DIMACS Technical Report 96–22
July 1996

A Correctness Proof for Pipelining in RISC Architectures

by

E. Börger and *S. Mazzanti*
Dipartimento di Informatica
Corso Italia, 40
56125 Pisa, Italy
boerger, mazzanti@di.unipi.it
Fax +39(0)50-887226, Tel.-887230

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

2

ABSTRACT

We describe a technique for specifying and verifying the control of pipelined microprocessors which can be used where traditional or purely automatic methods do not scale up to complex commercial microprocessor design. We illustrate our approach through a formal specification of Hennessy's and Patterson's RISC processor DLX [HP90] for which we prove the correctness of its pipelined model with respect to the sequential model.

First we concentrate our attention on the provably correct refinement of the sequential ground model DLX to the pipelined parallel version DLX^p in which structural hazards (resource conflicts) are eliminated. Then we extend the result to the model DLX^{data} in which also data hazards for not jump instructions are treated. The next step consists of building the model DLX^{ctrl} in which control hazards are eliminated. In the last step we define DLX^{pipe} and prove that it refines DLX^{ctrl}correctly and takes care also of data hazards relative to jump instructions.

Due to the systematic use of successive refinements, which are organized around the different pipelining problems and the methods for their solution, our approach can be applied for the design-driven verification as well as for the verification-driven design of RISC cores at any level of abstraction. The proof method supports the designer's intuitive reasoning; "local" argumentations, which are typical for incremental design and for optimizations, are supported by the semantical modularity of the design and proof method. The specification method supports the descriptive part of the designer's actual work; it is general and can be applied to other microprocessors and pipelining techniques as well. Since our models come in the form of evolving algebras, they can be made executable by evolving algebra 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]. (See [Ta95] for an excellent detailed survey.)

We develop here a method which reduces the labor required to do formally supported design and verification of microprocessors by orders-of-magnitude. The method exploits the power of abstraction which is offered by the notion of evolving algebra [G95][1] ; it allows us to define a hierarchy of refinement steps each of which is focussed on a particular feature of the processor under study and comes with a correctness proof which can be followed by the human designer or reader. The guiding principle of these successively refined specifications is to mimic as closely as possible the way the designers proceed when the architecture is built out of components which guarantee the correct execution of instructions, without or with conflicts due to pipelining. We add to this incremental approach a locality principle (see the notions of projection and of relevant locations below) which yields as a by-product that we can break the proof of the properties of interest into elementary inductions and a few natural case distinctions which correspond to the different conflict types and the methods to solve them; thus we prepare the ground for a mechanical verification using sophisticated proof development systems such as PVS, HOL, IMPS, etc. or model checking systems.

This divide-and-conquer approach to design-driven formal verification has proved to be practically viable for complex systems where traditional approaches failed; see for example the proofs in [BR95] [BD95] for the correctness of compiling Prolog programs to the Warren Abstract Machine or Occam programs to the Transputer the first of which is currently in the process of being machine checked by the Karlsruhe theorem prover KIV [A95] and in parts in Isabelle [P96]. Another good example is Huggins' evolving algebra based correctness proof for the real Kermit protocol in [Hug95] which we would like to see machine-checked by a sufficiently powerful system[2]. For breaking 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 in a position to separate specification from mechanical verification concerns. One thing is to support in a rigorous manner the reasoning which accompanies the designer's work

[1] It is interesting to observe in this context the analysis which has been reported in [ALLSW94] on the UCD microcoded Viper verification project (see [C88] [C89]). It is argued there (pg. 239) that the reason for the incompleteness of the proof of Viper *"is that the jump in abstraction between the electronic block model and the specification is too great."* The proposed namely is the following: *"By introducing intermediate levels between the two extreme models, the overall proof becomes one of establishing more, but simpler, proofs."* To the statement on page 240 that *"The handcrafting of levels of abstraction is what is needed to simplify the verification of complex systems"* we want to add that this is true also for the heuristics or formalizations which have to be used for the preparation of (automatic or interactive) machine verifications or of correctness proofs which use model checking–when it comes to deal with really complex systems.

[2] The KIV group in Karlsruhe seems to be considering to undertake this effort.

and helps him to structure his objects into intellectually manageable pieces; another thing is the detailed logical encoding which is unavoidable when it comes to make the specification understandable and checkable not any more for a human user, but for a machine. Both forms of "understanding" and of "proving" have their own logic, their own necessities and their own merits. In our view only a conscious and well thought through combination of 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 and rigorously defined models which are related by lemma as stating the properties we are interested in, the proof of the desired overall behavior of the system can be split into separate, possibly mechanizable, proofs of those lemmas. Flexible and powerful systems for machine assisted verification can incorporate such hierarchical decomposition techniques; a good candidate is PVS, see remark 5.3.4.3. on page 46 in [Cy93]. We advocate a brain–AND–brawn approach (see [Bo95]) which can be used for both, design-driven post-verification and verification-driven design using on-the-fly-verification.

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. As concrete example to which we apply our method for the specification and the correctness proof we choose the standard pipelined RISC processor DLX, developed 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 for their parallel execution, 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. 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.

During the last years also theorem provers have been used to verify pipelined processors, but either the processors are simple or the verification is rather complex (see [Cy93] [Ro92] [SGGH91] [SB90] [BB93]). For two recent projects to formally verify instruction pipelines in RISC cores using HOL and PVS see [TaKu93] [TaKu94] [Ta95] [TaKu95] and [Cy95] respectively, where also the DLX processor has been chosen for a concrete illustration of the method. [3]

For an interesting new model checking application to the verification of a subset of the

[3]There are various intimate relations between Tahar's and Kumar's hierarchical HOL-specification and our refinement steps. For details see below. In a sense, Cyrluk's specification and implementation can be viewed as a PVS formalization of the semantics of (some of) the rules of our models DLX (really DLX_0) and DLX^p. However neither Cyrluk nor Tahar and Kumar have our notion of relevant locations and of result locations which allow us to structure and to localize the proof obbligations and to boil them down to the bare minimum. The crucial point is that 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. For details see section 4.

pipelined DLX see [BD94]. The main difference between this and our work is methodological. Dill's goal is an automatic verification procedure where the human intervention is confined to the development of the operational descriptions of the specification and implementation. Once the user has provided these descriptions, they are automatically compiled into transition functions which are encoded by vectors of symbolic expressions and can be computed by symbolic simulation. This yields a logical formula which is (shown to be) valid if and only if the implementation is correct with respect to the specification, and which can be checked for validity by an algorithm. Our goal is to support explicitly the rigorous but simple behavioral modelling of both the specification and the implementation and to relate the two by transparent definitions and proofs of relevant properties. Our primary concern is to support the actual design work by a simple method which can be used by the working computer architect to lay down the design steps and to reason about their effect in a rigorous, checkable and falsifiable way. With this goal in mind 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 way the corresponding parts of the sequential and the pipelined execution models. In this way we provide a conceptual framework and a method for (documenting and supporting) design and analysis which can still be applied when instead of DLX we have to deal with more complex microprocessors, more advanced pipeling techniques or more sophisticated memory systems[4]. As said above, we consider our work to be complimentary to approaches which are based on (automatic or interactive) theorem proving or model checking techniques and whose facilities we are glad to use whenever we can let machines do some work they do better than humans.

The overall structure of our design-driven refinement hierarchy is determined by the major instruction pipelining principles[5] which can be grouped following the kind of conflict they are designed to avoid: structural hazards, data hazards and control hazards. In order to keep our proofs as simple and transparent as possible, we dedicate a separate treatement to 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 instruction types are performed in parallel—an equivalent sequential DLX–subcomputation of the one instruction type under analysis. For this first refinement step we abstract from the more sophisticated data or control hazards and stalls, i.e. in the correctness proof of the parallel (DLX^p) with respect to the sequential

---

[4]The work on this paper grew out from a reverse engineering project of a parallel architecture, reported in [BoDC95], where we had to face pipelining together with the 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.

[5]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.

(DLX) execution we assume the compiler to organize the sequence of instructions in such a way that they are sufficiently independent upon entering the pipe (so-called instruction scheduling or pipeline scheduling)[6]. This allows us to concentrate in the first refinement step on the current techniques to make the pipelined version of DLX free from structural hazards, i.e. conflicts about resources which would result when multiple instructions are overlapped in execution [7].

In the second and third refinement step we show that for the refined models the assumption on data and control hazard freeness can 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 the refinement DLX^{data} of DLX^{p}—and data hazards for jumps instructions—solved in the refinement DLX^{pipe} of another intermediate model DLX^{ctrl} which refines DLX^{data}.) By doing this we show how the compiler can be freed from the obligation to provide code of sufficiently independent instructions (by reordering the latter or by constraints imposed on the original program) so that for the correctness statement of DLX^{pipe} no assumptions on software constraints or instruction ordering need to be made. We will introduce step by step the major techniques for coping with data and control hazards and prove their correctness.[8]

Thus we develop the notions and the techniques which make the following assertion into a precise mathematical theorem, amenable to mathematical and subsequently machine checked proof.

Main Theorem (Correctness of DLX^{pipe} with respect to DLX).
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}.

Let us note that at various stages of this process we have to cope with an intricate interplay of datapath refinements—i.e. pure data (signature) refinements—and refinements of control, i.e. refinements of actions which constitute the dynamics of the pipelined processor; a further complication comes from additional time requirements under which this interplay becomes effective when we have to relate parallel and pipelined computations to sequential and non–pipelined ones. The notion of evolving algebra turned out to be particularly appropriate for a transparent formalization of such time–sensitive combinations of data and action refinements, separating clearly the underlying logic of all three features (data, dynamics, time). 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

[6] See [HMC94] where instruction schedulers are derived from timing specifications for the processor.

[7] In pipelines without structural hazards there is no termination problem, i.e. it is easy to prove that every scheduled instruction will eventually be executed (enter and exit the pipe); see part a) of the proof of the DLX^{p}-lemma and the corresponding lemmas in the refined models below. The termination problem becomes more difficult, and also additional interference problems do arise, if structural hazards can occur due to out-of-order completion, super-scalar instruction issue, data-dependent control etc. [AL95] propose a conceptual scheme for proving the correctness of pipelining with structural hazards.

[8] [SB90] verify a three stage pipelined machine where no data and control hazards can occur, whereas [WC95] verify a model for a microprocessor with a five stage pipeline which presents data hazards, but no structural hazards. They give hints how to incorporate control hazards.

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.

The paper is organized as follows. In section 2 we develop the sequential ground model (datapath and control) of DLX. In section 3 we define the basic pipelining model DLX^p by introducing: *a)* for the datapath what is needed to avoid structural conflicts, and *b)* for the control the concurrent rules which are executed under lock–step parallelism characteristic for pipelined processors. In section 4 we develop the notions of *relevant locations* and of *result locations* by which we *project* the sequential computations in DLX out of the parallel computations in DLX^p; this makes our correctness assertion into a precise mathematical statement. We define in section 4 also a decomposition technique of parallel DLX^p–computations into *instruction cycles*. This will allow us to modularize the proof in such a way that the proof obligation boils down to a case analysis at the level of single (classes of) instructions[9]. This makes our method applicable also to other instruction sets than that of DLX. In section 5 and 6 we refine the model DLX^p to models DLX^{data}, DLX^{ctrl} and DLX^{pipe} and prove that they cope also with data and control hazards respectively.

For our proofs we assume the reader to be acquainted with the definition of the semantics of sequential evolving algebras in [G95]. The mere specifications of DLX, DLX^p, DLX^{data}, DLX^{ctrl} and DLX^{pipe} and the way how these evolving algebra models work can be understood however without knowing the definition of evolving algebras; it suffices to read our rules as pseudo–code over abstract data types. We therefore abstain from repeating here the definitions given in [G95].

2 The sequential ground model of DLX

In [HP90] Hennessy and Patterson define the hardware components and the instruction set of an ideal RISC architecture, called DLX. We develop in this section a simple abstract mathematical model of this architecture, modelling its datapath and its control structure. The model comes in the form of a sequential evolving algebra which will be the starting point for the stepwise formalization of the current basic pipelining principles on RISC architectures. 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.

The construction of this DLX model is straightforward and serves mainly to lay down those abstractions which will 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),

[9]A similar idea appears in [TaKu93, TaKu95] in the reduction of the conflict freeness proof to a proof that there are no dual–stage–conflicts and no dual–RAW–conflicts.

etc. We do care however not to abstract away crucial control features like the user–requested interrupt handling. 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 view), but with its refinement where it becomes already visible that each instruction is executed in stages, i.e. a sequence of three to five elementary steps (five-stages-pipelining). Obviously we have to make sure that the semantics of instruction execution in this model coincides with the semantics in the assembly language model, as proved in [M95].

