1)) = true$
then $PC \leftarrow ival\ (IR1) + PC1$
else $PC \leftarrow A$

LINK

if *not load_risk*
then if $opcode\ (IR1) \in JLINK$
then $C \leftarrow PC1$

BRANCH

if *not load_risk*
then if $opcode\ (IR1) \in BRANCH$
then if $reg\ (A) = 0$
then $PC \leftarrow PC1 + ival\ (IR1)$

MEM

STORE

if $opcode\ (IR2) \in STORE$
then if $opcode\ (IR3) \in LOAD$ **and** $dest\ (IR3) = scdop\ (IR2)$
then $mem\ (MAR) \leftarrow \overline{(LMDR)}$
else $mem\ (MAR) \leftarrow SMDR$

LOAD

WRITE_BACK

Preserv IR2

Preserv C

as in DLX^p

D Machine DLX^{ctrl} precomputing control code

IF

FETCH **if** *not load_risk* **then if** *not pc_risk*
then $IR \leftarrow mem_{instr}\ (PC)$
$PC \leftarrow next\ (PC)$
else $IR \leftarrow undef$
else $IR2 \leftarrow undef$

where *pc_risk* = *opcode (IR)* $\in JUMP \cup BRANCH$
load_risk = as in DLX^{data}.

ID

Preserv PC **if** *not load_risk* **then** $PC1 \leftarrow PC$

BRANCH
if *not load_risk*
then if *opcode* $(IR) \in BRANCH$
then if *reg* $(fstop\ (IR)) = 0$
then $PC \leftarrow PC + ival\ (IR)$

JUMP
if *not load_risk* **and**
opcode $(IR) \in PLAINJ \cup JLINK$
then if *iop* $(opcode\ (IR)) = true$
then $PC \leftarrow PC + ival\ (IR)$
else $PC \leftarrow fstop\ (IR)$

TRAP_{PC}
if *not load_risk*
then if *opcode* $(IR) = TRAP$
then $PC \leftarrow ival\ (IR)$

Preserv IR OPERAND *as in* DLX^{data}

EX

TRAP_{IAR} **if** *not load_risk*
then if *opcode* $(IR1) = TRAP$
then $IAR \leftarrow PC1$

Other rules of group EX as in DLX^{data}.

MEM

STORE LOAD Preserv C Preserv IR2 as in DLX^{data}

WB

WRITE_BACK as in DLX^{data}

E The fully pipelined machine DLX^{pipe}

IF

FETCH **if** *not load_risk*
then if *not pc_data_risk*
then if *not pc_risk*
then $IR \leftarrow mem_{instr}\ (PC)$
$PC \leftarrow next\ (PC)$
else $IR \leftarrow undef$
else $IR1 \leftarrow undef$
else $IR2 \leftarrow undef$

pc_data_risk = *opcode (IR)* $\in BRANCH \cup JUMP$ **and**
[(fstop (IR) = dest (IR2) **and** *opcode (IR2)* $\in LOAD$)
or *(fstop (IR) = dest (IR1))].*
pc_risk = *opcode (IR)* $\in JUMP \cup BRANCH$.
load_risk = *opcode (IR2)* $\in LOAD$ **and**
[(dest (IR2) $\in$ {*fstop (IR1), scdop (IR1)*}
and *IR1* $\in$ *REG - (JUMP*$\cup$*BRANCH)*
or *(dest (IR2) = fstop (IR1)* **and** *IR1* $\in$ *MEM)].*

ID

Preserv IR
if *not load_risk* **and** *not pc_data_risk*
then $IR1 \leftarrow IR$

Preserv PC
if *not load_risk* $\wedge$ *not pc_data_risk*
then $PC1 \leftarrow PC$

OPERAND
if *not load_risk* **and** *not pc_data_risk*
then if *nthop (IR) = dest (IR3)*
or *nthop (IR) = dest (IR2)*
then $nthReg \leftarrow C'$
else $nthReg \leftarrow nthop\ (IR)$

$TRAP_{PC}$
if *not load_risk* $\wedge$ *not pc_data_risk*
then if *opcode (IR) = TRAP*
then $PC \leftarrow ival\ (IR)$

BRANCH **if** *not load_risk* **and** *not pc_data_risk*
then if *opcode (IR)* $\in BRANCH$
then if *fstop (IR)* $\in$ {*dest (IR3), dest (IR2)*}
then if *reg* $(PC') = 0$
then $PC \leftarrow PC +_{PC}\ ival(IR)$
else if *reg (fstop (IR))* $= 0$
then $PC \leftarrow PC +_{PC} ival\ (IR)$