2.1 Fetch-Execute cycle

The overall structure of *DLX* is characterized by the usual fetch-execute cycle, reflected by a 0-ary function *mode* with initial value *FETCH*, and other values for the different phases of the execution period (see below).
When fetching, the processor accesses the memory—at the place indicated by the content *reg (PC)* of the program counter (0-ary function) *PC*— to load from there the instruction register *IR* with the next instruction to be executed. The place where to access the memory can be calculated reading the content of the register *PC*, by applying the unary function *reg*. We abstract from the details of this memory access by using a function *mem* in the update:

$$reg\ (IR) := mem\ (reg\ (PC)).$$

For sake of brevity we will abbreviate this by suppressing the function *reg* which applied to a given register provides its contents, writing shortly:

$$IR \leftarrow mem\ (PC).$$

Clearly when an instruction is fetched the program counter must be advanced, using an abstract *next* function, and the *mode* must switch to the first phase of the execution cycle, which consists in reading the operands of the instruction in *IR*.

The preceding explanation is formalized by the following rule:

$$\boxed{\text{FETCH}}\quad \textbf{if}\ mode = FETCH\ \textbf{then}\ \begin{array}{l} IR \leftarrow mem\ (PC) \\ PC \leftarrow next\ (PC) \\ mode := OPERAND \end{array}$$

What has to be done during the execution of an instruction depends on the instruction type, which can be:

- arithmetical-logical, concerning the ALU and including the so called SET operations, i.e. simple tests on the operands (see below);
- data-transfer between the register-file of the processor and the memory (coming in three forms: loading or storing of general data and loading of interrupt information);
- jump, absolute or conditional.

The instruction type is determined by the operation encoded in the instruction which we access abstractly by a function *opcode*:

$$\textit{opcode}: \textit{INSTRUCTION} \rightarrow \textit{OPERATION}.$$

The different groups of operations have different execution phases, namely the following four:

- decoding the instruction (which includes reading the operands of the operation to be executed);
- the execution proper (using the ALU to compute a logical-arithmetical function, a memory address, an interrupt address or an instruction address);
- accessing the memory (for loading or storing);
- writing back the computed result into the final destination in the Register File (RF).

For each phase there is a corresponding group of rules, which we are going to explain now.

2.2 Reading the operands

As said above the execution of each operation starts with reading the operands. *DLX* instructions can have two operands which we decode by applying functions *fstop* and *scdop* to *IR* (where typically *IR* will be notationally suppressed) and which are assigned to the two operand registers, called A and B respectively and representing exits of the register file:

$$\textbf{if } mode = OPERAND \textbf{ then } A \leftarrow fstop,\ B \leftarrow scdop.$$

Depending upon the result of decoding, denoted by *opcode (IR)*, the *mode* will be updated to one of the following:

- ALU if *opcode (IR)* requires to compute an arithmetical or logical or a set (test) function;
- MEM_ADDR if *opcode (IR)* requires to compute a memory address for a load or store operation;
- IAR if *opcode (IR)* requires to compute an interrupt address;
- $JUMPS$ if *opcode (IR)* requires to compute the instruction address for a jump operation.

This is summarized by the following rule:

$$\boxed{\text{OPERAND}} \quad \textbf{if } mode = OPERAND \textbf{ then } \begin{array}{l} A \leftarrow fstop,\ B \leftarrow scdop \\ mode := new\_mode \end{array}$$

$$\text{where} \quad new\_mode \equiv \begin{cases} ALU & \textbf{if } opcode \in ALU \cup SET \\ MEM\_ADDR & \textbf{if } opcode \in LOAD \cup STORE \\ IAR & \textbf{if } opcode \in \{MOVS2I, MOVI2S\} \\ JUMPS & \textbf{if } opcode \in JUMP \cup BRANCH \end{cases}$$

2.3 Execution of ALU-operations

In RISC architectures the instructions are divided into two classes depending on whether their operands are taken from some register or whether they are immediately provided by the decoding of the current instruction itself; we represent this distinction by a boolean-valued function *iop* (immediate operation), and access the numerical value of the immediate operand of an immediate instruction by applying an abstract function *ival* to *IR*. This distinction becomes relevant for determining the correct second operand of a logical-arithmetical operation which might be in B (0-ary function) or equal to *ival (IR)*. In order to economize on the number of control states Hennessy and Patterson introduce an auxiliary register *TEMP* into which the correct second operand is loaded before the computation of the result of the function (in an intermediate mode say *ALU'*). We formalize this by the following rule:

$$\boxed{\text{ALU}} \quad \textbf{if } mode = ALU \textbf{ then } \begin{array}{l} \textbf{if } iop\ (opcode) = true \\ \textbf{then } TEMP \leftarrow ival \\ \textbf{else } TEMP \leftarrow B \\ mode := ALU' \end{array}$$

Once the correct operands are available in A and *TEMP*, the result of applying an ALU-function to them can be computed and be assigned to a register C which represents the entry of the register file. The operations which occur in logical-arithmetical instructions are +, -, and, or, shift, etc., including the so-called SET operations, i.e. tests of type $A < B$, $A = B$, etc. ([HP90]). Since for the analysis of the effect of pipelining we do not need to know how the ALU computes those functions, we simply write $\overline{opcode\ (IR)}$ for the not furthermore specified function which is associated to the operation code.[10]

The *DLX* architecture is of register-register type where the operands of instructions are found in registers only. Thus no ALU instruction needs an access to the memory and the mode switches from *ALU'* directly to WRITE_BACK. In WRITE_BACK mode the result is written back into the final register-file destination determined by the function *dest* applied to *IR*.

This is formalized by the following two rules:

$$\boxed{\text{ALU'}} \quad \begin{array}{l} \textbf{if } mode = ALU' \\ \textbf{then } C \leftarrow \overline{opcode}\ (A,\ TEMP) \\ \qquad mode := WRITE\_BACK \end{array} \qquad \boxed{\text{WRITE\_BACK}} \quad \begin{array}{l} \textbf{if } mode = WRITE\_BACK \\ \textbf{then } dest \leftarrow C \\ \qquad mode := FETCH \end{array}$$

The functions *fstop, scdop, iop, ival, dest* allow us to abstract from the particular instruction format.

[10] This abstraction corresponds to what Dill calls *uninterpreted functions* in [BD94], in evolving algebra terminology (static) *oracle functions* (see [Bo95]). The same abstraction from the details of the ALU functions appears also in [?, Ta95] and in various other papers cited there. This abstraction is appropriate in the present context because our focus is on the control features of pipelined versus sequential execution where the semantics of the single ALU functions does not play any role. The abstraction makes the correctness proof uniform in the ALU–operations.

2.4 Data transfer from and to memory

Data are moved between the memory and the register-file by load and store instructions. *DLX* uses two registers as interface, namely the memory address register *MAR* (holding the address of the involved memory location), and the memory data register *MDR* (holding the data to be moved from and to the memory). For the execution of a load or store operation, i.e. in mode *MEM_ADDR*, the processor first has to put the relevant memory address into *MAR*—namely *A + ival (IR)*—and then can access the memory—in mode *MEM_ACC*—for getting or putting data there. In the case of storing there is an additional intermediate step due to the fact that the item to be stored has to be passed from *B* to the memory data interface register *MDR*[11].

This is formalized in the following rule:

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

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

For the memory access there is a slight complication in the case of loading due to the fact that in the *DLX* RISC architecture the interaction with the memory is word based (see the codomain of the function *mem* in appendix F). Even if an instruction requests only a certain portion af the location addressed by *MAR*, the processor first loads the whole word into *MDR*, and then the relevant subword is put into C[12]. This is formalized by the following two rules, using an intermediate mode *SUBWORD*.

We remind the reader that after the final result has been loaded into *C*, it is written from there into the destination register in the register-file by the rule *WRITE_BACK*.

$\boxed{\text{STORE}}$
if $mode = MEM\_ACC$
and $opcode \in STORE$
then $mem(MAR) \leftarrow MDR$
  $mode := FETCH$

$\boxed{\text{LOAD}}$
if $mode = MEM\_ACC$
and $opcode \in LOAD$
then $MDR \leftarrow mem(MAR)$
  $mode := SUBWORD$

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

Note that our notion of address is completely abstract which makes our proof uniform with respect to the address space and the addressing mechanism of the architecture.

2.5 Execution of interrupt address handlings

DLX provides a special interrupt address register *IAR* with two special instructions for reading/writing access. *MOVS2I* reads the special interrupt address register *IAR* and moves

[11] In *DLX* there is no direct physical link between the registers *B* and *MDR*; the ALU is used to pass a value without change from the former register to the latter. We use the intermediate *mode*–value *Pass_B_to_MDR* to reflect that for executing STORE instructions the ALU is used twice, first to calculate the memory address and then to pass *B* to *MDR*. In the pipelined version of *DLX* this second use of the ALU is avoided by paying the price for a connection between the two registers.

[12] The work in [TaKu95] is based on bit vectors as data type which is used throughout all the abstraction levels. We leave the data type completely abstract, leaving its specification to an orthogonal refinement step which could be introduced at any level of abstraction.

its content to C (from where it will be written into the destination register in the register-file by the rule WRITE_BACK); *MOVI2S* moves the interrupt address value A by writing it into the special register *IAR*, and then proceeds to the next intruction.

$\boxed{\text{MOVS2I}}$
if *mode* = *IAR* **and** *opcode* = *MOVS2I*
then $C \leftarrow IAR$
mode := *WRITE_BACK*

$\boxed{\text{MOVI2S}}$
if *mode* = *IAR* **and** *opcode* = *MOVI2S*
then $IAR \leftarrow A$
mode := *FETCH*

2.6 Execution of jumps

Jump instructions come in two forms, for absolute and for conditional jumps. For the latter—the so called branching instructions—the processor checks whether the condition is true for the data read in A, and in that case updates *PC* by adding to its current value the immediate value *ival (IR)*. Notice that the branch condition is a simple test with the value 0, namely *reg (A) = 0*, so we can use this directly in the following *BRANCH* rule.

There are three types of absolute jumps, namely the system jump *(TRAP)*, plain jumps *(PLAINJ)* and so called jumplinks *(JLINK)*. To execute the *TRAP* instructions means to save the current value of *PC* in *IAR* and to update *PC* by *ival (IR)*. For the other jump instructions the update of *PC* depends on whether the given instruction is immediate or not: in the former case *ival (IR)* is added to *PC*, in the latter *PC* is updated by A. In case of a jumplink instruction in addition the current value of *PC* is saved into C (from where it will be written into the destination register by the rule WRITE_BACK). This is formalized by the following two rules:

$\boxed{\text{BRANCH}}$
if *mode* = *JUMPS* **and** $opcode \in BRANCH$
then if *reg* $(A) = 0$ **then** $PC \leftarrow PC + ival$
mode := *FETCH*

$\boxed{\text{TRAP}}$
if *mode* = *JUMPS*
and *opcode* = *TRAP*
then $IAR \leftarrow PC$
$PC \leftarrow ival$
mode := *FETCH*

$\boxed{\text{JUMP}}$
if *mode* = *JUMPS* **and** $opcode \in PLAINJ \cup JLINK$
then if *iop* (*opcode*) = *true*
then $PC \leftarrow ival + PC$
else $PC \leftarrow A$
if $opcode \in PLAINJ$
then *mode* := *FETCH*
else $C \leftarrow PC$
mode := *WRITE_BACK*

Hennessy and Patterson define the 31^{th} register as destination register for jumplink instructions. Therefore we stipulate *dest (IR) = R31* in case *opcode (IR)* $\in$ *JLINK*. By leaving the function *ival* unspecified, we can abstract from the details about the length of immediate values which in *DLX* varies for different groups of instructions. This concludes the definition of the *DLX* ground model which is reassumed in appendix A. In [M95] it is shown how this model can be obtained as refinement of another evolving algebra model which directly formalizes the *DLX* instruction set architecture as seen by the programmer.

3 The basic pipelining model DLX^p

In this section we refine the basic architecture model DLX to a pipelined model DLX^p in which the so called structural conflicts are avoided, i.e. conflicts about resources which could result when multiple instructions are overlapped in execution.

The idea of pipelining is simple. Five basic execution steps appear in the ground model DLX of the previous section, namely instruction fetch *(IF)*, instruction decode which includes 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)*. Ideally one can pipeline DLX by letting the processor execute during each clock cycle one different basic step—so called *pipe stage*— simultaneously for each of five instructions, including fetching a new instruction on each clock cycle. This can be realized by eliminating the sequential control by *mode* (and by replacing in the DLX rules where necessary the *mode* guards by *opcode* guards which correspond to the pipe stage of the instruction in question), and by interpreting the resulting rule system under the *lock-step parallelism* semantics of evolving algebras; under this semantics at each moment each rule whose guard is true is applied (for details of the definition see [G95]).

To make this idea work, first of all 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 of DLX being available at each step only once.

3.1 Resolving structural conflicts

The simplicity of the DLX instructions set results in a small number of structural conflicts (i.e. competition about resources) and in simple architectural (datapath and control) refinements which suffice to avoid them. Following the analysis in [HP90, ch. 6.3] there are four major groups of resources which have to be doubled so that any combination of operations can occur in simultaneously executed pipe stages on every clock cycle, namely the memory access (to fetch instructions), an addition mechanism (to increment *PC*), the memory data register (for overlapping load and store instructions), and latches for *IR, PC* and the ALU output C (to hold values which are needed later in the pipeline)[13].

We are going now to consider each of these resources and show the resulting modification of the signature and of the rules of DLX.

Instruction fetching and incrementing *PC***.** Pipelining has a crucial impact on the memory access resources. Fetching a new instruction on each clock cycle would create a

[13]The concept of simultaneous execution of multiple evolving algebra rules allows us to abstract from the distinction of pipe stages into a writing and a subsequent reading phase (see [TaKu95]). This justifies the simultaneuos execution of for example *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.

memory access conflict with load/store instructions. This can be avoided by increasing the memory bandwith. For the basic pipelined model DLX^p of DLX we introduce an additional memory access function mem_{instr} which is used only for fetching instructions and which is supposed to be a subfunction of *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. [14]

Another resource conflict which would appear at each clock cycle concerns the ALU had we to use it for incrementing the program counter *PC*. The usual solution consists in providing a separate *PC*-incrementer, namely our abstract function *next* which will be used in case *PC* is not updated by executing a jump or branch instruction (see below the definition of the macro *jumps*).

Thus we have the following new rule *FETCH* which we classify as belonging to the pipe stage set *IF*. The new rule *OPERAND*, belonging to the pipe stage set *ID*, is obtained from the *DLX* homonym by deleting the *mode* guards and updates.

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

$\boxed{\text{OPERAND}}$ $A \leftarrow fstop\ (IR),\ B \leftarrow scdop\ (IR)$

Here is the place to define the macro *jumps*, namely as:

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

This condition 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.(See below for the definition of *IR1* cointaining the value *IR* had in the previuos clock cycle).

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 modified 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 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:

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

Similarly two additional 0-ary functions *PC1* and *C1* are needed to save the values of *PC* and *C* respectively for one pipe stage, because they are subject to be updated at each clock cycle. *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* of an instruction

[14] Notice that 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 like DLX.

I a copy of the ALU output value C computed in the pipe stage EX of I (which is the case for instructions with *ALU/SET*–operations, for *JLINK* instructions and for *MOVS2I*).

Thus we have the following two preservation rules, belonging to the sets ID and MEM respectively:

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

For reasons to be explained in section 4.4 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 register MDR is the only interface between the register-file and the memory and serves for both loading and storing to the memory. In the pipelined version for DLX however 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 consequently 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 correspondingly the DLX–rules *Pass_B_to_MDR* and *MEM_ADDR*, both belonging to the set EX. The DLX–rule *MEM_ACC* is divided in DLX^p into two refined rules, one for $LOAD$ and one for $STORE$, both belonging to the set MEM.

$$\boxed{\text{MEM\_ADDR}} \quad \begin{array}{l}\textbf{if } opcode\ (IR1) \in LOAD \cup STORE \\ \textbf{then } MAR \leftarrow A + ival\ (IR1)\end{array} \qquad \boxed{\text{Pass\_B\_to\_MDR}} \quad \begin{array}{l}\textbf{if } opcode\ (IR1) \in STORE \\ \textbf{then } SMDR \leftarrow B\end{array}$$

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

Note that 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.

3.2 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. In this section we introduce two simple refinements of DLX which help to balance the lenght of the pipeline stages.

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 $TEMP$.

$$\boxed{\text{ALU}} \quad \textbf{if } opcode\ (IR1) \in ALU \cup SET \textbf{ then } \begin{array}{l}\textbf{if } iop\ (\overline{opcode\ (IR1)}) = true \\ \textbf{then } C \leftarrow \overline{opcode\ (IR1)}\ (A,\ ival\ (IR1)) \\ \textbf{else } C \leftarrow \overline{opcode\ (IR1)}\ (A,\ B)\end{array}$$

The DLX–rule $SUBWORD$ (which selects and outputs to C the required portion of the word just loaded from the memory) is incorporated into the *WRITE_BACK*–rule under the guard that the value to be written comes through a loading instruction from the memory, i.e.

has not been computed by executing an *ALU/SET, JLINK, MOVS2I* instruction in which case the value comes from *C1*. This is done at the expense of 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*.

Thus in DLX^p we have:

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

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

Note that transfering a subword of *LMDR* into a destination register can be realized without using the ALU by relying upon the usual shift functions of registers like *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. The rules are fully spelled out in the appendix B.

Note that in case of branch operations the ALU is not needed for testing equality of A to 0—this can be done using the standard exit of registers.

4 The Correctness of DLX^p

In this section we deal with the relation between the sequential *DLX* model specified in section 2 and its (simplified) pipelined implementation by DLX^p. Before starting to prove that these two models are properly related, we have to assolve the not completely trivial task to *define* this relation in a rigorous but manageable way. We therefore *a)* first define the notions which make the correctness claim for DLX^p with respect to *DLX* into a precise statement and *b)* then prove the statement. The reader who is interested only in the specification and not in the correctness proof may skip this section. Decomposing DLX^p–computations will allow us to prove the correctness theorem instructionwise, using induction and case analysis.

The notions defined below for DLX^p will be used also for the further refinements in the following sections, the main conceptual problem being the same, namely to figure out precisely how the various components of DLX^p-states, occurring at different stages of the execution of an instruction I, have to be mapped to corresponding parts in the *DLX*-state in which I is executed in the sequential model.

We define 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. This technique directly supports the way the computer architect thinks and argues about the realation between sequential and pipelined execution when the pipeline design is layed down and explained.

4.1 DLX/DLX^p–Computation

The notion of DLX/DLX^p–computation is defined by the semantics of evolving algebras as transition systems of rules which transform structures. Since the structures of the signature of DLX/DLX^p represent machine states we call them also states. We refer the reader for details to [G95] and recall here only that in every state, for all rules whose guard is satisfied in this state, all updates which appear in these rules are executed simultaneously. This reflects faithfully the clock synchronized architectural parallelism.

In this and the next section 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 we assume *reg (PC1) = reg (C1) = reg (IRi) = undef* for i=1,2,3. We will often use that *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* .

4.2 Instruction Cycles

We decompose 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 will prove the correctness of DLX^p with respect to DLX 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, i.e. the state in which I_1, I'_1, start. 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. In the next subsection we define what is meant by the result of an instruction cycle.

4.3 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 such a class becomes visible in a DLX/DLX^p–computation, namely in certain registers or memory

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* stands for the value of *reg (fstop (instr))* + *ival (instr)* at the moment of fetching *instr* in DLX).

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]. Note that *dest (instr)* = *R31* in case *instr* $\in$ *JLINK*, and *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 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.

4.4 Used Locations and Data and Control Hazard Freeness

We define now the data and control conflicts which we assume to be avoided in DLX^p for instructions which come together in a pipe.

Given an instruction–cycle for I in a given 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. Data and control

[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 try to 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.

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 [17], 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 IAR. The table also defines the critical pipe stage during which the machine has to have the correct value of that location available. A simple analysis of the DLX^p–rules shows that three cases of I' using a result of I can be discarded immediately as conflict free.

1. No conflict can arise from using $< reg, IAR >$. Stated otherwise *MOVS2I*, the only instruction which uses IAR, can never be in conflict with any preceding instruction. Indeed 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, so that between I and I' there is no conflict.

2. No conflict can arise from using a memory location. Otherwise stated, load instructions—the 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.[18]

3. No conflict can arise with respect to the use of $< reg,\ PC >$ by pairs of instructions none of which is a branching instruction with true branching condition or a jump instruction.

The preceding discussion 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* in the next section, it is convenient to distinguish whether or not the data dependence of case *a)* concerns a jump or branch instruction. This leads us to the following definition.

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 computation in DLX^p is called *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.

[17] The notion of used resources appears also in [TaKu93]

[18] We remind the reader that DLX does not support self-modifying code. See footnote 14 at page 12.

When a jump or branch instruction I is fetched, we face the problem that 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 DLX^p 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.

There is a price we have to pay for the decision to separate as much as possible the three issues involved in the correctness proof for the pipelined with respect to the sequential DLX, namely parallelism (simultaneous execution of up to five instructions), data conflicts and control conflicts. 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* (the empty instruction). 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. We note also that the value *PC1* is needed only by jump instructions and therefore we can always load *PC1* with *next (next (PC))*: in case of a jump instruction *PC1* will have the right value, otherwise we will not need the value of *PC1*. We define therefore the PC–preservation rule in DLX^p as follows[19]:

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

The Correctness Theorem for DLX^p can now be formulated as follows and will be proved in the next section; see the Appendix B for the full set of DLX^p–rules.

DLX^p **Correctness Theorem.** *Let P be an arbitrary DLX–program and 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.*

4.5 The Correctness Proof for DLX^p

In this section we prove the correctness (and completeness) of DLX^p with respect to DLX. 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

[19] There is no need to guard this update by the condition *mem (reg (PC)) = undef* because the value of *PC1* is relevant only for jump or branch instructions.

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 pipelined and their corresponding sequential states, avoiding the flushing technique used in [BD94] and [Cy95]. This 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".[20] The projection of relevant and of result locations, out of sequences of computation steps of the pipelined processor, represents the sequential execution of the instruction under investigation. It is easily seen to be semantically correct in case no potential conflict does occur; this is the essence of the proof in this section which should be machine verifiable without difficulty once the notions of relevant and of result location are defined. The following refinements of DLX^p show how the architectural modifications handle correctly the potential conflicts among simultaneously executed instructions in a pipe.

We are going to show the following:

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^p_n be the n-th instruction cycle in C, C^p respectively. a) If C^p is data hazard free, then IC_n and IC^p_n 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 and IC^p compute the same result.*

Any location l used by *instr* (as defined by table 2) 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*;

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*.

Remark. Our definition of relevance will be refined in the subsequent models by admitting as additional irrelevant locations all those where the refined architecture will take care of providing the right values for them when needed. In this way we make it explicit where and

[20] 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].

Location	**Used by *instr* in**	**Critically in stage**
<*reg, nthop*>	$ALU \cup SET \cup$ $BRANCH \cup \{MOVI2S\}$ $JUMP$-$\{TRAP\}$	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 location.

how the compiler assumptions can be weakened if the hardware is strenghthened (to solve a given type of conflicts). This illustrates the potential of evolving algebra modelling to deal with hardware/software co-design problems in a rigorous but nevertheless simple and transparent way[21]. For another example see [BoDC95].

The lemma clearly implies the theorem.

Proof: 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 critical dynamic functions *reg* and *mem*. Remember that by convention I_0, I_0' represent the initialization of C ,C^p, i.e. the result of zero computation steps. 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. In particular, 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 true branching condition nor a jump; otherwise the $n+1$–th instruction cycle in C^p starts after the third step of IC_n^p due to the following *Jump Lemma.*

Jump Lemma. *If a jump or branch instruction I is fetched in a DLX^p–computation started with P^p, 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).

Proof: By induction on the number of fetched jumps.

□

[21][TaKu95] separate the hardware part EBM from the software constraints SW_Constr for their contribution to imply the pipelining correctness property. This distinction enables the authors to split the correctness proof 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.

It remains to prove part *b)* of the DLX^p–Lemma. Since the result locations for *instr* depend on the instruction type we are led to a natural case distinction. For each case we show that through corresponding updates in IC and IC^p, the same value is computed for the result location. See tab.3 which shows the correspondence between (sets of) DLX-updates and (sets of) DLX^p-updates for the execution of given instructions and is the basis for the projection of DLX-states out of sequences of successive DLX^p-states[22].

The application of the *FETCH*–rule in the first stage of IC_{n+1} produces the correct new value *next (reg (PC))* for the result location $<reg, PC>$ for *instr* in case *instr* is neither a jump nor a branch instruction with true branching. Since *instr* is not the empty instruction, the *Jump Lemma* implies that the instruction fetched two steps ago is neither a jump nor a branching instruction with true branching condition; therefore $\neg jumps$ is satisfied and *FETCH* in IC^p produces the same value *next (reg (PC))* for the location $<reg, PC>$. (Remember the assumption that *next*, a static function, has been initialized correspondingly for C and C^p .) We will show below that also in case $instr \in JUMP \cup BRANCH$, both IC and IC^p compute the correct new value for the result location $< reg, PC >$.