JUMP **if** *not load_risk* **and** *not pc_data_risk*
then if *opcode (IR)* $\in PLAINJ \cup JLINK$
then if *iop (opcode (IR)) = true*
then $PC \leftarrow PC +_{PC} ival\ (IR)$
else if *fstop (IR)* $\in$ {*dest (IR3), dest (IR2)*}
then $PC \leftarrow PC'$
else $PC \leftarrow fstop\ (IR)$

$$\textbf{where } PC' = C' = \begin{cases} C1 & \textbf{if } nthop\ (IR) = dest\ (IR3) \quad last\ modification\ in \\ & \textbf{and } nthop\ (IR) \neq dest\ (IR2) \quad ante-ante-preceding \\ & \textbf{and } opcode\ (IR3) \notin LOAD \quad not\ load\ instr \\ \\ \overline{LMDR} & \textbf{if } nthop\ (IR) = dest\ (IR3) \quad last\ modification\ in \\ & \textbf{and } nthop\ (IR) \neq dest\ (IR2) \quad ante-ante-preceding \\ & \textbf{and } opcode\ (IR3) \in LOAD \quad load\ instr \\ \\ C & \textbf{if } nthop\ (IR) = dest\ (IR2) \quad last\ modification\ in \\ & \qquad\qquad\qquad\qquad\quad ante-preceding\ instr \end{cases}$$

$nth \in \{fst,\ scd\}$, $fstReg = A$, $scdReg = B$, $\overline{LMDR} = \overline{opcode\ (IR3)}\ (LMDR)$.

EX

ALU

if $(not\ load\_risk)$ **and** $opcode\ (IR1) \in ALU \cup SET$
then if $iop\ (opcode\ (IR1)) = true$
then if $fstop(IR1) = dest(IR2)$
or $[fstop(IR1) = dest(IR3)$ **and** $opcode(IR3) \in LOAD]$
then $C \leftarrow \overline{opcode(IR1)}\ (val_{fst},\ ival\ (IR1))$
else $C \leftarrow \overline{opcode(IR1)}\ (A,\ ival\ (IR1))$
else if $dest\ (IR2) \in \{fstop\ (IR1),\ scdop\ (IR1)\}$
or $[dest(IR3) \in \{fstop(IR1),\ scdop(IR1)\}$
and $opcode(IR3) \in LOAD]$
then $C \leftarrow \overline{opcode\ (IR1)}\ (val_{fst},\ val_{scd})$
else $C \leftarrow \overline{opcode(IR1)}\ (A,\ B)$

MEM_ADDR

if $(not\ load\_risk)$ **and** $opcode\ (IR1) \in LOAD \cup STORE$
then if $fstop\ (IR1) = dest\ (IR2)$
or $[fstop\ (IR1) = dest\ (IR3)$ **and** $opcode\ (IR3) \in LOAD]$
then $MAR \leftarrow val_{fst} + ival\ (IR1)$
else $MAR \leftarrow A + ival\ (IR1)$

Pass_B_to_MDR

if $(not\ load\_risk)$ **and** $opcode\ (IR1) \in STORE$
then if $scdop\ (IR1) = dest\ (IR2)$
or $[scdop\ (IR1) = dest\ (IR3)$ **and** $opcode\ (IR3) \in LOAD]$
then $SMDR \leftarrow val_{scd}$
else $SMDR \leftarrow B$

MOVI2S

if $(not\ load\_risk)$ **and** $opcode\ (IR1) = MOVI2S$
then if $fstop\ (IR1) = dest\ (IR2)$
or $[fstop\ (IR1) = dest\ (IR3)$ **and** $opcode\ (IR3) \in LOAD]$
then $IAR \leftarrow val_{fst}$
else $IAR \leftarrow A$

MOVS2I **if** *not load_risk*
then if *opcode (IR1)* = *MOVS2I*
then $C \leftarrow IAR$

where $val_{nth} = \begin{cases} C & \textbf{if } nthop\ (IR1) = dest\ (IR2) \\ \overline{LMDR} & \textbf{if } nthop\ (IR1) = dest\ (IR3) \textbf{ and } opcode\ (IR3) \in LOAD \\ & \textbf{and } nthop\ (IR1) \neq dest\ (IR2) \\ nthReg & \textbf{otherwise.} \end{cases}$

TRAP$_{IAR}$
if *not load_risk*
then if *opcode (IR1)* = *TRAP*
then $IAR \leftarrow PC1$

LINK
if *not load_risk*
then if *opcode (IR1)* $\in$ *JLINK*
then $C \leftarrow PC1$

Preserv IR1
if *not_load_risk*
then $IR2 \leftarrow IR1$

MEM

Preserv IR2 $IR3 \leftarrow IR2$ Preserv C $C1 \leftarrow C$

STORE
if *opcode (IR2)* $\in$ *STORE*
then if *opcode (IR3)* $\in$ *LOAD*
and *dest (IR3)* = *scdop (IR2)*
then $mem\ (MAR) \leftarrow \overline{(LMDR)}$
else $mem\ (MAR) \leftarrow SMDR$

LOAD
if *opcode (IR2)* $\in$ *LOAD*
then $LMDR \leftarrow mem\ (MAR)$

WB

WRITE_BACK
if *opcode (IR3)* $\in ALU \cup SET \cup \{MOVS2I\} \cup JLINK$
then $dest\ (IR3) \leftarrow C1$

if *opcode (IR3)* $\in$ *LOAD*
then $dest\ (IR3) \leftarrow \overline{LMDR}$

This article was processed using the LaTeX macro package with LLNCS style