We follow now the remaining rules of IC and verify for each case that the corresponding subsequence of (mostly homonymous) rules in IC^p computes the same value for the result locations of *instr* as IC. We omit mentioning of the preservation rules in IC^p and of the fact that by definition the IC^p–rules homonymous to IC–rules always operate on the correct copies of the current values of *IR, PC*.

The effect of the IC–rule *OPERAND* is obtained similarly by the application of the IC^p–rule *OPERAND* in the pipe stage ID(*instr*). By the assumption that *instr* is not data dependent on any instruction in the pipe, the value which is found in the second pipe stage of IC^p in the register *fstop (instr)* of DLX^p is the correct value for the first argument of *opcode* (*instr*) as previously computed in IC^p (induction hypothesis) or determined by its initialization (which corresponds to that of C); the same holds for the value in *scdop (instr)* in case *instr* is not an immediate operation.

The EX(*instr*)–rules in IC^p operate then on the correct arguments built up from static values like *ival (instr)* and from the values of the copies of the *instr*–used locations which in the pipe stage EX(*instr*) are available through *A, B, IAR* or *PC1*. If the value of *reg (IAR)* is used in stage EX(*instr*) by $instr = MOVS2I$ without that location being relevant, then this value is nevertheless correct because $< reg, IAR >$ has been updated in IC^p by the last occurrence of *MOVI2S* or of *TRAP* preceding *instr* in the pipe (see the discussion on *MOVESPECIAL* instructions at the beginning of 4.4). If the value of $< reg, PC >$ is used in stage EX(*instr*), then $instr \in JUMPS \cup BRANCH$ and the value of $< reg, PC1 >$ in this stage of IC^p is correct by the *Jump Lemma*: *PC1* contains the value to which *PC* has been updated in DLX when *instr* was fetched in IC.

Therefore the EX(*instr*)–rules in IC^p compute the correct values for *C, MAR, SMDR, IAR, PC* in the same way as their IC–homonyms do (using *TEMP* and *MDR*); the DLX^p–

[22]This update table is similar to the stage instructions defined in [TaKu95]. Sofiene Tahar has told us that such a detailed table appears also in [Deho94].

DLX	DLX^p	Updates
$\left.\begin{array}{l} IR \leftarrow mem(PC) \\ PC \leftarrow next(PC) \end{array}\right\} \quad \star$ $A \leftarrow fstop\ (IR)$ $B \leftarrow scdop\ (IR)$	$\left.\begin{array}{l} IR \leftarrow mem_{instr}(PC) \\ PC \leftarrow next(PC) \end{array}\right\} \quad \star\star$ $A \leftarrow fstop\ (IR)$ $B \leftarrow scdop\ (IR)$ $IR1 \leftarrow IR$	always
$TEMP \leftarrow \begin{cases} ival\ (IR) \\ \quad \textbf{if}\ iop\ (opcode\ (IR)) = true \\ B \\ \quad \textbf{if}\ iop\ (opcode\ (IR)) = false \end{cases}$ $C \leftarrow \overline{opcode\ (IR)}\ (A,\ TEMP)$ $dest\ (IR) \leftarrow C.$	$C \leftarrow \begin{cases} \overline{opcode(IR1)}\ (A, ival(IR1)) \\ \textbf{if}\ iop(opcode(IR1)) = true \\ \overline{opcode(IR1)}\ (A, B) \\ \textbf{if}\ iop(opcode(IR1)) = false \end{cases}$ $IR2 \leftarrow IR1$ $IR3 \leftarrow IR2$ $C1 \leftarrow C$ $dest\ (IR3) \leftarrow C1.$	ALU ∪ SET
$\star$ $C \leftarrow IAR$ $dest\ (IR) \leftarrow C.$	$\star\star$ $C \leftarrow IAR$ $IR2 \leftarrow IR1$ $IR3 \leftarrow IR2$ $C1 \leftarrow C$ $dest\ (IR3) \leftarrow C1.$	MOVS2I
$\star$ $IAR \leftarrow A.$	$\star\star$ $IAR \leftarrow A.$	MOVI2S
$\star$ $MAR \leftarrow A + ival\ (IR)$ $MDR \leftarrow B$ $mem\ (MAR) \leftarrow MDR.$ $MDR \leftarrow mem\ (MAR)$ $C \leftarrow \overline{opcode(IR)}\ (MDR)$ $dest\ (IR) \leftarrow C.$	$\star\star$ $MAR \leftarrow A + ival\ (IR1)$ $SMDR \leftarrow B$ $IR2 \leftarrow IR1$ $mem\ (MAR) \leftarrow SMDR.$ $LMDR \leftarrow mem\ (MAR)$ $IR3 \leftarrow IR2$ $dest\ (IR3) \leftarrow \overline{opcode(IR3)}\ (LMDR).$	STORE ∪ LOAD STORE LOAD
$\star$ **if** $reg\ (A) = 0$ **then** $PC \leftarrow ival\ (IR) + PC.$	$\star\star$ **if** $reg\ (A) = 0$ **then** $PC \leftarrow ival\ (IR1) +_{PC}\ PC1.$	BRANCH
$\star$ $PC \leftarrow \begin{cases} ival(IR) + PC. \\ \quad \textbf{if}\ iop(opcode(IR)) = true \\ A. \\ \quad \textbf{if}\ iop(opcode(IR)) = false \end{cases}$ $dest\ (IR) \leftarrow C.$	$\star\star$ $PC \leftarrow \begin{cases} ival(IR1) +_{PC} PC1. \\ \textbf{if}\ iop(opcode(IR1)) = true \\ A. \\ \textbf{if}\ iop(opcode(IR1)) = false \end{cases}$ $IR2 \leftarrow IR1$ $C \leftarrow PC1$ $IR3 \leftarrow IR2$ $C1 \leftarrow C$ $dest\ (IR3) \leftarrow C1.$	JUMPS JLINK
$\star$ $IAR \leftarrow PC$ $PC \leftarrow ival\ (IR).$	$\star\star$ $IAR \leftarrow PC1$ $PC \leftarrow ival\ (IR1).$	TRAP

Table 3: Corresponding updates for DLX/DLX^p-instructions (run projection).

rule ALU counts here as homonym to the sequence ALU ALU' in DLX. This includes the computation of the correct new value for PC in case $instr \in BRANCH$ with true branching condition and in case $instr \in JUMP$. Note that if the branching condition is not true, then PC has been updated to *next (reg (PC))* both in IC and IC^p during the first pipe stage by the $FETCH$–rule.

The MEM($instr$)–stage and the WB($instr$)–stage rules in IC^p have the same effect as their IC–homonyms (modulo using $LMDR$ and $SMDR$ instead of MDR and considering the sequence $SUBWORD$ $WRITE\_BACK$ in DLX as homonym of $WRITE\_BACK$ in DLX^p). If the value of $< mem, arg >$ is used in stage MEM($instr$) without this location being relevant, then its value in stage MEM($instr$) is nevertheless correct because it has been updated in IC^p by the last occurrence of a $STORE$ instruction for the same memory location arg (see the discussion on $STORE$ instructions followed by $LOAD$ instructions at the beginning of 4.4).
□

5 Data Hazards for non jump instructions

In this section we weaken the data hazard freeness assumption guaranteeing the correctness of the architecture by enriching the rules with three standard features, namely new hardware links coming with appropriate additional control logic (multiplexers), forwarding and stalling. We refine DLX^p to a model DLX^{data} which we can prove to work correctly also for non jump or branch instructions with data dependence on a previous instruction in the pipe. Stated otherwise we modify the hardware so that it can be proven to cope with computations where some I, I' may satisfy the following (read after write hazard) condition[23]:

(i) $I \overset{1,2,3}{<} I'$ **and** ($dest$ (I) $= fstop$ (I') **or** $dest$ (I) $= scdop$ (I'))
and $I' \notin JUMP \cup BRANCH$.

We will define DLX^{data} as a conservative extension of DLX^p, that is to say 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 will allow us to prove the correctness of DLX^{data} by a simple 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. Correspondingly, in the DLX^{data}–correctness statement below, we

[23] 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.

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 4: Domain of definition of *dest, fstop, scdop.*

will weaken the assumption 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 (because in DLX^{data} they are taken care of by the architecture). This is a typical example how we use the technique of conservative refinements together with the localization or projection technique to mimic (i.e. to support) the way the computer architect proceeds when he enriches the processor by more and more complex features like the hazard resolution considered here.

For DLX^{data}–computations we still keep the assumption that computations are started with programs P^p where each occurrence of a jump or branch instruction is followed by two empty instructions, and that:

(ii) no jump or branch instruction occurs in a pipe when it is data dependent on a previous instruction in that pipe.

This implies that for instructions I, I' satisfying (i) we can assume in this section the following:

a) $I \in ALU \cup SET \cup LOAD \cup JLINK \cup \{MOVS2I\}$.
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 4 and remember that *dest (I)* = *R31* for $I \in JLINK$).

We distinguish cases depending on whether the data hazard involves a memory access or not. We distinguish subcases depending on the distance between data dependent instructions in the pipe. When explaining the conflicts and how to solve them by refining our rules we also prove that this solution is correct. These arguments will be put together below for the proof of the DLX^{data} correctness theorem.

We want to stress that the case distinctions are due to the nature of the problem; they are characterized by a different type of solution of the conflict for each case.

Let $MEM = LOAD \cup STORE$ and $REG = INSTRUCTION - MEM = ALU \cup SET \cup BRANCH \cup JUMP \cup \{MOVI2S, MOVS2I\}$.

5.1 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).

5.1.1 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')*. Note that in case of two successive updates of *dest (I)*, the last one counts (due to the sequentiality of the execution of P in DLX).

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*)

For a shorter display of this rule we will use in the sequel the following notational abbreviation:

$\boxed{\text{OPERAND}}$
if *nthop* (*IR*) $\in$
 $\{$*dest* (*IR3*), *dest* (*IR2*)$\}$
then *nthReg* $\leftarrow C'$
else *nthReg* $\leftarrow$ *nthop* (*IR*)

where

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

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

Reflecting the strengthening of the architecture by the refinement of *OPERAND*, in the DLX^{data}–*Lemma* below we have to weaken the assumptions by enlarging the set of non relevant I'–used locations, namely 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 $I \in REG \wedge I \overset{3,2}{<} I'$.

Remark. The price for this hazard resolution is a direct link between the register file–exits A and B and C and *C1*. For the case $I \overset{3}{<} I'$ our solution avoids the introduction of two file accesses per clock cycle (the first one for writing and the second one for reading [HP90]).

5.1.2 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* [HP90] 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.

$\boxed{\text{ALU}}$ **if** $opcode\ (IR1) \in ALU \cup SET$
then if $iop\ (opcode\ (IR1)) = true$
  then if $fstop\ (IR1) = dest\ (IR2)$
    then $C \leftarrow \overline{opcode(IR1)}(val_{fst}, ival(IR1))$
    else $C \leftarrow \overline{opcode(IR1)}(A, ival(IR1))$
  else if $dest(IR2) \in \{fstop(IR1), scdop(IR1)\}$
    then $C \leftarrow \overline{opcode(IR1)}(val_{fst}, val_{scd})$
    else $C \leftarrow \overline{opcode(IR1)}(A, B)$

$\boxed{\text{MOVI2S}}$
if $opcode\ (IR1) = MOVI2S$
then if $fstop(IR1) = dest(IR2)$
  then $IAR \leftarrow val_{fst}$
  else $IAR \leftarrow A$

$\boxed{\text{MEM\_ADDR}}$
if $opcode\ (IR1) \in LOAD \cup STORE$
then if $fstop\ (IR1) = dest\ (IR2)$
  then $MAR \leftarrow val_{fst} + ival\ (IR1)$
  else $MAR \leftarrow A + ival\ (IR1)$

$\boxed{\text{Pass\_B\_to\_SMDR}}$
if $opcode\ (IR1) \in STORE$
then if $scdop\ (IR1) = dest\ (IR2)$
  then $SMDR \leftarrow val_{scd}$
  else $SMDR \leftarrow B$

$$\textbf{where } val_{nth} = \begin{cases} C & \textbf{if } nthop\ (IR1) = dest\ (IR2) \\ nthReg & \textbf{otherwise} \end{cases}$$
and $nth \in \{fst,\ scd\}, fstReg = A,\ scdReg = B.$

Since the refinement of these *EX*–rules solves the data conflict under study, we add to the non relevant I'–used locations (see the DLX^{data}–*Lemma* below) the following ones:

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} guarantee that the correct arguments for the EX–stage rules of I' are provided in both cases, through A and 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 $I \overset{1}{<} I'$.

Remark. Another frequently used implementation of *forwarding* uses buffers *(buff_val, buff_reg)* which store the value computed by the ALU and the name *dest (I)* of the Register-File–destination register where this value has to be transfered. It is straightforward to formulate the buffer rule and the refined *EX*–rules in terms of buffers.

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

From $I \in MEM$, $I' \in REG$ and (i), (a), (b) above it follows that:

$$I \in LOAD \textbf{ and } I' \in ALU \cup SET \cup \{MOVI2S\}.$$

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 at the expense of direct links between *LMDR* and the Register File-exits A and B.

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.

5.2.1 Subcase $I \overset{3,2}{<} I'$

For the refinement of the *OPERAND*–rule, making use of our abbreviated notation it suffices to refine C' as follows for the case of data dependency of the ante–ante–preceding instruction:

$$C' = \begin{cases} C1 & \textbf{if } nthop\ (IR) = dest\ (IR3) & last\ modification\ in \\ & \textbf{and } nthop\ (IR) \neq dest\ (IR2) & ante - ante - preceding \\ & \textbf{and } opcode\ (IR3) \notin LOAD & not\ load\ instr \\[1ex] \overline{LMDR} & \textbf{if } nthop\ (IR) = dest\ (IR3) & last\ modification\ in \\ & \textbf{and } nthop\ (IR) \neq dest\ (IR2) & ante - ante - preceding \\ & \textbf{and } opcode\ (IR3) \in LOAD & load\ instr \\[1ex] C & \textbf{if } nthop\ (IR) = dest\ (IR2) & last\ modification\ in \\ & & ante - preceding\ instr \end{cases}$$

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

Since in this case $I' \in ALU \cup SET \cup \{MOVI2S\}$, we need to refine only two *EX*–rules. For both we refine the definition of val_{nth} as follows:

$$val_{nth} = \begin{cases} C & \textbf{if } nthop\ (IR1) = dest\ (IR2) \\[1ex] \overline{LMDR} & \textbf{if } nthop\ (IR1) = dest\ (IR3) \textbf{ and } opcode\ (IR3) \in LOAD \\ & \textbf{and } nthop\ (IR1) \neq dest\ (IR2) \\[1ex] nthReg & \textbf{otherwise} \end{cases}$$

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

We have to anticipate that the full *EX*–rule will be under an additional guard which prevents the following two *EX*–rules from being executed for *LOAD*–instructions in *IR3* which are data dependent on the instruction in *IR2*.

$\boxed{\text{ALU}}$

if $opcode\ (IR1) \in ALU \cup SET$
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 $C \leftarrow \overline{opcode(IR1)}(A,\ B)$

$\boxed{\text{MOVI2S}}$ **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 $IAR \leftarrow A$

Since the further refinement of the rules *OPERAND, ALU, MOVI2S* solves the conflict considered in this case, we have to add to the nonrelevant I'–used locations in the DLX^{data}–*Lemma* below the following ones:

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* guarantee that the correct arguments for the *EX*–stage rule applications are provided through *A, B* (in case of no data conflict) or through the forwarded value freshly loaded by I in case of data dependence on $I \overset{2}{<} I'$, $I' \in REG$, $I \in LOAD$.

5.2.2 Subcase $I \overset{1}{<} I'$ (stall)

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 new hardware–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 as follow:

load_risk : BOOLEAN.
load_risk = 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).*

5.3 Case I, $I' \in$ MEM

As in the previous two cases we have to distinguish the two subcases whether I' enters the pipe 3 or 2 steps or only 1 step after I. By table 5 we have in this case $I \in LOAD$.

5.3.1 Subcase $I \overset{2,3}{<} I'$

Analogously to the previous two cases, the data conflict can be resolved by using refined *EX*–stage rules or the *OPERAND*–rule 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 following two refined *EX*–stage rules (at the expense of introducing direct links between *LMDR* and *MAR, SMDR*) and the new non relevant locations defined below.

$\boxed{\text{MEM\_ADDR}}$

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 $MAR \leftarrow A + ival\ (IR1)$

$\boxed{\text{Pass\_B\_to\_SMDR}}$

if *not load_risk*
then if *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 $SMDR \leftarrow B$

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).*

5.3.2 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 at the expense of a direct link between *LMDR* and the memory input port and with a corresponding addition to the set of non relevant locations. We write $\overline{LMDR}$ as abbreviation for $\overline{opcode(IR3)}(LMDR)$.

$$\boxed{\text{STORE}}\ \begin{array}{l} \textbf{if } opcode\ (IR2) \in STORE \\ \textbf{then if } opcode\ (IR3) \in LOAD \textbf{ and } dest\ (IR3) = scdop\ (IR2) \\ \qquad \textbf{then } mem\ (MAR) \leftarrow \overline{LMDR} \\ \qquad \textbf{else } mem\ (MAR) \leftarrow SMDR \end{array}$$

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 considered here, namely *dest (IR2) = fstop (IR1)* **and** *reg (IR1)* $\in$ *MEM*, and by adding the following location to the non relevant ones:

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

This concludes the definition of DLX^{data} (see the appendix C for the full set of rules).

DLX^{data} **Correctness Theorem.** *Let P be an arbitrary DLX-program and 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^{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: The proof is by induction over the given *DLX*–computation. In the inductive step we want to use the analysis made for the proof of the DLX^p*–Lemma* in case the considered instruction is not data dependent on any other instruction in the pipe. For this we need a

stronger inductive hypothesis. We therefore 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.

Relevance of locations is meant in this section as defined for DLX^{data} by the conditions Irrelev 1–9.

It is clear that the DLX^{data} Correctness Theorem follows from the DLX^{data} *Lemma*; use in case *b)* also the DLX^p*–Lemma* to infer that the result of IC^{data} is the same as that of IC.

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*; note that 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)* we proceed as follows. If I' is not data dependent on 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.

It remains to prove *c)*. Assume in C^{data} we have $I \overset{1,2,3}{<} I'$ and $I' \notin JUMP \cup BRANCH$ for some I such that *dest (I)* $\in$ {*fstop (I'), scdop (I')*}. Depending on the type of the instructions involved we distinguish three cases, namely *a)* $I \in REG$, *b)* $I \in MEM$ and I' $\in REG$, and *c)* I, $I' \in MEM$. In each case we distinguish subcases depending on whether the distance between I and I' is 3, 2 or 1 pipe stage. For each case we have to show that the values of the result locations of I', as produced by executing I' in IC^{data} and in IC respectively,

are the same. This has been done above as part of the explanation of the refinement of DLX^p–rules to DLX^{data}–rules, corresponding to each of those cases of data dependency of I' with respect to I. We have shown there that the data hazard is resolved correctly by the corresponding rule refinement and that the result of executing I' in IC^{data} coincides with the result of executing I' in IC.
□

6 Elimination of Control Hazards

Control hazards are those created by jump instructions (i.e. in *BRANCH* or *JUMP*). Sometimes they are also called branch or sequencing hazards.

They present two problems, namely *a)* data dependence of a jump instruction on a preceding instruction in the pipe, and *b)* the problem 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 newly updated through the execution of I'. As part of our *divide and conquer* approach we have avoided these two problems up to now by *a)* excluding data dependent jump instructions from DLX^{data}–computations and by *b)* assuming that the compiler inserts two empty instructions after each jump instruction in P and that DLX^{data}–computations are always started with the "compiled" version P^p of P. The latter assumption provided the possibility to use the *Jump Lemma*. In order to refine DLX^{data} to a model which can cope also with control hazards, in this section we transform DLX^{data} first to a model DLX^{ctrl} which is correct with respect to DLX^{data} but does not need any more the assumption on compilation of empty instructions after jumps. Then we refine DLX^{ctrl} to DLX^{pipe} and prove that it implements DLX correctly, also in presence of data dependent jump instructions.

6.1 No compilation of empty instructions after jumps

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. In DLX^{data} the corresponding effect was achieved because the compiler was assumed to provide an empty instruction to be fetched after I. One can avoid to stall the pipe for a second pipe stage as follows.

We use the special PC-incrementer, introduced already in section 3.1. and denoted now as $+_{PC}$, which allows us to avoid the use of the ALU for computing the new value of PC. Using in addition some special decoding which permits to detect jump instructions immediately after the *IF*–stage, we can then anticipate the computation of the new PC–value in the *ID*–stage (instead of the *EX*–stage used in DLX^p and 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 in DLX^{ctrl} which are obtained by replacing *IR1, PC1* and *reg (A)* by *IR, PC* and *reg (fstop (IR))* respectively. Note that the *IAR*–updating part $TRAP_{IAR}$ of *TRAP* as well as the *LINK*–rule remain in *EX*–stage, because they update result locations of I which are different from *PC* and whose computation needs not to be changed in going to DLX^{ctrl}. Note that the test of the condition in a *BRANCH*–instruction is a zero–test which 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 in DLX^{ctrl}).

The *FETCH*–rule of DLX^{data} is refined by introducing an additional guard *pc_risk* [24] 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 can define *pc_risk* by

$$pc\_risk \equiv opcode\ (IR) \in JUMP \cup BRANCH.$$

Given this definition it is clear that we have to delete the guard $\neg jumps$ in the *FETCH*–rule. Thus *FETCH* and *BRANCH* are defined in DLX^{ctrl} as follows:

```
[FETCH] if not load_risk
        then if not pc_risk
                then IR ← mem_instr (PC)
                     PC ← next (PC)
                else IR ← undef
        else IR2 ← undef
```

```
[BRANCH]
if not load_risk
then if opcode (IR) ∈ BRANCH
        then if reg (fstop (IR)) = 0
                then PC ← PC +_PC ival (IR)
```

Since $TRAP_{IAR}$ and *LINK* are the only rules in DLX^{ctrl} which still use *PC1*, we can replace the DLX^{data}–rule for preservation of *PC* by its else-branch *PC1* ← *PC*.

For easy comparison the complete rule set of DLX^{ctrl} is collected in the appendix D.

Relevance of locations is understood in this section as defined for DLX^{data} through the conditions Irrelev 1–9.

DLX^{ctrl} **Correctness Theorem.** *Let P be any DLX–program, C the computation of P by DLX and* C^{ctrl} *the corresponding computation of P by* DLX^{ctrl}*. If* C^{ctrl} *has no occurrences of jump instructions which are data dependent on any instruction in the pipe, then C and* C^{ctrl} *produce the same result.*

Proof: The proof is by reduction to the DLX^{data} correctness theorem and relies upon the idea that the application of homonymous rules in DLX^{data} and DLX^{ctrl} computes the same

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

result. (We consider the update guarded by $\neg jumps$ 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 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}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 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 ∪ 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* ∪ *BRANCH*.

Case 1. $I \in$ *JUMP* ∪ *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 I in P^p guarantee that during the *ID*–stage of I, *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*).

The *EX*–stage part $TRAP_{IAR}$ of the *TRAP* rule and the *LINK* rule are the only ones which still use *PC1* in DLX^{ctrl}. 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 (PC))*.

The guard *pc_risk* in the *FETCH*–rule prevents a possibly inconsistent update of *PC* by *next (PC)* and updates *IR* by *undef*. The latter update 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*.)
□

6.2 Data Dependent 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. [25] From table 4 we know that in this case

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

We distinguish now 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*. We explain now the rule refinement which will ensure that the correctness theorem holds also for DLX^{pipe}.

[25] Sofiene Tahar has told us that on architecture which is similar to our DLX^{pipe} has been implemented in [DeTa94].

6.2.1 $I \overset{3}{<} I'$ 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* we have:

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

For a short display of the full jump rule below we will abbreviate this to:

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

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

$$\textbf{and } \overline{LMDR} = \overline{opcode(IR3)}(LMDR).$$

In the *BRANCH*–rule we add the clause:

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

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:

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

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

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

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. Consequently 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 $I \notin LOAD$).*

6.2.2 $(I \overset{2}{<} I'$ and $I \in$ LOAD$)$ or $(I \overset{1}{<} I'$ and I $\notin$ LOAD$)$

If *($I \overset{2}{<} I'$ and $I \in LOAD$)* or *($I \overset{1}{<} I'$ and $I \notin LOAD$)*, then 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*, and by adding the new rule **if** *pc_data_risk* **then** *IR1* $\leftarrow$ *undef*. We incorporate this additional rule into the refined *FETCH*–rule and define *pc_data_risk* as follows:

pc_data_risk = *opcode (IR)* $\in$ *BRANCH* $\cup$ *JUMP* **and**
[(fstop (IR) = dest (IR2) **and** *opcode (IR2)* $\in$ *LOAD)* **or**
(fstop (IR) = dest (IR1))].

Note that 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*. We will prove below that in both cases DLX^{pipe} correctly resolves the data hazard between I' and I.

Assume *($I \overset{2}{<} I'$ and $I \in LOAD$)* or *($I \overset{1}{<} I'$ and $I \notin LOAD$)*. Then 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 benn dealt with already in the preceding case.

As a consequence we can enlarge the set of irrelevant locations as follows, where we anticipate 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** $I \overset{1}{<} I'$.

6.2.3 $I \overset{1}{<} I'$ and $I \in$ LOAD

If $I \overset{1}{<} I'$ and $I \in LOAD$, then I' has to wait two clock cycles during which I can load the needed value. The pipelining is stopped in this case 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 definition of DLX^{pipe} (see the appendix E for the full set of rules).

DLX^{pipe} **Correctness Theorem.** *Let P be an arbitrary DLX-program, C the computation of P by DLX and C^{pipe} the corresponding computation od P by DLX^{pipe}. Then C and C^{pipe} compute the same result.*

Proof: The proof is by induction over the given DLX–computation and follows from the following *Lemma*, using the DLX^{ctrl}–Correctness Theorem.
□

DLX^{pipe}–**Lemma.** *Let P, C, C^{ctrl}, C^{pipe}, IC_n, IC_n^{ctrl} as in the theorem and in the DLX^{ctrl}–Lemma and let IC^{pipe} be the n–th instruction cycle in C^{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 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 fact that 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 for a data dependence case and because the additional guard *pc_data_risk*, which has been introduced for the *IF* and *ID*–rules, does concern only instructions in *JUMP* ∪ *BRANCH* with a data dependence.

For *c)* we distinguish the three cases 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 to show that the values of the result locations, as produced by executing I' in IC^{pipe} and IC respectively, are the same. This has been shown above as part of the explanation of the refinement of the DLX^{ctrl} to the DLX^{pipe}–rules; we have proved there 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 *IF* and *ID*–rules) and that the result of executing I' in DLX^{pipe} coincides with the result of executing I' in IC.
□

7 Conclusion

We have shown how evolving algebras can be used to handle aspects of modern processor design which are most susceptible to errors. The approach supports modular design and analysis techniques and provides the possibility to pinpoint design errors at an early stage (see also the discussion of the falsifiability property of evolving algebra prototypes in [Bo95]). The models come in the form of evolving algebras and therefore can be (implemented and) executed using evolving algebra interpreters, providing the possibility to use the models as prototypes for simulation. The method also provides 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.

The method is economical and fast; its application requires no special theoretical training and directly supports the designer's operational view at the appropriate level of abstraction. The 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.

There is no other limitation of the method than the limit of mathematical structuring and formalization, i.e. the limits of the human capabilities of rigorous thought. Applications of the method become really interesting where mechanical tool oriented methods face intrinsic limitations (see for example the discussion of the *"major bottleneck"* for model checking techniques, identified in [BD94] as the computational efficiency of logical decision procedures). Our method can be used in such situations to decompose a given complex system into an intellectually manageable structure of components to which tool-based and in particular machine verification techniques can be applied satisfactorily.

Acknowledgement. The first author wants to express his thanks to $DIMACS$ for the hospitality during the Fall of 1995 when part of this work was done. We are grateful to the following colleagues for their interest in this work and for the stimulating and illuminating discussion: Rajeev Alur, Jörg Bormann, David van Campenhout, David Cyrluk, David Dill, Hans Eveking, Friedrich von Henke, Thomas Henzinger, Ramayya Kumar, Philipp Kutter, Kent Palmer, Natarajan Shankar, Mandajan Srivas, Sofiene Tahar and Klaus Waldschmidt.

A Rules of DLX0

$\boxed{\text{FETCH}}$

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

$\boxed{\text{EX-SET}}$

if $mode \ = \ EXECUTE$**and** $opcode \in SET$
then $dest \leftarrow cond$
$mode \ := \ FETCH$

$\boxed{\text{EX-ALU}}$

if $mode \ = \ EXECUTE$**and** $opcode \in ALU$
then $dest \leftarrow fct - value \ (reg \ (IR))$
$mode \ := \ FETCH$

$\boxed{\text{MEM}}$

if $mode = EXECUTE$
then if $opcode \in \overline{LOAD}$
then $dest \leftarrow \overline{opcode}(mem(addr))$
if $opcode \in STORE$
then $mem \ (addr) \leftarrow scdop$

$\boxed{\text{INTERRUPT}}$

if $mode = EXECUTE$
then if $opcode \in GET - INTERRUPT$
then $dest \leftarrow IAR$
if $opcode \in PUT - INTERRUPT$
then $IAR \leftarrow fstop$
$mode := FETCH$

where addr = (reg (fstop (reg (IR)))) + ival (reg (IR)).

$\boxed{\text{EX-JUMP}}$

if $mode = EXECUTE$**and** $opcode \in JUMP$
then $PC \leftarrow newpc \ (IR, \ PC)$
if $opcode \in JUMPLINK$
then $RET \leftarrow PC$
if $opcode \in TRAP$
then $IAR \leftarrow PC$
$mode \ := \ FETCH$

where $cond \equiv (reg \ (fstop \ (reg \ (IR)))) = 0.$

$\boxed{\text{EX-BRANCH}}$

if $mode = EXECUTE$**and** $opcode \in BRANCH$
then if $cond = true$
then $PC \leftarrow newpc \ (IR, \ PC)$
$mode \ := \ FETCH$

N.B.: $R1 \leftarrow R2$ stands for *reg (R1) := reg (IR2).*
For most functions the standard argument IR is suppressed in the shorthand notation. For example $opcode \in LOAD$ stands for *opcode (reg (IR))* $\in$ *LOAD*, similary $dest \leftarrow IAR$ stands for *dest (reg (IR)):= reg (IAR).*

B Rules of DLX

$\boxed{\text{FETCH}}$

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

$\boxed{\text{ALU}}$

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

$\boxed{\text{ALU'}}$

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

$\boxed{\text{MEM\_ADDR}}$

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

$\boxed{\text{STORE}}$

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

$\boxed{\text{SUBWORD}}$

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

$\boxed{\text{BRANCH}}$

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

$\boxed{\text{MOVS2I}}$

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

$\boxed{\text{OPERAND}}$

if $mode = OPERAND$
then $A \leftarrow fstop$
$B \leftarrow scdop$
$mode := new\_mode$

where $new\_mode =$

ALU	**if** $opcode \in ALU \cup SET$
MEM_ADDR	**if** $opcode \in LOAD \cup STORE$
IAR	**if** $opcode \in \{MOVS2I, MOVI2S\}$
$JUMPS$	**if** $opcode \in JUMP \cup BRANCH$

$\boxed{\text{WRITE\_BACK}}$

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

$\boxed{\text{Pass\_B\_to\_MDR}}$

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

$\boxed{\text{LOAD}}$

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

$\boxed{\text{TRAP}}$

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

$\boxed{\text{JUMP}}$

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

$\boxed{\text{MOVI2S}}$

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

C Rules of DLX^p

IF

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

where $jumps = opcode\ (IR1) \in JUMP$ **or** $(opcode\ (IR1) \in BRANCH$ **and** $reg\ (A) = 0)$.

ID

$\boxed{\text{Preserv IR}}$
$IR1 \leftarrow IR$

$\boxed{\text{Preserv PC}}$ $PC1 \leftarrow next\ (next\ (PC))$

$\boxed{\text{OPERAND}}$
$A \leftarrow fstop,\ B \leftarrow scdop$

EX

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

$\boxed{\text{Preserv IR1}}$
$IR2 \leftarrow IR1$

$\boxed{\text{MEM\_ADDR}}$
if $opcode\ (IR1) \in LOAD \cup STORE$
then $MAR \leftarrow A + ival\ (IR1)$

$\boxed{\text{Pass\_B\_to\_MDR}}$
if $opcode\ (IR1) \in STORE$
then $SMDR \leftarrow B$

$\boxed{\text{MOVS2I}}$
if $opcode\ (IR1) = MOVS2I$
then $C \leftarrow IAR$

$\boxed{\text{MOVI2S}}$
if $opcode\ (IR1) = MOVI2S$
then $IAR \leftarrow A$

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

$\boxed{\text{TRAP}}$
if $opcode\ (IR1) = TRAP$
then $IAR \leftarrow PC1$
$PC \leftarrow ival\ (IR1)$

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

$\boxed{\text{LINK}}$
if $opcode\ (IR1) \in JLINK$
then $C \leftarrow PC1$

MEM

$\boxed{\text{STORE}}$ **if** $opcode\ (IR2) \in STORE$
then $mem\ (MAR) \leftarrow SMDR$

$\boxed{\text{LOAD}}$ **if** $opcode\ (IR2) \in LOAD$
then $LMDR \leftarrow mem\ (MAR)$

$\boxed{\text{Preserv C}}$ $C1 \leftarrow C$

$\boxed{\text{Preserv IR2}}$ $IR3 \leftarrow IR2$

WB

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

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

D Rules of DLX^{data}

IF

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

where $jumps = opcode\ (IR1) \in JUMP$
  or $(opcode\ (IR1) \in BRANCH$ **and** $reg\ (A) = 0)$.
and $load\_risk = opcode\ (IR2) \in LOAD$
  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

$\boxed{\text{Preserv IR}}$
if *not load_risk*
then $IR1 \leftarrow IR$

$\boxed{\text{Preserv PC}}$
if *not load_risk*
then $PC1 \leftarrow next(next(PC))$

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

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

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

EX

$\boxed{\text{ALU}}$
if *not load_risk*
 then if $opcode\ (IR1) \in ALU \cup SET$
  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 $C \leftarrow \overline{opcode\ (IR1)}\ (A,\ B)$

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

$\boxed{\text{MEM\_ADDR}}$
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 $MAR \leftarrow A + ival\ (IR1)$

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

$\boxed{\text{MOVI2S}}$
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 $IAR \leftarrow A$

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

$\boxed{\text{Preserv IR1}}$ **if** *not load_risk* **then** $IR2 \leftarrow IR1$

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

$\boxed{\text{JUMP}}$
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 $PC \leftarrow A$

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

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

MEM

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

$\boxed{\text{LOAD}}$,
$\boxed{\text{Preserv IR2}}$,
$\boxed{\text{Preserv C}}$
as in DLX^p

WB

$\boxed{\text{WRITE\_BACK}}$ *as in* DLX^p

E Rules of DLX^{ctrl}

IF

$\boxed{\text{FETCH}}$ **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 $pc\_risk = opcode\ (IR) \in JUMP \cup BRANCH$
and $load\_risk =$ defined as in DLX^{data}.

ID

$\boxed{\text{Preserv IR}}$ $\boxed{\text{OPERAND}}$ *as in* DLX^{data}

$\boxed{\text{Preserv PC}}$ **if** *not load_risk* **then** $PC1 \leftarrow PC$

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

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

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

EX

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

Other rules as in DLX^{data}.

MEM

$\boxed{\text{STORE}}$ $\boxed{\text{LOAD}}$ $\boxed{\text{Preserv C}}$ $\boxed{\text{Preserv IR2}}$ as in DLX^{data}

WB

$\boxed{\text{WRITE\_BACK}}$ as in DLX^{data}

F Rules of DLX^{pipe}

IF

$\boxed{\text{FETCH}}$ **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 $IR2 \leftarrow undef$

where

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

$\boxed{\text{Preserv IR}}$
if *not load_risk* **and** *not pc_data_risk*
then $IR1 \leftarrow IR$

$\boxed{\text{Preserv PC}}$
if *not load_risk* **and** *not pc_data_risk*
then $PC1 \leftarrow PC$

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

$\boxed{\text{TRAP}_{PC}}$
if *not load_risk* **and** *not pc_data_risk*
then if $opcode\ (IR) = TRAP$
  then $PC \leftarrow ival\ (IR)$

$\boxed{\text{BRANCH}}$ **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 $PC \leftarrow PC +_{PC}\ ival\ (IR)$

$\boxed{\text{JUMP}}$ **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 $PC \leftarrow fstop\ (IR)$

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

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

EX

$\boxed{\text{ALU}}$

if $not\ load\_risk$
then if $opcode\ (IR1) \in ALU \cup SET$
  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 $C \leftarrow \overline{opcode\ (IR1)}\ (A,\ B)$

$\boxed{\text{MEM\_ADDR}}$

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 $MAR \leftarrow A + ival\ (IR1)$

$\boxed{\text{Pass\_B\_to\_MDR}}$

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

$\boxed{\text{MOVI2S}}$

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 $IAR \leftarrow A$

$\boxed{\text{MOVS2I}}$ **if** $not\ load\_risk$
  then if $opcode\ (IR1) = MOVS2I$
    then $C \leftarrow IAR$

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

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

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

$\boxed{\text{Preserv IR1}}$
if *not_load_risk*
then $IR2 \leftarrow IR1$

MEM

$\boxed{\text{Preserv IR2}}$ $IR3 \leftarrow IR2$ $\boxed{\text{Preserv C}}$ $C1 \leftarrow C$

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

$\boxed{\mathrm{LOAD}}$
if $opcode\ (IR2) \in LOAD$
then $LMDR \leftarrow mem\ (MAR)$

WB

$\boxed{\mathrm{WRITE\_BACK}}$
if $opcode\ (IR3) \in ALU \cup SET \cup \{MOVS2I\} \cup JLINK$
then $dest\ (IR3) \leftarrow C1$

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

G Signature and Pictures

Functions

$$mode \in \{FETCH, ALU, ALU', MEM\_ADDR, IAR, JUMPS, MEM\_ACC, WRITE\_BACK, Pass\_B\_to\_MDR, SUBWORD\}$$

$$\begin{array}{llcl}
reg: & REGISTER & \rightarrow & WORD \\
next: & INSTR_ADDR & \rightarrow & INSTR_ADDR \\
mem: & ADDRESS & \rightarrow & WORD \\
 & \multicolumn{3}{l}{\textbf{constraint}: mem\ (INSTR_ADDR) \in INSTRUCTION} \\
opcode: & INSTRUCTION & \rightarrow & OPERATION \\
dest: & INSTRUCTION & \rightarrow & REGISTER \qquad (dest\ (JLINK) = R31) \\
fstop: & INSTRUCTION & \rightarrow & REGISTER \\
scdop: & INSTRUCTION & \rightarrow & REGISTER \\
addr: & INSTRUCTION & \rightarrow & ADDRESS \\
iop: & OPERATION & \rightarrow & BOOLEAN \\
ival: & INSTRUCTION & \rightarrow & WORD
\end{array}$$

$$\overline{opcode\ (reg\ (IR))}: \ WORD \times WORD \rightarrow WORD \quad for\ reg\ (IR) \in ALU \cup SET$$

$$\overline{opcode\ (reg\ (IR))}: \ WORD \rightarrow SUBWORD \quad for\ reg\ (IR) \in LOAD$$

Universes

$$REGISTER = \begin{cases} REGISTER\ FILE \cup & in\ DLX \\ \{PC, IR, A, B, C, TEMP, IAR, MAR, MDR\} & \\ & \\ REGISTER\ FILE \cup & otherwise \\ \{PC, PC1, IR, IR1, IR2, IR3, A, B, C, C1, & \\ \ TEMP, IAR, MAR, SMDR, LMDR\} & \end{cases}$$

$$\begin{array}{ll}
BOOLEAN & \\
WORD & \\
ADDRESS & (\subset WORD) \\
INSTR_ADDR & (\subset ADDRESS),\ \textbf{constraint}: reg\ (PC) \in INSTR_ADDR \\
INSTRUCTION & \textbf{constraint}: INSTRUCTION \subset WORD \\
OPERATION & union\ of: \\
 & ALU, SET, LOAD, STORE, MOVESPECIAL, JUMP, BRANCH
\end{array}$$

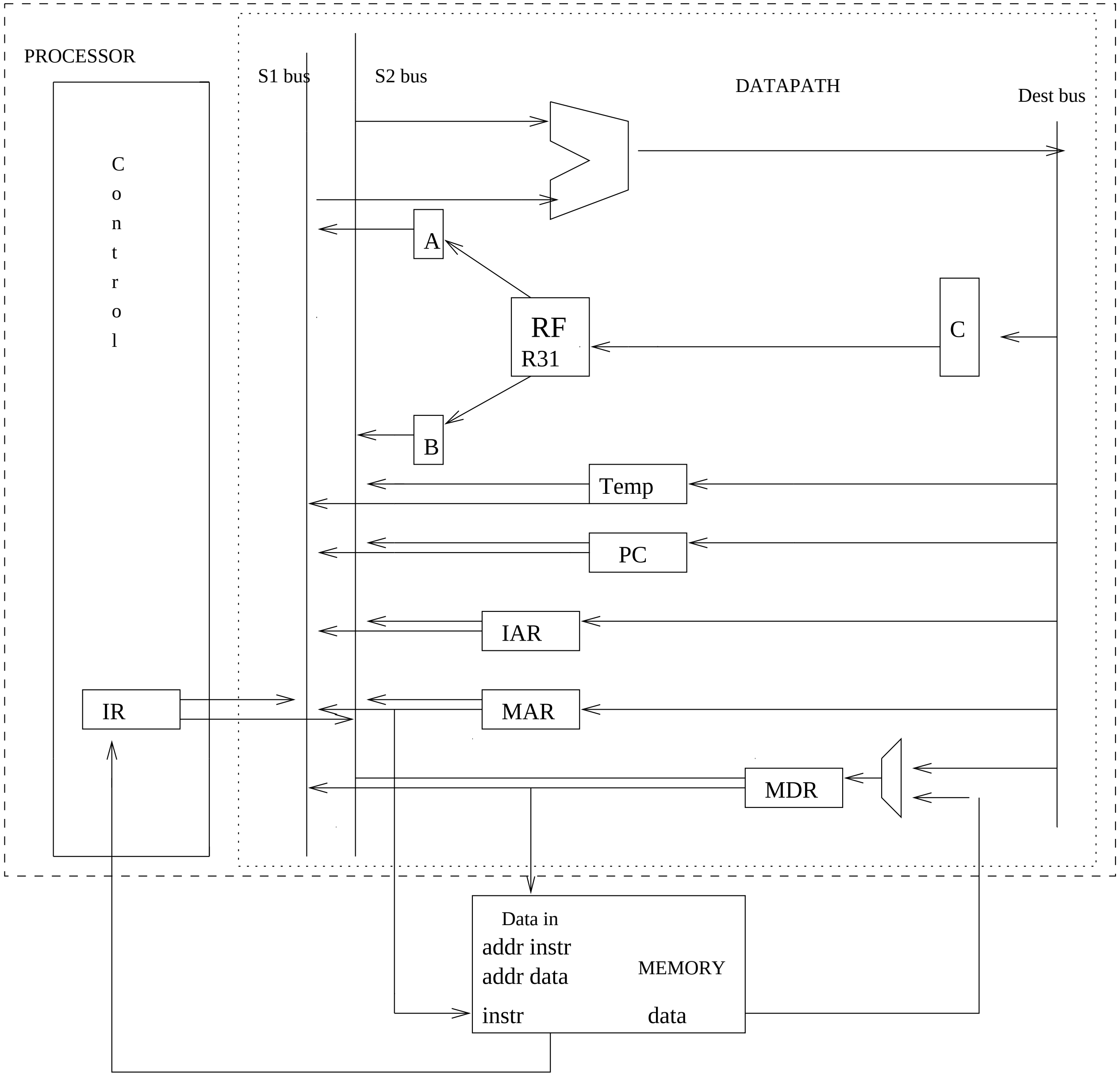


Figure 1: DLX architecture for sequential execution.

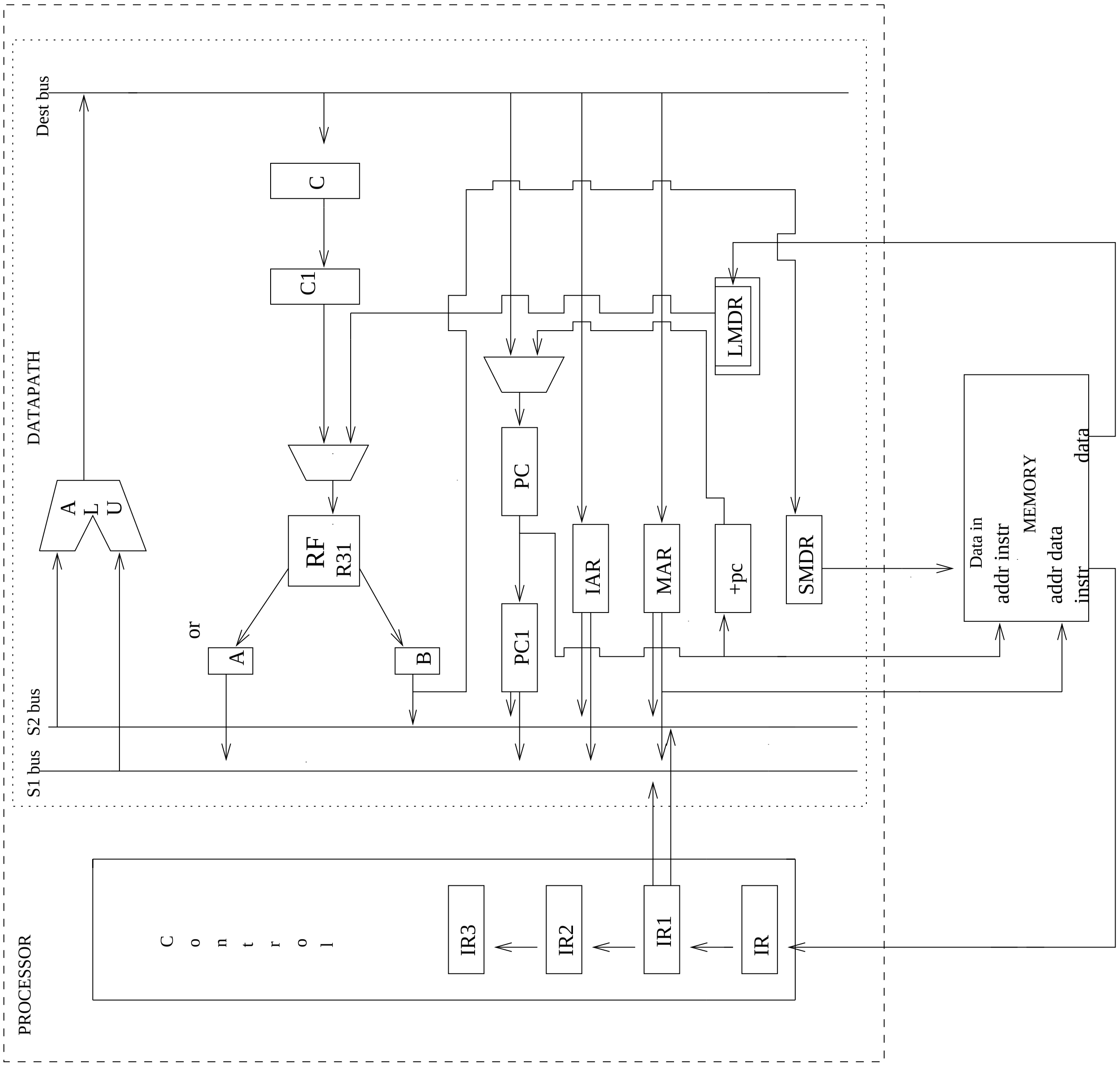


Figure 2: DLX^p architecture for pipelined execution (avoiding structural conflicts).

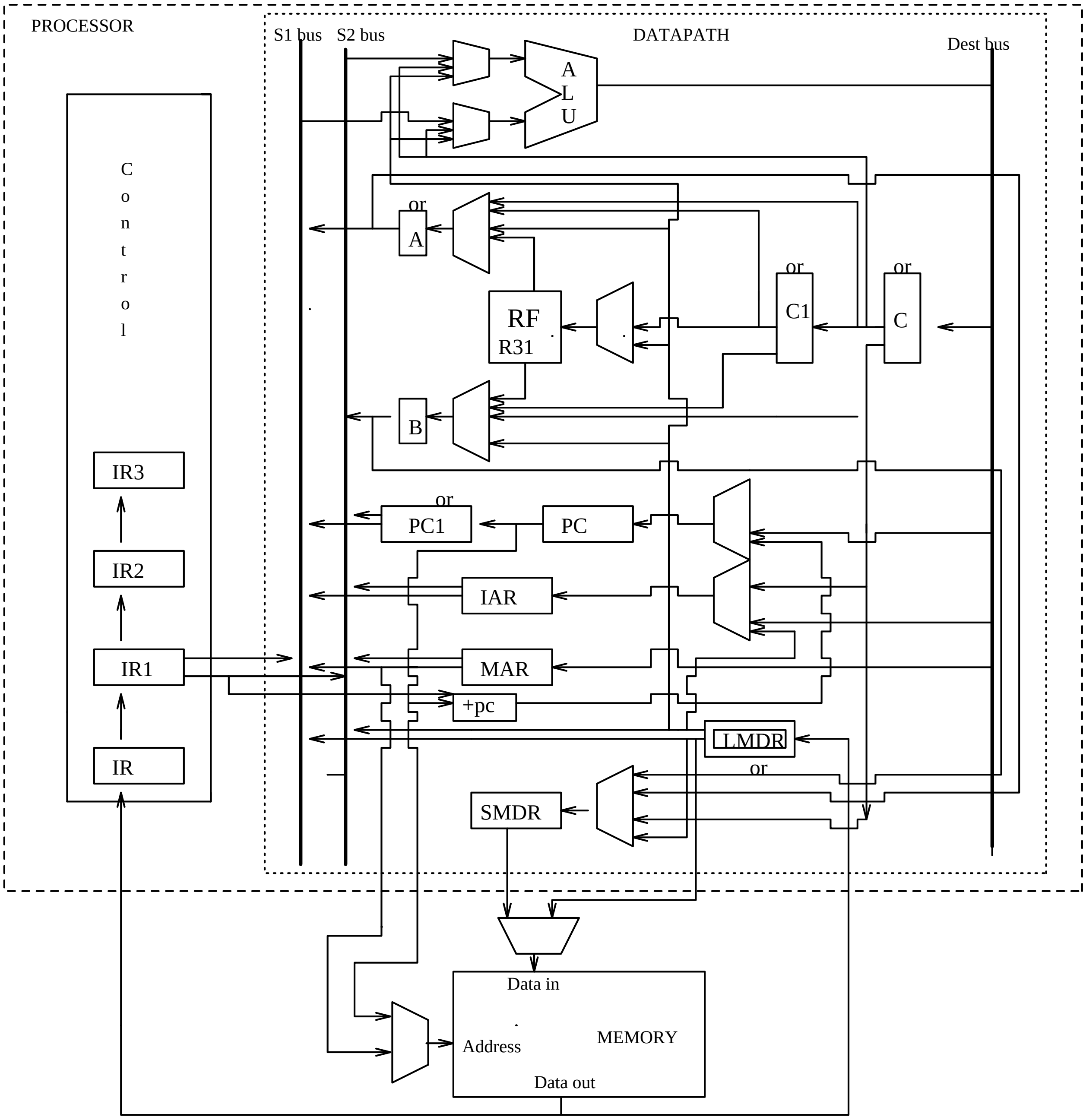
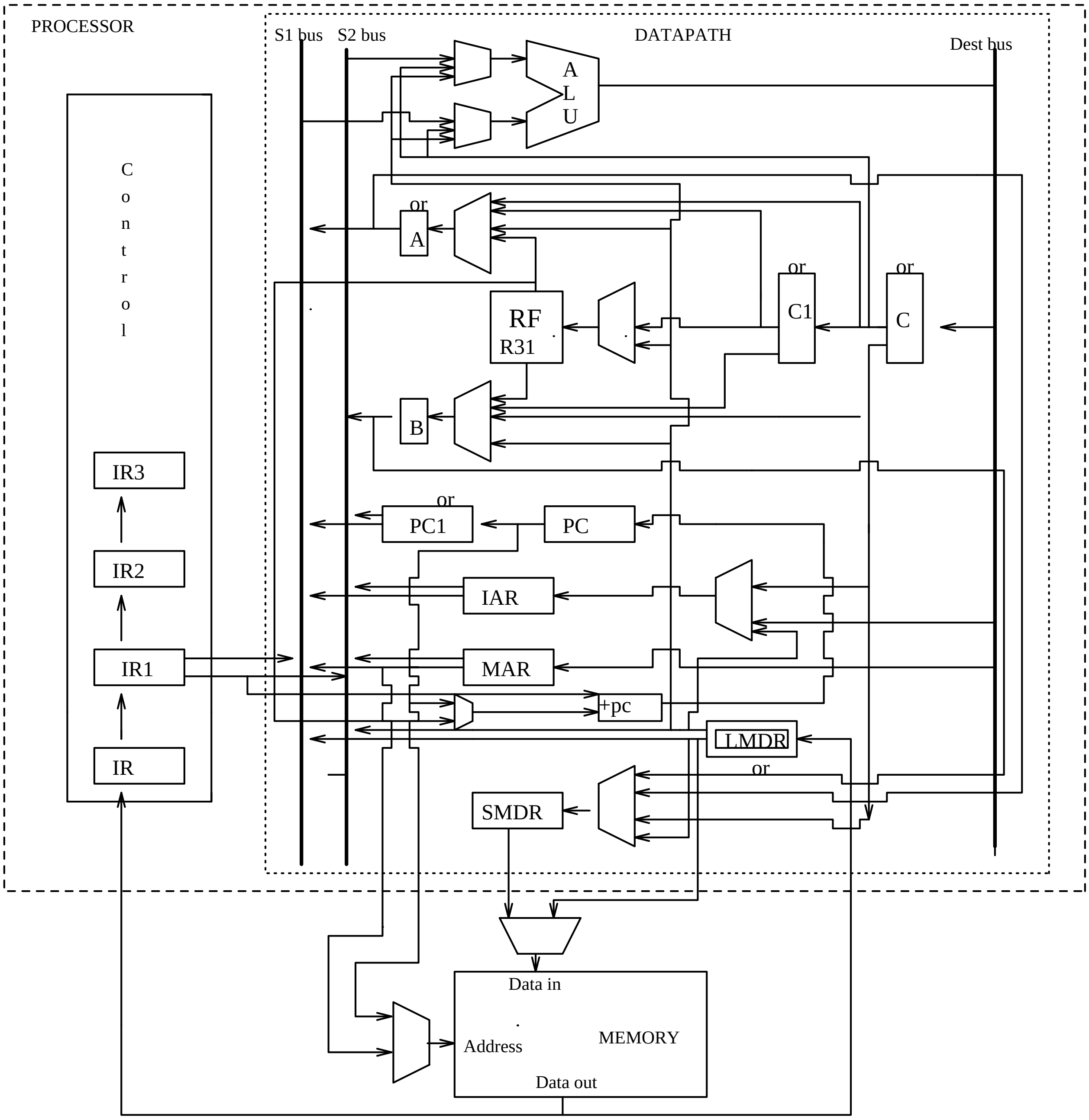


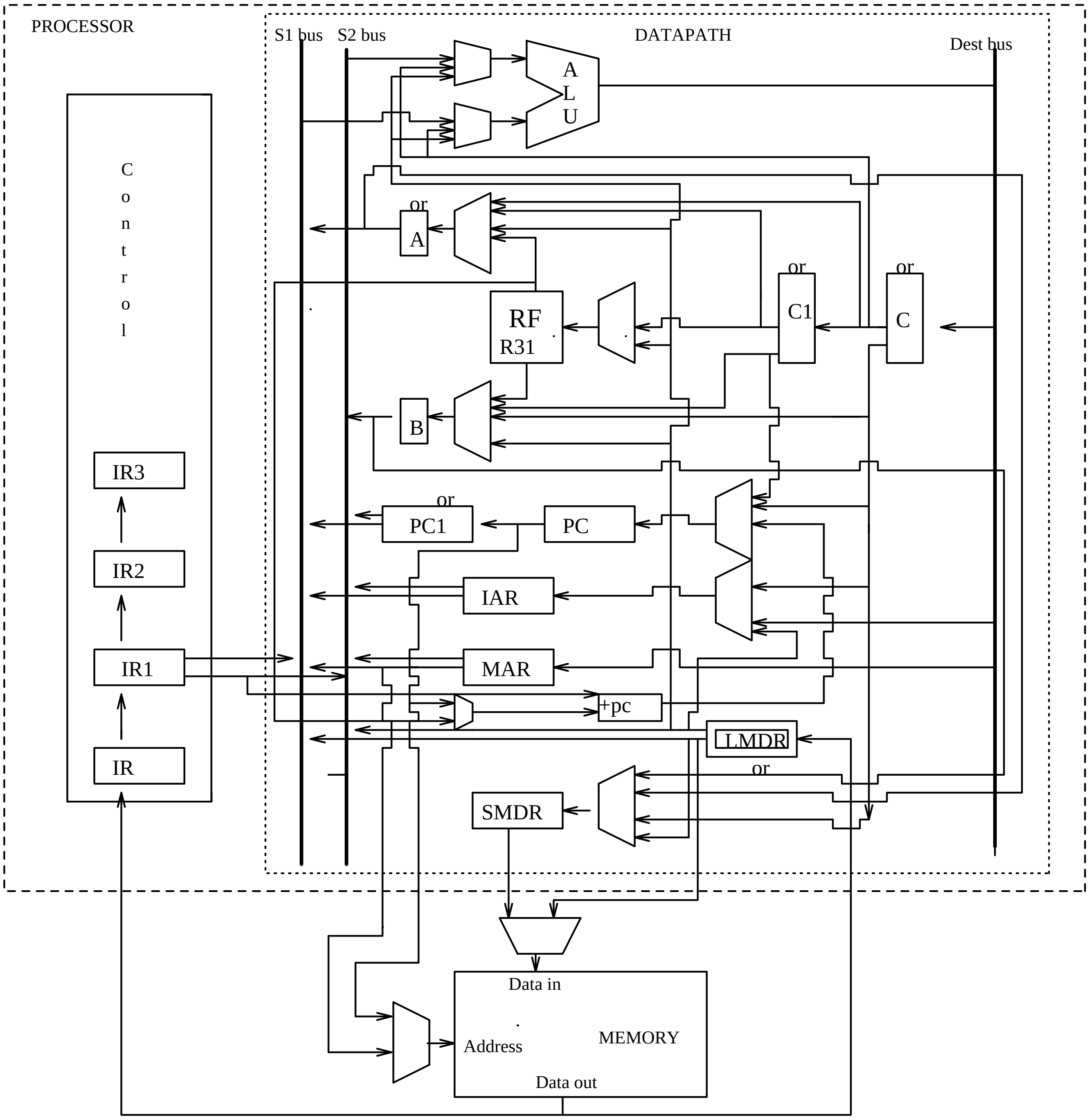
Figure 3: DLX^{data} architecture.



IR - instruction register
PC - program counter

IAR - iterrupt address register
MAR - memory address register

Figure 4: DLX^{ctrl} architecture.



IR - instruction register
PC - program counter
IAR - iterrupt address register
MAR - memory address register

Figure 5: DLX^{pipe} architecture.

Contents

References

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

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

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

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

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

[Be93] D.L.Beatty. *A Methodology for Formal Hardware Verification, with Application to Microprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, Aug. 1993.

[Bo95] E. Börger. *Why use evolving algebras for hardware and software engineering.* in: M. Bartosek, J. Staudek, J. Wiedermann (Eds), SOFSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics, Springer Lecture Notes In Computer Science, vol. 1012, 1995, pp.236–271.

[BD95] E. Börger and I. Durdanović. *Correctness of Compiling Occam to Transputer Code.* in: Computer Journal 1996 (to appear). Preliminary version in: Yuri Gurevich and Egon Börger, "Evolving Algebras. Mini-Course", Technical Report BRICS-NS-95-4, BRICS, University of Aarhus, July, 1995.

[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 on *Engineering of Complex Computer Systems* (ICECCS'95). IEEE Computer Society Press, Los Alamitos, California, 1995, pp.145-148.

[BR95] E. Börger and D. Rosenzweig. *The WAM - Definition and Compiler Correctness.* in: *Logic Programming: Formal Methods and Pratical Applications* (C.Beierle, L.Plümer, Eds.), Elsevier Science B.V. North Holland, Series in Computer Science and Artificial Intelligence, 1995, pp. 20-90 (chapter 2).

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

[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. Also published as University of Cambridge Computer Laboratory Technical Report No. 134.

[Cy93] D.Cyrluk. *Microprocessor verification in PVS: A methodology and simple example.* Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 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.* Diplomarbeit, Inst. f. Technik der Informationsverarbeitung, Universität, Karlsruhe, August 1994.

[DeTa94] M. Dehof, 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, 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.* Technical Report 100, Computer Lab. University of Cambridge, 1986.

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

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

[M95] S. Mazzanti. *Algebre dinamiche per il DLX.* Tesi di Laurea, Pisa, 1995.

[P96] C.Pusch. *Verification of Compiler Correctness for the WAM.* Manuscript, Technical University Munich, 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.* Technical Report 78, DEC System Research Center, Sept. 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, Duesseldorf 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.